DOI: 10.2298/CSIS110302054L

Quantitative Analysis for Symbolic Heap Bounds of CPS Software

Renjian Li[1], Ji Wang[1], Liqian Chen[1], Wanwei Liu[2], Dengping Wei[2]

[1] National Laboratory for Parallel and Distributed Processing
410073 Changsha, China
[2] School of Computer, National University of Defense Technology
410073 Changsha, China
{li.renjian@gmail.com, wj@nudt.edu.cn, lqchen@nudt.edu.cn,
wwliu@nudt.edu.cn, dpwei@nudt.edu.cn}

Abstract. One important quantitative property of CPS (Cyber-Physical Systems) software is its heap bound for which a precise analysis result needs to combine shape analysis and numeric reasoning. In this paper, we present a framework for statically finding symbolic heap bounds of CPS software. The basic idea is to separate numeric reasoning from shape analysis by first constructing an ASTG (Abstract State Transition Graph) and then extracting a pure numeric representation which can further be analyzed for the heap bounds. A quantitative shape analysis method based on symbolic execution is defined in the framework to generate the ASTG. The numeric representation is extracted based on program slicing technique and inputted into an abstract interpretation tool for computing the heap bounds. We take list manipulating programs as an example to explain how to instantiate the framework for important data structures and to exhibit its practicability. A novel list abstraction method is also presented to support the instantiation of the framework.

Keywords: CPS software, heap bounds, quantitative shape analysis, symbolic execution, program slicing.

1. Introduction

Conformance with quantitative constraints over temporal-spatial resources (such as execution time, energy, memory, etc.) is central to the correctness of CPS software. Compared with general purpose software, CPS software often suffers from very limited memory [1, 6, 8]. One of the most important quantitative properties of CPS software should be its heap bounds.

CPS software often adopts dynamic memory allocation schemes, where a program can at any time request the operating system to allocate additional memory from heap. The failure of dynamic memory allocation request may cause the failure of CPS software or even the whole CPS system. Usually depending on environmental parameters and/or user inputs, symbolic heap bounds are extremely important for CPS software which tends to feature a tight coupling between physical and software components and runs in open environments. Besides, precise symbolic heap bounds could also be very useful for inter-procedural static analysis and hardware synthesis [2].

There are several obstacles for finding precise heap bounds of CPS software written in imperative languages like C. Firstly, loops and recursive procedures are what make heap usage exceed its bounds. However, finding loop bounds may be difficult even for numeric programs and harder when loop bounds depend on the shape of the heap. Secondly, both shape analysis and numeric computation are needed for finding heap bounds. However, a casual combination of these techniques should involve a large increase in complexity, both in terms of the verification problem and the implementation [3]. Last but not the least, programmers often adopt shared mutable data structures, such as trees and lists, to develop CPS software for the sake of effectiveness and convenience. However, none of the available heap bounds analysis techniques can handle these shared mutable data structures full automatically.

In this paper, we try to tackle these obstacles and present a novel framework for analyzing heap bounds of CPS software. The basic idea is to separate numeric reasoning from shape analysis and to make full use of existed static analysis techniques and tools for finding precise heap bounds. In detailed, the framework will first construct an ASTG via quantitative shape analysis based on symbolic execution [4]. The ASTG is employed as an intermediate representation during the analysis and the transformation. A numeric representation maintaining the heap usage properties of the original program is further extracted based on the main idea of program slicing. The abstract interpretation tool Interproc [24] is finally used to find the heap bounds.

The framework can be instantiated for various data structures manipulating programs. In order to explain how the framework should work, we take list manipulating programs as an example. A new list abstraction model which maintains both shape and quantitative properties is presented and used during instantiating the framework. The new list abstract model stores the relationship between variables and list nodes in a singly-linked list implicitly, and represents list states in a compact manner. Compared with other abstraction models for list, such as shape graph and separation logic, it enjoys lower space overhead and higher implementation efficiency.

This paper has several main technical contributions:

– We present a new framework for analyzing heap bounds of CPS software. It separates numeric reasoning from shape analysis by extracting a numeric representation which maintains the heap usage of the original program.
– We further show how the framework could be instantiated for important data structures taking list manipulating programs as an example. With proper modifications and extensions, the framework should also work for

programs containing more complex data structures such as circular lists, doubly-linked lists, etc.

– We present a novel quantitative shape analysis method based on symbolic execution. It generates an ASTG (Abstract State Transition Graph) and is more precise than classic shape analysis methods.

The paper is organized as follows. Section 2 presents the related work. Section 3 explains our main idea through a simple example. Section 4 presents the framework for analyzing heap bounds of CPS software. In Section 5 we introduce how to instantiate the framework for programs manipulating lists. Section 6 presents the experimental results. Section 7 makes a conclusion.

2. Related Work

Quantitative properties of CPS software have gained a lot of attention within the past several years, as shown by the recent publications on the subject [7-10]. But they mainly focus on the WCET problem, while we try to find heap bounds of CPS software in this paper.

Early work for heap bounds analysis and verification [11-13] mostly focuses on functional programs where data structures are basically immutable and easier to handle. These works often needn't treat shape or the shared mutable data structures.

For imperative Object Oriented programming languages such as Java, the method proposed in [14] relies on a type system and type annotations. It is therefore up to the programmer to annotate the sizes of data structures and the amount of heap memory required for each method. Hofmann et al. [15] also propose a type based heap space analysis for Java style OO programs with explicit deallocation. It uses an amortised analysis and calculates heap memory usage with an LP-solver based on function inputs during the type inference. Albert et al. introduce a Java memory-bounds tool in [16]. It uses a heap abstraction and applies heuristics based on arithmetic simplification to find a memory bound.

For assembly-level programs, Chin et al. [25] present a method to find memory resource bounds for each method in terms of the symbolic values of its parameters. However, the system does not handle shared objects.

Different from previous work [14-16, 25], we focus on the C language which is found in many critical CPS software implementations. Finding heap bounds for C programs needs both quantitative shape analysis and numeric reasoning. Previous work often omitted shape analysis; while our method uses a more precise shape abstraction, which is crucial for dealing with our examples.

He et al. [17] try to reuse 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. They can verify quite a number of programs that cannot be handled by previous approaches, such as doubly linked lists, cyclic linked lists and binary trees. However, they need to supply memory specifications for the programs manually, while our framework could find heap bounds automatically.

Cook et al. [18] present a constraint-based method to find symbolic bounds for C programs combining several known methods and tools. They use the shape analysis tool THOR [20] to produce a new program without heap operations and use constraint-based techniques to find the heap bounds. Magill et al. [19] present a formal system for producing numeric abstractions of heap-manipulating programs based on the work of [18, 20, 21]. Our quantitative shape analysis procedure is based on symbolic execution techniques and that is different from THOR which is based on separation logic invariants generation. Another key difference between their method and ours should be the abstract model for list. Their work uses separation logic to model the abstract list state, while our work adopts the newly presented list abstract model. By focusing on specific data structures, our framework is able to obtain more precise results than their work while without have to ask the user to supply any annotations. Our numeric representation extraction algorithm is based on program slicing technique, which makes our result numeric CFG be smaller than theirs when applying to heap bounds analysis.

Shape graph is the most frequently used abstract model in static analysis; however, it can't express quantitative properties of heap. Some researchers [14, 17-20] used separation logic to describe the abstract state of list. Bouajjani et al. [22] use counter automata to model the abstract state of list. Our list abstract model has the equal expression ability with their counter automata. But our method enjoys lower space overhead and better scalability. Besides, the method in [22] is not implemented automatically; while we have implemented a prototype tool based on our list abstract model.

3. A Motivating Example

The example in figure 1 is taken from [2] with minor modifications, which may denote a frequently used programming pattern in CPS software. The procedure reads integers from an input signal `i` and returns every `n` inputted integers to an output signal `o` in inverse order. The primitive `input()` reads one integer from `i`, and the primitive `output()` writes one integer to `o`. The data structure `LIST` is used to represent singly-linked lists (with fields `data` and its `next` element). The bound of heap usage for `prio` should be `8n` (assuming that `sizeof(LIST) = 8`).

This example is fairly simple but exhibits all the obstacles we want to overcome in this paper when finding precise symbolic heap bounds of CPS software. Using the method presented in this paper, we are able to find such a bound for this example. An intermediate representation including only numeric variables will be constructed and analyzed for the heap bounds in our framework. An equivalence program of the numeric representation written in C is given in figure 2 for understanding convenience. The numeric representation may contain some variables from the original program (such as *k* and *n*) and some instrumentation variables such as *heap_now*, *heap_peak* (which track the heap usage) and X, Y (which track the quantitative properties of shape, and in this case they represent the length of lists). Now analyzing the following numeric program, we could know the biggest value of *heap_peak* is 8n, which is just the heap bound of the original program.

```
void prio(int n, in_signal i, out_signal o) {
LIST *head,*cur;
    while(1){
//  Build up an n-sized buffer
      head = (LIST*)malloc(sizeof(LIST));
      head->data = input(i);
      for(int k = 0;k<n-1;k++){
        cur = (LIST*)malloc(sizeof(LIST));
        cur->data = input(i);
        cur->next = head;
        head = cur;}
//  Send the buffer to the output and deallocate it
    cur = head;
    while(cur != NULL) {
       output(o, cur->data);
       head = cur->next;
       free(cur);
       cur = head; }}}
```

Fig. 1. A motivating example

4. A Symbolic Heap Bounds Analysis Framework

In this section, we introduce a new framework for finding symbolic heap bounds statically. The framework is presented in figure 3. After getting the CFG (Control Flow Graph) of the original program, we go forward with a quantitative shape analysis which can generate shape invariants for each program point. We do not annotate the abstract states and transitions in the original CFG, but construct a new intermediate representation named as Abstract State Transition Graph (ASTG, for short). ASTG is a core internal representation in our framework which could be used to extract the numeric representation. The final numeric representation is actually a CFG which maintains the heap usage properties of the original programs and manipulates only numeric variables. A numeric reasoning tool such as Interproc [24] could be then used to find the heap bounds. We will introduce these steps in detail in the following subsections. In this paper, we take list manipulating programs as an example for explaining the main idea of the framework. When extending to programs manipulating other kinds of data structures, firstly, you need to adopt a suitable abstract model for these data structures, and then make some proper modifications when implementing the core algorithms.

```
void prio_numeric(int n, in_signal i, out_signal o)
   {int heap_now, heap_peak, k, X, Y;
    heap_now = 0;
    heap_peak = 0;
    while(1){
      heap_now = heap_now + 8;
      if(heap_now > heap_peak)
        heap_peak = heap_now;
      k = 0;
      X = 1;
      while(1){
        if(k>=n-1)
          break;
        heap_now = heap_now + 8;
        if(heap_now > heap_peak)
          heap_peak = heap_now;
        k = k + 1;
           X = X + 1:}
       Y = X;
       while(1){
         if(Y==1)
           break;
         heap_now = heap_now - 8;
         Y = Y - 1;}
       heap_now = heap_now - 8; }}
```

Fig. 2. The numeric program tracking the heap bounds

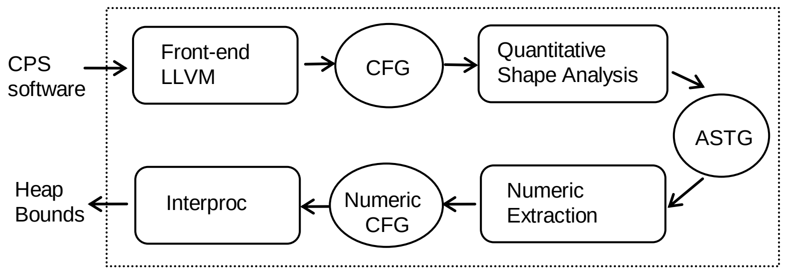


Fig. 3. Static analysis framework for symbolic heap bounds

4.1. Abstract State Transition Graph

ASTG plays an important role in our framework, so we first give its definition in this subsection.

Definition 1. An Abstract State Transition Graph (ASTG) is a 5-tuple $\langle Q,q_0,P,\rightarrow,L\rangle$, where:

- Q is a finite set of abstract states. Each $q\in Q$ is a 2-tuple $q = \langle sg,pc\rangle$ where sg is an abstract shape representation in program point pc.
- $q_0\in Q$ is the starting state.
- $P\subseteq Q$ is the set of exit states.
- $\rightarrow\subseteq Q\times Q$ is the set of transitions.
- L is a labeling function which labels each $\tau\in\rightarrow$ with program commands.

The abstract shape representation must maintain both shape properties and quantitative properties of the current shape. Supposing sg can be further divided into shape part sg^s and quantitative part sg^q. Given an abstract shape representation sg, we record it with $sg = sg^s\oplus sg^q$. However, it's obvious that these two parts may rely on each other and are not fully independent with each other. How to express the abstract shape depends on the concrete implementation and the abstract model for shared mutable data structures.

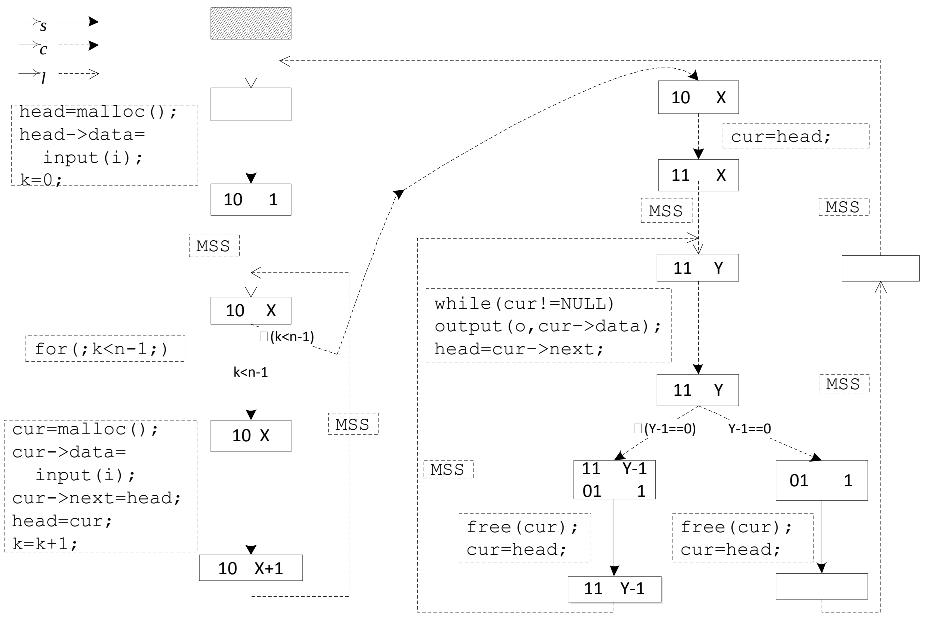
One key difference between our method and existed methods (such as [21]) is that we classified the transitions. The transitions in $\rightarrow$ could be classified into three disjoint subsets. $\rightarrow_s$ stands for the kind of transitions which are labeled with statements from the original program; $\rightarrow_c$ stands for the conditional transitions which are labeled with Boolean expressions; and $\rightarrow_l$ stands for the kind of transitions which enter a loop structure and are labeled with a special command MakeShapeSymbolic. Any $\tau\in\rightarrow$ could be treated as a transfer function which maps a source abstract state to a target abstract state.

The transitions in $\rightarrow_s$ are easily understood. Given an input state, it just generates one output state according to the semantics of the labeled program statements. It's worth noting that the definition of ASTG doesn't require the statements labeled on $\rightarrow_s$ must be assignment statements, as you can see soon from the example ASTG in figure 4.

There are some cases that a statement could generate two output abstract states. One case is when the branch condition of a branch statement could either be true or false for an input abstract state. The other case is when some special assignments might also generate two abstract states, according to the operational semantics of the abstract shape model for the underline data structures. For these two cases, we must bring in conditional transitions which are labeled with transition conditions and add them to $\rightarrow_c$.

In order to handle loop structures, we bring in a special transition for each edge entering a loop structure in the CFG and add them to $\rightarrow_l$. The target abstract state of each transition in $\rightarrow_l$ is a symbolic representation of the source abstract state. We label these transitions with a special command named MakeShapeSymbolic. It means that we should construct a new symbolic abstract state. The quantitative part sg^q of the abstract shape representation should contain only new symbolic variables.

There are some optimizations or constraints we would like to make for the transitions in ASTG in order to reduce the abstract states set Q and to simplify the implementation of our framework. As for the transitions in $\rightarrow_s$, if a continuous fragment of statements can only generate one output abstract state for each inputted abstract state, then they could be merged into a compound transition. The compound transition takes the source state of the first transition and the target state of the last transition and is labeled with the statements from all these transitions sequentially. There are some cases that the condition of a branch statement is definitely evaluated to *true* or to *false* for the input abstract state. We treat these branch statements as normal assignment transition in this case and label these branch statements with transitions in $\rightarrow_s$. The underline abstract modeling method must assure that an assignment statement should never generate more than two abstract states for the correctness of our method. Suppose the conditions labeled on the two outgoing transitions from one common source abstract states are $cond\_true$ and $cond\_false$, it must be assured that $cond\_true = \neg(cond\_false)$ and $cond\_false = \neg(cond\_true)$. As for the transitions in $\rightarrow_l$, the shape parts of the source state and the target state must be identical.



```
head=malloc();
head->data=
  input(i);
k=0;
```

```
for(;k<n-1;)
```

```
cur=malloc();
cur->data=
  input(i);
cur->next=head;
head=cur;
k=k+1;
```

```
cur=head;
```

```
while(cur!=NULL)
output(o,cur->data);
head=cur->next;
```

```
free(cur);
cur=head;
```

```
free(cur);
cur=head;
```

Fig. 4. The ASTG generated for the motivating example

The generated ASTG for the example by our framework in figure 1 is given in figure 4. Here each solid line box stands for an abstract shape sg and its position should exhibit the program counter pc. The abstract shape is expressed with our list abstract model which will be explained in the next section. You don't have to doubt why an abstract shape can be expressed like that now. The starting state q is slash marked. The program will runs forever so there is no exit states. Three kinds of transitions are denoted with different arrows as shown in the figure. The dotted line boxes positioned aside present what are labeled for each transition. The operation MakeShapeSymbolic is represented with MSS for brevity. As you can see, the ASTG describes all the abstract states that may occur during the execution of the original program. A program point of the original CFG may be separated into several abstract states with different abstract shape parts. An ASTG could be treated as the result of refining the original CFG based the shape analysis result in some ways.

4.2. Quantitative Shape Analysis

In this part we introduce how we can construct an ASTG from a CFG based on the idea of symbolic execution [4]. The algorithm is presented in figure 5. Its main idea is to start symbolic execution from the initial state and record the abstract states and the transitions that can arise during symbolic execution. Semantics of all the basic shape operations must be defined at first in order to implement the algorithm. Before explain how the algorithm works, we first define the abstract subsumption relationship between two abstract states.

Definition 2. Given two abstract states s_1,s_2, supposing $s_1 = \langle sg_1, pc_1\rangle$ and $s_2 = \langle sg_2, pc_2\rangle$, $sg_1 = sg^s{}_1 \oplus sg^q{}_1$, $sg_2 = sg^s{}_2 \oplus sg^q{}_2$. We would call s_1 is subsumed by s_2 and record with $s_1 \lhd s_2$ if and only if $pc_1 = pc_2 \wedge sg^s{}_1 = sg^s{}_2$ and $sg^q{}_2$ includes only atom symbolic variables.

The algorithm in figure 5 maintains two sets of abstract states, where NEW maintains the abstract states needed to be analyzed and OLD keeps the ones that have been analyzed. The algorithm will start symbolic execution from a selected abstract state in NEW and runs along the original CFG. When the set NEW is empty we could get the final ASTG. The method of selecting the next abstract state to analyze from NEW is not fixed and depends on the adopted search strategy. With a selected state from NEW, the algorithm will keep executing until it reaches one of the following three special cases:

- When reaching an exit abstract state, it will select another abstract state from NEW, and start a new symbolic execution process.

- When reaching a statement that may generate two possible abstract states, it will first construct a new $\rightarrow_s$ transition and two new $\rightarrow_c$ transitions. If any branched new abstract state is not subsumed by some abstract state in OLD, then a new abstract state has occurred and must be added to NEW.

- When reaching an edge which enters a loop structure in the CFG, it will check whether the state could be subsumed by some abstract state in OLD. If not, then a new abstract state has occurred and must be created with MakeShapeSymbolic command and added to NEW. Besides, a new $\rightarrow_l$

transition and a new $\rightarrow_s$ transition may also be constructed accordingly. The MakeShapeSymbolic operation means making a shape representation become a more general symbolic representation. Its concrete implementation depends on the underline abstract model for shared mutable data structures.

As an example, you can refer to figure 4 which gives an ASTG generated by the algorithm for the motivating example in figure 1.

```
Algorithm 1: QuantitativeShapeAnalysis
INPUT:   $q_0$     // the initial abstract state
         $P$       // the set of exit abstract states
         $cfg$     // the CFG of the original program
OUTPUT:  $astg = \langle OLD, q_0, P, \rightarrow_s \cup \rightarrow_c \cup \rightarrow_l, L \rangle$
         // the ASTG of the original CFG
begin
  1: $OLD = \emptyset$; $NEW = \{q\}$;
  2: while($NEW \neq \emptyset$) do
  3:   select and remove $s$ from $NEW$, add it to $OLD$;
  4:   start symbolic execution from $s$ until the following cases happen:
          // suppose the temporal abstract state before the interrupt is $s'$
  5:      In case of reaching a statement that may generate two different
          abstract states $s_1$, $s_2$:
  6:         if $s \neq s'$ then
  7:            add $\langle s, s' \rangle$ to $\rightarrow_s$, label it with corresponding statements;
  8:         add $s_1$ to $NEW$ if $\forall s_i \in OLD. \neg(s_1 \lhd s_i)$;
  9:         add $s_2$ to $NEW$ if $\forall s_i \in OLD. \neg(s_2 \lhd s_i)$;
 10:         add $\langle s', s_1 \rangle, \langle s', s_2 \rangle$ to $\rightarrow_c$, label it with corresponding conditions;
 11:         continue;
 12:      In case of reaching an edge entering a loop structure in the CFG:
 13:         if $s \neq s'$ then
 14:            add $\langle s, s' \rangle$ to $\rightarrow_s$, label it with corresponding statements;
 15:         if $\forall s_i \in OLD. \neg(s' \lhd s_i)$ then
 16:            $s''$ = MakeShapeSymbolic($s'$);
 17:            add $s''$ to $NEW$;
 18:            add $\langle s', s'' \rangle$ to $\rightarrow_l$, label it with MakeShapeSymbolic;
 19:         else   // suppose $\exists s_i \in OLD. s' \lhd s_i$
 20:            add $\langle s', s_i \rangle$ to $\rightarrow_l$, label it with MakeShapeSymbolic;
 21:         continue;
 22:      In case of $s' \in P$
 23:         continue;
 24: od
end
```

Fig. 5. The QuantitativeShapeAnalysis algorithm

4.3. Numeric Extraction

In this subsection, we will first introduce how we can model the heap usage of the original program with two numeric variables, and then introduce the main steps for extracting a numeric CFG from the ASTG.
Heap bound is a quantitative property intending to find a peak value for heap usage. Here we bring in two instrumentation variables *heap_now* and *heap_peak* which represent the heap usage at present and the peak heap usage until now respectively. There are two cases when we need to modify these two variables.

When programs call library functions such as `malloc()` to allocate some amount of memory from heap, *heap_now* should be increased by the amount of allocated heap memory. Besides, we have to determine whether *heap_now* is greater than *heap_peak*, and update *heap_peak* if it was. For each original statement `ptr=malloc(malloc_size)`, we should instrument with the following statements:

```
heap_now = heap_now + malloc_size;
if(heap_now>heap_peak)
     heap_peak = heap_now;
```

Other library functions such as `realloc()` and `calloc()` could also be handled in this way with respect to their operation semantics. We will not list them in detail.

When programs call library function `free()` to give back some amount of memory to heap, *heap_now* should be decreased by the amount of deallocated heap memory. Suppose the size of the freed memory `free_size` has been gained by a pre-analysis task, we will instrument `free(ptr)` with the following statements:

```
heap_now = heap_now - free_size;
```

We can traverse all statements labeled on the transitions of ASTG and complete the instrumentation work based on syntax analysis. The biggest value of *heap_peak* should be the heap bounds of the original program. However, besides depending on numeric program variables, *heap_peak* may also be controlled by loops and branches which may further depend on the shape of the heap, as we can see from the example in figure 1. Existed numeric reasoning tools could not be adopted directly. We will try to overcome these obstacles by constructing a pure numeric representation of the original program. The good news is that ASTG contains plentiful information for transforming these syntax structures into corresponding numeric versions.

We can transform these loops depending on the shape of the heap as following. Because we have refined the original loop structures in the quantitative shape analysis phase, all new loop structures in ASTG enjoy the good character that the shape parts of the abstract shape representations in the loop entries are identical. So the loop body can only affect the quantitative properties of the abstract states. We try to bring in new instrumentation variables to describe the change of the quantitative parts of the abstract representations. Each transition in $\rightarrow_l$ is also well designed requiring that the shape parts of the source abstract state and the target abstract state must be identical. Besides, the quantitative part of the target abstract state of a $\rightarrow_l$ transition contains only atom symbolic variables. We could take these atom symbolic variables as new numeric instrumentation variables, and assign the corresponding symbolic expressions from the quantitative part of the source abstract state of $\rightarrow_l$ to them. These assignment statements could then reflect the effect of the loop body for the abstract state.

As for the branches depending on the shape of the heap, we can replace the shape related branch conditions with equivalent quantitative properties of the shape. It's fortunately that the generated ASTG has already transformed these branch conditions into numeric versions, as you can see in figure 4. We will explain how it is possible for us to make the transformation taking lists manipulating programs as an example in the next section.

Now we can extract the statements that affect the value of *heap_peak* and construct the numeric CFG. The extraction algorithm presented in figure 6 is based on the program slicing technique [5]. Program slicing can be used to extract program statements which are relevant to a particular computation. A program *slice* is an executable program whose behavior must be identical to a specific subset of the original program's behavior. The principle of getting this behavior subset is called *slicing criterion* and can be expressed as the value of some sets of variables at some set of statements and/or program points.

The numeric CFG is a heap bounds slice of the instrumented ASTG with the initial slicing criterion including all the statements that modify *heap_peak*. The slicing procedure then starts to find and label the statements on all these edges that lead the program reaching some slice criterion based on the main idea of classic program slicing. After getting the labeled ASTG, we could construct the numeric CFG easily.

```
Algorithm 2: ExtractNumericCFG
INPUT:  astg   // the intermediate representation
OUTPUT: cfg    // the final numeric representation tracking the heap bounds
                  of the original program
begin
   1: traverse astg and instrument it with heap_now, heap_peak;
   2: traverse astg and label all transitions in $\rightarrow_l$ with corresponding
      assignment statements;
   3: add all the statements that modify heap_peak into slicing criterion;
   4: slice the ASTG and construct cfg;
end
```

Fig. 6. The ExtractNumericCFG algorithm

As an example, the extracted numeric CFG for the motivating example by our framework is presented in figure 7. Each box may stand for a combination of several basic blocks. We present it like this on purpose to reflect the main idea of the ExtractNumericCFG algorithm and for simplicity. Suppose the initial value for *heap_now* and *heap_peak* are all zero, then we could get the heap bounds $8n$ with the abstract interpretation tool Interproc now.

5. Instantiate the Framework

In this section, we illustrate how to instantiate the framework for various shared mutable data structures. List is one of the most frequently used data structures in CPS software. So we will take list as an example and present a novel abstract model for lists in the first subsection. In the second subsection we will explain some special issues needed to be considered when instantiating the framework.

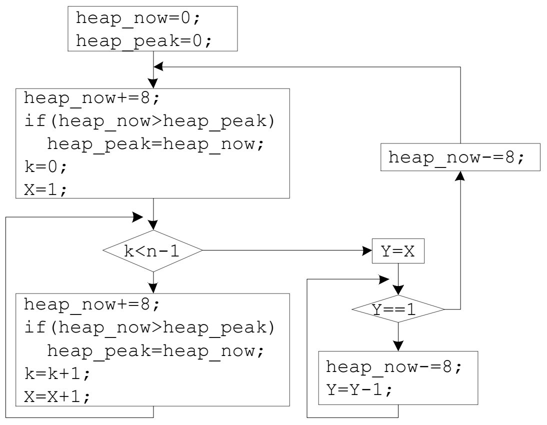


Fig. 7. The final numeric CFG for the motivating example

5.1. A New List Abstract Model

In order to express our basic idea more clearly, we focus on non-circled singly-linked lists at first. Although doubly-linked lists and circled lists are special, they can all be expressed using this abstract model with simple extensions. A singly-linked list node contains one *next* field pointing to the next list node; while all other fields can be treated as data fields. The abstract syntax considered in this paper is given in figure 8. Here *PVar* is a finite set of pointer variables of list type and *DVar* is a finite set of variables of primitive types (for simplicity, we only consider integer variables by now). Allowed syntax structures include assignment statements, branch statements, and loop statements. We suppose that there is at most one *next* operator in a list operation. All other cases could be transformed by bringing in temp variables. One example manipulating lists is presented in figure 1.

$$\begin{array}{rcl}
p, q \in PVar & & x \in DVar \\
e & := & x \mid p \to data \mid e_1 + e_2 \mid e_1 - e_2 \\
AsgnStmnt & := & x = e \mid p = null \mid p = q \mid p = q \to next \mid p \to next = q \mid \\
 & & p \to next = null \mid p = \mathrm{malloc}() \mid \mathrm{free}(p) \mid p \to data = e \\
Cond & := & p == q \mid p == null \mid e_1 > e_2 \mid e_1 < e_2 \mid \mathrm{true} \mid \mathrm{false} \\
 & & \neg Cond \mid Cond_1 \vee Cond_2 \mid Cond_1 \wedge Cond_2 \\
BranchStmnt & := & \mathbf{if}\ Cond\ \mathbf{then}\ \{Stmnt;^*\}\ [\mathbf{else}\ \{Stmnt;^*\}\ ]\ \mathbf{fi} \\
WhileStmnt & := & \mathbf{while}\ Cond\ \mathbf{do}\ \{Stmnt;^*\}\ \mathbf{od} \\
Stmnt & := & AsgnStmnt \mid BranchStmnt \mid WhileStmnt \\
Program & := & \{Stmnt;^*\}
\end{array}$$

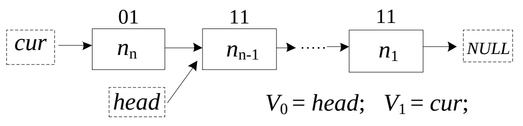
Fig. 8. The abstract syntax for operating lists

Suppose the set of list nodes is N, and the variables in *PVar* form a special subset of list nodes $V \subseteq N$. Another special node $NULL \in N$ is used to denote the *null* node. We define a binary relation E from N-$\{NULL\}$ to N-V: $\forall n_1,n_2 \in N, \langle n_1,n_2\rangle \in E$ *iff* n_1 points to list node n_2 when $n_1 \in V$ and n_2 is the *next* list node of n_1 otherwise. We record the transitive, irreflexive closure of E with E^+, and define a binary predicate $Reach(n_1,n_2)$ such that $\forall n_1,n_2 \in N$, $Reach\langle n_1,n_2\rangle$ evaluates to true *iff* $\langle n_1,n_2\rangle \in E^+$.

For the time being, we consider programs without recursion or concurrency constructs, and therefore all variables could be assumed to be global. We arrange all the variables in *PVar* in order, and for each $0 \le i \le |V|-1$, V_i stands for the ith variable. The binary predicate *Reach* describes the reachability property between list nodes in N. If $Reach(V_i,n)$ evaluates to true, then V_i could access list node n via a number of *next* operators and we would say that the variable V_i can *reach* list node n. For each list node $n \in N$-V, its reachability property for all the variables could always be expressed with a Boolean vector.

Definition 3. For each list node $n \in N$-V-$\{NULL\}$, its Variable Reachability Vector (VRV for short) γ_n is a $|V|$-sized Boolean vector $\gamma_n \in \{0,1\}^{|V|}$ where $\gamma_n[i]=1$ *iff* $Reach(V_i,n)$ evaluates to true.

Let's see an example of VRV. Suppose the example in figure 1 has just executed the statement in line 7 during the $(n$-$1)$th loop. The current list state may be like what is presented in the left part of figure 9. We denote list nodes with boxes, while denote those special nodes with boxes with dotted lines. If we define $V_0 = head$, $V_1 = cur$, then the VRVs listed on the top of each list node could describe their reachability. And as we can see, the list node n_n has just been created and pointed to by *cur*, so the VRV for n_n should be 01. The VRV for all the other list nodes should be 11 because they can all reached by both *head* and *cur*.



V_0V_1	*num*
0 1	1
1 1	n-1

Fig. 9. An example for VRV and VRVSC

Without confusing, we would also say that the variable V_i can *reach* VRV γ if $\gamma[i]=1$. Given a VRV γ, we define a set of integers $R_\gamma=\{i|\gamma[i]=1\}$which describes the variables that could reach γ. For two VRVs γ_1, γ_2, if $R_{\gamma 1}\subset R_{\gamma 2}$, then we would say γ_1 can *reach* γ_2 and record with $\gamma_1\subset\gamma_2$. Besides, for each variable V_i, we define a set of VRVs $\Gamma_i = \{\gamma|\gamma[i] = 1\}$ which contains all the VRVs that the variable V_i can reach. After defining the reachability relationships between VRVs, for each variable V_i, we can find the minimal element in Γ_i and record it with $\gamma_i^0 : \nexists\gamma\in\Gamma_i.\gamma\subset\gamma_i^0$. It's obvious that γ_i^0 must be the right VRV for the list node pointed to by V_i.

Let's see the example in figure 9 again, where $\Gamma_0 = \{11\}$, $\gamma_0^0 = 11$, so we can know that the list node pointed to by *head* has the VRV 11. Similarly, because $\Gamma_1 = \{01,11\}$, $\gamma_1^0 = 01$, we can know that the list node pointed to by *cur* has the VRV 01.

Given a list state, we can always construct a set of VRVs according to Definition 3. The relationship between these VRVs can describe the relative position of the corresponding list nodes. We can also get the VRV to which each variable points. There may exist any number of nodes with identical VRVs. Because we are only interested in the shape of heap and its quantitative properties, we can simply count the number of list nodes with identical VRV as following.

Definition 4. VRV Set with Counters (VRVSC for short) is a set of 2-tuples $VRVSC = \{\langle vrv,num\rangle\}$, where *vrv* is a VRV and *num* is the number of list nodes whose VRV equals to *vrv*.

According to the definition, all tuples in a VRVSC should be different in their *vrv*s and the *num* field for each tuple should always be greater than zero. We could always get one and only one VRVSC for each list state after defining the sequence of the variables. So VRVSC can be used as an abstract list model which maintains both the shape and quantitative properties.

For example, the VRVSC given on the right part of figure 9 could deliver the same information as the shape graph given on the left part. We could read from the VRVSC representation that there are n-1 node which could be accessed by both *head* and *cur* via a number of *next* operators, and there is one list node pointed to by *head* but not pointed to by *cur*. Also we can know the *next* node of the node pointed to by *cur* should be the same node pointed to by *head*.

The operation of data fields and primitive data types are the same as normal, so we only consider the abstract semantics for list operations. For simplicity, we use some simple recording symbols. Given a set of integers S and a VRVSC tuple $\langle vrv,num\rangle$, we use $vrv/_{S\leftarrow 0}$ to represent the operation of replacing all the bits of vrv in S with 0, meanwhile, use $vrv/_{S\leftarrow j}$ to represent the operation of replacing all the bits of vrv in S with $vrv[j]$. $new(S)$ means creating a new VRV in which all the bits in S are 1 and the other bits are 0. We also use p to represent the pth variable when not confusing.

An assignment statement can be treated as a transfer function for the abstract list model. Given a VRVSC *vrvsc*, the abstract semantics given in figure 9 describe how it can be updated according to the semantics of each assignment statement. For each assignment statement, a tuple $\langle vrv,num\rangle\in vrvsc$ may be changed only when $vrv\in\Gamma_p\cup\Gamma_q$ (or Γ_p if the assignment statement doesn't manipulate q). We will use $vrvsc_\Delta$ to represent the unmodified part of *vrvsc* in figure 10.

$$
\begin{aligned}
&[\![p = null]\!]\,(vrvsc) = \{\langle vrv', num'\rangle \mid \forall\langle vrv, num\rangle \in vrvsc \wedge vrv \in \Gamma_p.\\
&\qquad vrv' = vrv/_{\{p\}\leftarrow 0}, num' = num\} \cup vrvsc_\triangle\\
&[\![p = q]\!]\,(vrvsc) = \{\langle vrv', num'\rangle \mid \forall\langle vrv, num\rangle \in vrvsc \wedge vrv \in \Gamma_p \cup \Gamma_q.\\
&\qquad vrv' = vrv/_{\{p\}\leftarrow q}, num' = num\} \cup vrvsc_\triangle\\
&[\![p = q \rightarrow next]\!]\,(vrvsc) = \{\langle vrv', num'\rangle \mid \forall\langle vrv, num\rangle \in vrvsc \wedge vrv \in \Gamma_p \cup \Gamma_q.\\
&\qquad vrv' = vrv/_{\{p\}\leftarrow q}, num' = (vrv == \gamma_q^0)?num - 1 : num\}\\
&\qquad \cup \{\langle \gamma_q^0/_{\{p\}\leftarrow 0}, 1\rangle\} \cup vrvsc_\triangle\\
&[\![p \rightarrow next = null]\!]\,(vrvsc) = \{\langle vrv', num'\rangle \mid \forall\langle vrv, num\rangle \in vrvsc \wedge vrv \in \Gamma_p.\\
&\qquad vrv' = vrv/_{\{R_{\gamma_p^0}\}\leftarrow 0}, num' = (vrv == \gamma_p^0)?num - 1 : num\}\\
&\qquad \cup \{\langle \gamma_p^0, 1\rangle\} \cup vrvsc_\triangle\\
&[\![p \rightarrow next = q]\!]\,(vrvsc) = \{\langle vrv', num'\rangle \mid \forall\langle vrv, num\rangle \in vrvsc \wedge vrv \in \Gamma_p \cup \Gamma_q.\\
&\qquad vrv' = vrv/_{\{R_{\gamma_p^0}\}\leftarrow q}, num' = (vrv == \gamma_p^0)?num - 1 : num\}\\
&\qquad \cup \{\langle \gamma_p^0, 1\rangle\} \cup vrvsc_\triangle\\
&[\![p = malloc()]\!]\,(vrvsc) = \{\langle vrv', num'\rangle \mid \forall\langle vrv, num\rangle \in vrvsc \wedge vrv \in \Gamma_p.\\
&\qquad vrv' = vrv/_{\{i\}\leftarrow 0}, num' = num\} \cup \{\langle new(\{p\}), 1\rangle\} \cup vrvsc_\triangle\\
&[\![free(p)]\!]\,(vrvsc) = \{\langle vrv', num'\rangle \mid \forall\langle vrv, num\rangle \in vrvsc \wedge vrv \in \Gamma_p.\\
&\qquad vrv' = vrv/_{\{R_{\gamma_p^0}\}\leftarrow 0}, num' = (vrv == \gamma_p^0?)num - 1 : num\} \cup vrvsc_\triangle
\end{aligned}
$$

Fig. 10. Abstract semantics for list operations

When executing these list operations according to the abstract semantics, three cases in the output VRVSC may occur.

(1) There exist some tuples whose *num* fields are zero. The definition of VRVSC requires that all *num* fields must be greater than zero. If this case happens, we should delete these tuples from *vrvsc*.

(2) There exist several tuples whose *vrv* fields are identical. If this case happens, we should merge all these tuples into one tuple and take the sum of their *num* fields as the new *num* field.

(3) There exist some tuple whose *vrv* field is an all zero VRV. If the tuple's *num* field is also zero, then we simply delete the tuple from *vrvsc* according to the first case handling schema. However, if the *num* field is not zero, then it means some list nodes will never be accessed by any variables, so we will report a memory leak error.

We have defined a function *Compact*(*vrvsc*) to handle the above cases. The function should be called after each list operation by default.

The abstract semantics for list operations are fairly straight forward, so we won't explain in detail here, and only list them in figure 10. The detailed explanations and examples are given in Appendix A.

5.2. Some Special Issues to Be Considered

Our framework requires that the abstract model should maintain both shape properties and quantitative properties for the shared mutable data structures. VRVSC could be used as an abstract model of list meeting the above requirements. Next we show some special issues to be considered when instantiating the framework.

(1) Algorithm for checking subsumption

The quantitative shape analysis algorithm will check the subsumption relationship between two abstract states in a high frequency. So the algorithm plays an important role for improving the efficiency and extendibility of the framework. Based on Definition 2, implementing such an algorithm for the abstract list model is fairly easy.

In this case, a *vrvsc* plays the role of the abstract shape representation *sg*, the set of VRVs: $\{vrv|\langle vrv,num\rangle\in vrvsc\}$ plays the role of the shape part sg^s, and the constraints on the *num* fields play the role of the quantitative part sg^q. In order to check whether $sg^s_1 = sg^s_2$, we could iterate on the two set of VRVs, and check if they are identical. We design one practicable algorithm and present it in figure 11. The comparison of two Boolean vectors (checking whether vrv_1=vrv_2) could be implemented in a high efficiency way, so the checking subsumption algorithm could run efficiently.

(2) Implementation of the MakeShapeSymbolic Command

The MakeShapeSymbolic command constructs a more general symbolic representation. Based on the list abstract model, we can bring in a new symbolic variable for each *num* field of all the tuples in the VRVSC in the QuantitativeShapeAnalysis algorithm. When handling transitions in $\rightarrow_l$ in the ExtractNumericCFG algorithm, we could compare two VRVSCs of the source and the target abstract states, find two tuples with identical *vrv*, and assign the symbolic expressions kept in the *num* field of the source state to the symbolic variable kept in the *num* field of the target state.

```
Algorithm 3: CheckingSubsumption
INPUT: $s_1$, $s_2$     // two abstract states, supposing:
          $s_1 = \langle\{\langle vrv_1^1, num_1^1\rangle,\ldots,\langle vrv_1^i, num_1^i\rangle\}, pc_1\rangle$,
          $s_2 = \langle\{\langle vrv_2^1, num_2^1\rangle,\ldots,\langle vrv_2^j, num_2^j\rangle\}, pc_2\rangle$
OUTPUT: $yes$     // when $s_1 \lhd s_2$
        $no$      // otherwise
begin
  1:  return $no$ if $pc_1 \neq pc_2$;
  2:  return $no$ if $i \neq j$;
  3:  For each $1 \le k \le i$
  4:     subsumed = $no$;
  5:     For each $1 \le t \le j$
  6:        if $vrv_1^k == vrv_2^t$ and $num_2^t$ is an atom symbolic variable
  7:           subsumed = $yes$;
  8:           break;
  9:     if(subsumed == $no$)
  10:       return $no$;
  11: return $yes$;
end
```

Fig. 11. Algorithm for checking subsumption

(3) How to transform the shape dependent conditions to numeric conditions

In order to facilitate the ExtractNumericCFG algorithm, the abstract list model should be able to transform the shape dependent branch conditions into numeric versions. We focus on two kinds of branch conditions depending on the shape of the heap. They can both be transformed easily as following.

The first kind of conditions check whether a pointer variable is *null*. For example, $p==null$ means that all the list nodes should not be *reached* by the variable p. Given $vrvsc$, it equals to $\forall\langle vrv,num\rangle\in vrvsc.vrv[p]==0$. Considering the num fields may contain symbolic variables, we adopt another equal expression: $\forall\langle vrv,num\rangle\in vrvsc.vrv[p]\neq 0 \Rightarrow num==0$. Another kind of conditions check whether two pointer variables point to the same list node. For example, $p==q$ means that p and q should always point to the same list node. Following the above idea, we can express it with $\forall\langle vrv,num\rangle\in vrvsc.vrv[p]\neq vrv[q] \Rightarrow num==0$.

(4) How could an assignment statement become a branch statement?

A shape controlled branch may also affect the heap usage, for example we may call `malloc()` or `free()` in a shape controlled branch statement. However, when generating the ASTG, all these shape related branch conditions will be evaluated to a fixed value. So we don’t have to handle these branches specially. That’s because our framework has transferred the uncertainty of shape controlled branches to the uncertainty of some special assignment statements as following.

As we have pointed out in the previous sections, the *Compact*() operation must check if the *num* field of a tuple in VRVSC is zero. Because we adapt symbolic execution techniques during quantitative shape analysis, the underline SMT solver may report an answer *unknown* when verifying whether a symbolic expression equal 0. In this case, the *Compact*() operation may don't know whether to delete a tuple from the VRVSC. That should generate two abstract states and both of them should be treated with conservative care. In this case, the assignment statement acts like a branch statement and we should construct two transitions of $\rightarrow_c$ type to deal with this problem. Corresponding conditions with only numeric symbolic expressions are also labeled on the transitions. For example, when executing the statement `head = cur->next` in line 12 of the example presented in figure 1, we must check if the symbolic expression Y-1 equals to 0, and the adopted SMT solver will answer with *unknown*, so we add two transitions to the ASTG as shown in figure 4.

6. Experimental Results

In order to prove the practicability of our framework, we have designed and implemented a prototype tool for analyzing symbolic heap bounds of list manipulating programs statically. The prototype tool is implemented on top of the LLVM framework [27] which offers many useful facilities for the front-end analysis and the implementation of the numeric extraction algorithm. We adapt the core framework of KLEE [23] to implement the quantitative shape analysis procedure. The final numeric representation is inputted into Inerproc [24] for computing the biggest value of *heap_peak*.

We have carried our experiments for several small programs. The example given in figure1 and copy_and_delete are hand written. Hash_New_Table$_1$ and Hash_New_Table$_2$ are two Hash Table construction functions taken from the hash.c of heaplayer-0.1-benchmarks [26]. The other benchmarks are taken from [28] and can be downloaded from http://www.liafa.jussieu.fr/celia/ examples.html. Table 1 shows the statistics obtained for each analyzed program. The program size is evaluated in terms of number of lines of C code (Column 2). For each program, Column 3 represents the time for the preparation of CFG with LLVM infrastructure, Column 4 represents the time taken by quantitative shape analysis, Column 5 represents the time taken by numeric representation extraction, and Column 6 represents the time taken by Interproc to compute the biggest value for *heap_peak*. We also list the symbolic heap bounds reported by our tool and the expected results in the last two columns. Our experiments were done under Fedora 12 platform on Dual Core 1.8 GHz with 1GB main memory.

Table 1. Experimental results

Programs	Size (in lines)	Control Flow Analysis (in sec.)	Quantitative Shape Analysis (in sec.)	Numeric Extraction (in sec.)	Interproc (in sec.)	Report Result (in Bytes)	Expected Result (in Bytes)
figure 1	20	0.015	0.957	0.585	0.092	$8n$	$8n$
copy_and_ delete	26	0.007	0.736	0.523	0.070	$8n$	$8n$
Hash_New_ Table_1	18	0.009	0.688	0.424	0.290	65545328	65545328
Hash_New_ Table_2	25	0.011	0.791	0.459	0.397	81931660	81931660
intlist-lib-add	16	0.009	0.090	0.045	0.009	8	8
intlist-lib-add_tail	30	0.010	0.745	0.583	0.075	8	8
intlist-lib-init	16	0.009	0.701	0.421	0.289	$8len$	$8len$
intlist-fold-copyGe5	37	0.009	0.698	0.601	0.081	$8n$	$8n$
intlist-fold-splitV	42	0.012	0.684	0.583	0.087	$8n$	$8n$
intlist-fold2-concat	59	0.016	0.959	0.601	0.094	$8(n+m)$	$8(n+m)$
intlist-fold2-merge	90	0.019	1.219	0.893	0.138	$8(n+m)$	$8(n+m)$

Our tool reports precise heap bounds for all the programs. Although the original programs and their ASTGs vary very much, the final numeric CFGs are all very simple, so the time for running Interproc is almost the same. Another interest thing found during the experiments with Hash_New_Table() is that a first slicing before the shape analysis phase may be helpful sometimes. As our tool doesn’t handle arrays of pointers now, it can’t analyze Hash_New_Table() at first. The reason is that there exists an assignment statement for an array of pointers in the example. A first slicing can remove these assignment statements because they don’t affect the heap usage. The initial experimental results have shown that, the framework presented in the paper is practicable and the list abstraction model is effective.

7. Conclusion and Future Work

We have presented a framework for statically analyzing symbolic heap bounds of CPS software. When input CPS software, the framework will generate a numeric representation which tracks the heap usage of the original program and can further be inputted into Interproc for the heap bounds. We have taken list as an example to explain how the framework could be instantiated for shared mutable data structures. We have also presented a novel list abstraction method which maintains precise shape properties and quantitative properties. We have built a prototype tool which could analyze the heap bounds full-automatically.

As for the future work, we will first carry experiments with some more complex examples and then try to extend the framework for handling other critical data structures that may also be frequently used in CPS software such as doubly-linked lists, tree, etc.

Acknowledgments. This research is supported by the National High Technology Research and Development Program (863 Program) of China under the Grant No. 2011AA010106, and the National Natural Science Foundation of China (NSFC) under the Grant Nos. 61120106006, 90818024 and 60725206.

References

1. Xia, F., Sun, Y., Tian, Y., Tadé, M. O., Dong, J.: Fuzzy Feedback Scheduling of Resource-Constrained Embedded Control Systems. Int. J. of Innovative Computing, Information and Control, vol.5, no.2, 311-321. (2009)
2. Cook, B., Gupta, A., Magill, S., Rybalchenko, A., Simsa, J., Singh, S., Vafeiadis, V.: Finding heap-bounds for hardware synthesis. FMCAD2009, 205-212. (2009)
3. Magill, S., Tsai, M., Lee, P., Tsay, Y.: Automatic numeric abstractions for heap-manipulating programs. POPL 2010, 211-222. (2010)
4. King, J. C.: Symbolic Execution and Program Testing. Commun. ACM 19(7), 385-394. (1976).
5. Weiser, M.: Program Slicing. IEEE Trans. Software Eng. 10(4), 352-357. (1984).
6. Xia, F. Sun, Y.: Control and Scheduling Codesign: Flexible Resource Management in Real-Time Control Systems. Springer, 2008, 272 pages, ISBN: 978-3-540-78254-4. (2008)
7. Seshia, S. A., and Rakhlin, A.: Quantitative Analysis of Systems Using Game-Theoretic Learning. ACM Transactions on Embedded Computing Systems (TECS). To appear.
8. Lee, E. A., Seshia, S. A., Introduction to Embedded Systems, A Cyber-Physical Systems Approach, http://LeeSeshia.org, ISBN 978-0-557-70857-4. (2011)
9. Seshia, S. A., Kotker, J.: GameTime: A Toolkit for Timing Analysis of Software. TACAS 2011, 388-392. (2011)
10. Seshia, S. A..: Quantitative Analysis of Software: Challenges and Recent Advances. In 7th International Workshop on Formal Aspects of Component Software (2010).
11. Hofmann, M., Jost, S.: Static prediction of heap space usage for first-order functional programs. POPL 2003, 185-197. (2003)
12. Hughes, J., Pareto, L.: Recursion and Dynamic Data-structures in Bounded Space: Towards Embedded ML Programming. ICFP 1999, 70-81. (1999)
13. Unnikrishnan, L., Stoller, S. D.: Parametric heap usage analysis for functional programs. ISMM 2009, 139-148. (2009)
14. Chin, W., Nguyen, H., Qin, S., Rinard, M. C.: Memory Usage Verification for OO Programs. SAS 2005, 70-86. (2005)
15. Hofmann, M., Jost, S.: Type-Based Amortised Heap-Space Analysis. ESOP 2006, 22-37. (2006)
16. Albert, E., Genaim, S., Gómez-Zamalloa, M.: Heap space analysis for java bytecode. ISMM 2007, 105-116. (2007)
17. He, G., Qin, S., Luo, C., Chin, W.: Memory Usage Verification Using Hip/Sleek. ATVA 2009, 166-181. (2009)

18. Cook, B., Gupta, A., Magill, S., Rybalchenko, A., Simsa, J., Singh, S., Vafeiadis, V.: Finding heap-bounds for hardware synthesis. FMCAD 2009, 205-212. (2009)
19. Magill, S., Tsai, M., Lee, P., Tsay, Y.: Automatic numeric abstractions for heap-manipulating programs. POPL 2010, 211-222. (2010)
20. Magill, S., Tsai, M., Lee, P., Tsay, Y.: THOR: A Tool for Reasoning about Shape and Arithmetic. CAV 2008, 428-432. (2008)
21. Magill, S., Berdine, J., Clarke, E. M., Cook, B.: Arithmetic Strengthening for Shape Analysis. SAS 2007, 419-436. (2007)
22. Bouajjani, A., Bozga, M., Habermehl, P., Iosif, R., Moro, P., Vojnar, T.: Programs with Lists Are Counter Automata. CAV 2006, 517-531. (2006)
23. Cadar, C., Dunbar, D., Engler, D. R.: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. OSDI 2008, 209-224. (2008)
24. Lalire, G., Argoud, M., and Jeannet, B.: The interproc analyzer. http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc/index.html.
25. Chin, W., Nguyen, H., Popeea, C., Qin, S.: Analysing memory resource bounds for low-level programs. ISMM 2008, 151-160. (2008)
26. Berger, E. D., Zorn, B. G., McKinley, K. S.: Composing High-Performance Memory Allocators. PLDI 2001, 114-124. (2001)
27. Lattner, C., Adve, V. S.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. CGO 2004, 75-88. (2004)
28. Bouajjani, A., Drăgoi, C. et al. On Inter-Procedural Analysis of Programs with Lists and Data. PLDI2011. (2011)

Appendix A. The Abstract Semantics for List Operations

When explaining the abstract semantics for list operations, we will continue to use the recording symbols from section 5.1.

(1) $p = null$

After executing the assignment statement $p = null$, the variable p points to the special node $NULL$ and won't reach any list nodes, so the pth bit of the vrv should be 0. We can modify $vrvsc$ like following:

$$\llbracket p = null \rrbracket\,(vrvsc) = \{\langle vrv', num'\rangle \mid \forall \langle vrv, num\rangle \in vrvsc \wedge vrv \in \Gamma_p. \\ vrv' = vrv/_{\{p\}\leftarrow 0}, num' = num\} \cup vrvsc_{\triangle}$$

(2) $p = q$

The variables q and p will point to the same list node after the execution, so p will reach and only reach the list nodes formerly reached by q. We can construct the new abstract state by copying the qth bit of vrv to its pth bit. And $vrvsc$ can be modified like the following:

$$\llbracket p = q \rrbracket\,(vrvsc) = \{\langle vrv', num'\rangle \mid \forall \langle vrv, num\rangle \in vrvsc \wedge vrv \in \Gamma_p \cup \Gamma_q. \\ vrv' = vrv/_{\{p\}\leftarrow q}, num' = num\} \cup vrvsc_{\triangle}$$

(3) $p = q \rightarrow next$

After the assignment statement $p = q \rightarrow next$ is executed, the reachability properties of the two variables are identical for all the list nodes except the list node that q pointed to formerly. For each tuple $\langle vrv,num\rangle$ in $vrvsc$ with $vrv\in\Gamma_p\cup\Gamma_q$:

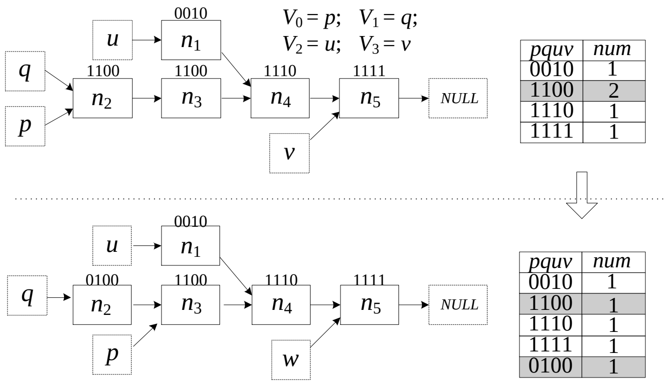
- 1. If $vrv \neq \gamma_q^0$, then we can replace the pth bit of vrv with its qth bit;

- 2. Otherwise, the corresponding list nodes could be divided into two categories. At first, because p will not reach the list node formerly pointed to by q, we can replace the pth bit of γ_q^0 with 0 and construct a new tuple with num equaling 1 to describe the list node. Secondly, for other list nodes with VRV γ_q^0, both p and q will reach them after the execution. We can replace the pth bit of vrv with its qth bit, making both the pth bit and the qth bit equal 1. But because we have excluded one list node, so the num part of the tuple should be decreased by 1.

To sum up, when executing p=$q\rightarrow next$, we can modify $vrvsc$ as following:

$$\begin{aligned}&[\![p = q \rightarrow next]\!]\,(vrvsc) = \{\langle vrv', num'\rangle \mid \forall\langle vrv, num\rangle \in vrvsc \wedge vrv \in \Gamma_p \cup \Gamma_q.\\ &\quad vrv' = vrv/_{\{p\}\leftarrow q}, num' = (vrv == \gamma_q^0)?num - 1 : num\}\\ &\quad \cup \{\langle \gamma_q^0/_{\{p\}\leftarrow 0}, 1\rangle\} \cup vrvsc_{\triangle}\end{aligned}$$

For an example, let's consider the execution in figure 12. Four variables may point to list nodes. The shape graphs are given on the left for convenient and the VRVSC are given on the right part. The gray cell in source VRVSC represents γ_q^0 and which stands for the two list nodes n_2,n_3. When executing the assignment statement p = $q\rightarrow next$, the pth bit will be replaced with the qth bit for the VRVs 1100,1110,1111 and become 1100,1110,1111 respectively, the num corresponding to γ_q^0 (1100) will be decreased by 1 and become 1.The tuple listed in the last cell stands for the newly constructed tuple by assigning the pth bit of γ_q^0 with 0 and making the num field equal 1.



pquv	num
0010	1
1100	2
1110	1
1111	1

pquv	num
0010	1
1100	1
1110	1
1111	1
0100	1

Fig.12. An example for executing p = $q\rightarrow next$

(4) $p\rightarrow next$ = $null$

After the assignment statement $p \rightarrow next = null$ is executed, for each tuple $\langle vrv,num \rangle$ in $vrvsc$ with $vrv \in \Gamma_p$:

- 1. If $vrv \neq \gamma_p^0$, then all the variables which reach γ_p^0 formerly will not reach vrv now. We can assign all the bits in $R_{\gamma_p^0}$ with 0.

- 2. If $vrv = \gamma_p^0$, then the corresponding list nodes could be divided into two categories. The VRV for the list node formerly pointed to by p should not change. In order to describe this list node, we can construct a new tuple with VRV equaling γ_p^0 and num equaling 1. As for other list nodes with VRV γ_p^0, none variables will not reach them after the execution. We can assign all the bits in $R_{\gamma_p^0}$ with 0. Because we have excluded one list node, the corresponding num should be decreased by 1.

To sum up, we can express the abstract semantics for $p \rightarrow next = null$ as following:

$$\begin{aligned}&[\![p \rightarrow next = null]\!]\,(vrvsc) = \{\langle vrv', num'\rangle \mid \forall \langle vrv, num\rangle \in vrvsc \wedge vrv \in \Gamma_p.\\ &\quad vrv' = vrv/_{\{R_{\gamma_p^0}\}\leftarrow 0}, num' = (vrv == \gamma_p^0)?num-1 : num\}\\ &\quad \cup \{\langle \gamma_p^0, 1\rangle\} \cup vrvsc_{\triangle}\end{aligned}$$

(5) $p \rightarrow next = q$

After executing the assignment statement $p \rightarrow next = q$, the reachability properties of the two variables are identical for all the list nodes except the list node that p pointed to formerly. For each tuple $\langle vrv,num \rangle$ in $vrvsc$ with $vrv \in \Gamma_p \cup \Gamma_q$:

- 1. If $vrv \in \Gamma_q$, then vrv can be reached by the variables which can reach γ_p^0 formerly. We can replace all the bits in $R_{\gamma_p^0}$ with the qth bit, making these bits equal 1;

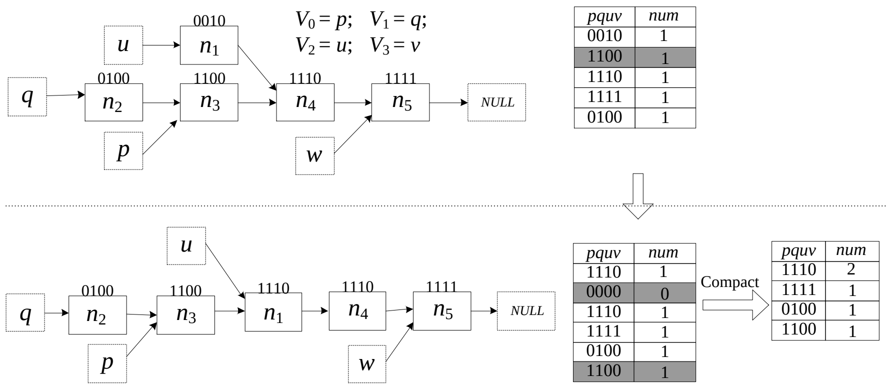
- 2. If $vrv \in \Gamma_p - \Gamma_q$ but $vrv \neq \gamma_p^0$, then vrv will not be reached by the variables which formerly reach γ_p^0. We can replace all the bits of vrv in $R_{\gamma_p^0}$ with the qth bit, making these bits equal 0;

- 3. If $vrv = \gamma_p^0$, then the corresponding list nodes could be divided into two categories. In order to describe this list node, we can construct a new tuple with VRV equaling γ_p^0 and num equaling 1. The VRV for the list node formerly pointed to by p should not change. As for other list nodes with VRV equaling γ_p^0, none variables will not reach them after the execution. Because we only consider non-circular list now, $\gamma_p^0[q]$ must equal 0. We can replace all the bits in $R_{\gamma_p^0}$ with the qth bit, making all these bits of vrv equal 0. But because we have excluded one list node, the num should be decreased by 1.

To sum up, we can express the abstract semantics for $p \rightarrow next = q$ as following:

$$[\![p \to next = q]\!]\,(vrvsc) = \{\langle vrv', num'\rangle \mid \forall \langle vrv, num\rangle \in vrvsc \wedge vrv \in \Gamma_p \cup \Gamma_q.$$
$$vrv' = vrv/_{\{R_{\gamma_p^0}\}\leftarrow q}, num' = (vrv == \gamma_p^0)?num - 1 : num\}$$
$$\cup \{\langle \gamma_p^0, 1\rangle\} \cup vrvsc_{\triangle}$$

For an example, let's consider the execution of $p \to next = u$ in figure 13. The black cell in source abstract state represents γ_p^0 which stands for the list node n_3. When executing the assignment statement $p \to next = u$, the bits in $R_{\gamma_p^0}$ ($\{p,q\}$in this case) will be replaced with the uth bit for the VRVs 0010,1100,1110,1111 generating 1110,0000,1110,1111 respectively. The *num* corresponding to γ_p^0 (1100) will be decreased by 1 and become 0. The tuple listed in the last cell describes the VRV for n_3. As we have pointed in section 5.1, a default called operation *Compact*() should be called to handle the output VRVSC after each assignment statement is executed. For this example, we should delete a tuple ⟨0000,0⟩ and join the two tuples with the same VRV 1110 as you can see in figure 13.



pquv	*num*
0010	1
1100	1
1110	1
1111	1
0100	1

pquv	*num*
1110	1
0000	0
1110	1
1111	1
0100	1
1100	1

pquv	*num*
1110	2
1111	1
0100	1
1100	1

Fig.13. An example for executing $p \to next=u$

(6) $p = malloc()$

After executing the assignment statement $p = malloc()$, the variable p will point to a new created list node and won't reach all the already existed list nodes, so the pth bit of all the *vrv* for all tuples in *vrvsc* should be 0. In order to describe the new created list node, we can create a tuple with only the pth bit of its *vrv* equaling 1 and its *num* equaling 1.:

We could express the operational semantics for $p = malloc()$ as following.

$$[\![p = malloc()]\!]\,(vrvsc) = \{\langle vrv', num'\rangle \mid \forall \langle vrv, num\rangle \in vrvsc \wedge vrv \in \Gamma_p.$$
$$vrv' = vrv/_{\{i\}\leftarrow 0}, num' = num\} \cup \{\langle new(\{p\}), 1\rangle\} \cup vrvsc_{\triangle}$$

(7) $free(p)$

After executing $free(p)$, all the variables which reach γ_p^0 formerly will never reach the VRVs in Γ_p. We could assign 0 to all the bits in $R_{\gamma_p^0}$ for all the VRVs in Γ_p. Because we have deallocated a list node, the *num* corresponding to γ_p^0 should be decreased by 1. So we can express the abstract semantics as following:

$$[\![free(p)]\!]\,(vrvsc) = \{\langle vrv', num'\rangle \mid \forall\langle vrv, num\rangle \in vrvsc \wedge vrv \in \Gamma_p. \\ vrv' = vrv/_{\{R_{\gamma_p^0}\}\leftarrow 0}, num' = (vrv == \gamma_p^0?)num - 1 : num\} \cup vrvsc_{\triangle}$$

Renjian Li received his M.S. degree in computer science from National University of Defense Technology (NUDT) in 2006. He is currently a Ph.D. candidate in School of Computer, NUDT. His research interests are in the areas of program analysis and verification.

Ji Wang received his Ph.D. degree in computer science from National University of Defense Technology in 1995. He is currently a professor at National Laboratory for Parallel and Distributed Processing of China. His research interests include high confidence software and systems, software engineering and distributed computing. He is a senior member of China Computer Federation.

Liqian Chen is an assistant professor at the School of Computer Science, National University of Defense Technology, Changsha, China. He received his PhD degree in Computer Science from National University of Defense Technology in 2010. His research interests include software engineering, program analysis and verification.

Wanwei Liu receives his Ph.D degree in Jun. 2009 at NUDT. He is currently a lecturer in School of CS, NUDT. His research interests mainly include: automata theory, logic in CS and model checking.

Dengping Wei is an assistant professor at the School of Computer Science, National University of Defense Technology, Changsha, China. She received her PhD degree in Computer Science from National University of Defense Technology in 2011. Her research interests include Semantic Web, Web service and Cyber-Physical Systems.

Received: March 2, 2011; Accepted: April 22, 2011.