THOR: A Tool for Reasoning About Shape and Arithmetic*
(Tool Paper)

Stephen Magill[1], Ming-Hsien Tsai[2], Peter Lee[1], and Yih-Kuen Tsay[2]

[1] Carnegie Mellon University
[2] National Taiwan University

Abstract. We describe Thor (**T**ool for **H**eap-**O**riented **R**easoning), a tool based on separation logic that is capable of reasoning automatically about heap-manipulating programs. There are several such systems in development now. However, Thor is unique in that it provides not only shape analysis, but also arithmetic reasoning via a novel combination procedure. Also, considerable effort has been put into making the output clear and easy to understand. Thor uses Javascript and HTML to produce an interactive representation of the analysis results.

1 Introduction

There has been a surge of interest in the use of separation logic to automatically prove memory safety. This has resulted in a number of program analysis tools that use separation logic to describe program states. One such tool is under development at Berkeley [5]. Another is the Space Invader tool [6], developed at Queen Mary's, which is able to scale to programs with over 10,000 lines of code [12]. SLAyer [2], developed at Microsoft Research, also has scalability as its primary goal and is focused on proving memory safety of large programs.

We have taken a different approach with our tool, Thor. Instead of trying to see how far we can scale a memory safety analysis, we are interested in seeing how precise an analysis we can develop while maintaining full automation. In particular, we have focused on the combination of list reasoning and arithmetic. Thor implements a shape analysis based on separation logic that is capable of reasoning about doubly-linked lists. It then adds support for arithmetic reasoning involving stack-based integers, integers in the heap, and the lengths of lists. This arithmetic support is provided by utilizing an "off-the-shelf" arithmetic analysis tool as described in [9]. This is interesting for two reasons. First, it provides very robust arithmetic reasoning, since the precision of the combination improves with the precision of the arithmetic tool being used. Secondly, the integer programs produced by the shape analysis phase provide a new source of test programs for arithmetic analysis tools. And our experiments indicate that the arithmetic programs produced by our method pose a challenge for some existing tools.

* This work was partially supported by the iCAST project sponsored by the National Science Council, Taiwan, under the Grant No. NSC96-3114-P-001-002-Y.

```
1   int i = malloc(sizeof(int));
2   List *curr = NULL;  *i = 0;
3   while(*i < n) {
4       t = (List*) malloc(sizeof(List));
5       t->next = curr;
6       curr = t;
7       *i = *i + 1;
8   }
9   free(i);  int j = 0;
10  while(j < n) {
11     t = curr->next;
12     free(curr);
13     curr = t;
14     j++;
15  }
```

Fig. 1. Example showing motivation for combined shape and arithmetic reasoning.

```
1  int a = 0;
2  int k = 0;
3  while(a < n) {
4    a = a + 1;
5    k = k + 1;
6  }
7  int j = 0;
8  while(j < n) {
9    if(k = 0)
10     goto ERROR;
11   else
12     k = k - 1;
13   j++;
14 }
```

Fig. 2. Arithmetic counterexample program produced by the shape analysis.

Figure 1 contains an example of the sort of programs that THOR is targeting. In this program, a list of length `n` is constructed and then deallocated. The deallocation code relies on the fact that the list being freed is of length `n`. Proving that this loop is memory safe involves showing that at the beginning of each loop iteration, `n - j` is the length of the still-allocated portion of the list.

Ours is not the first approach to a static analysis combining shape and arithmetic reasoning. MUTANT [3] is similar in spirit to our approach but does not output arithmetic programs and relies on weaker arithmetic invariants. Also, THOR's treatment of list lengths can be viewed as a realization of the connection between list programs and counter automata that Bouajjani et al. described in [4]. However, we believe that leveraging a separation logic-based shape analysis to provide the connection gives a more generally applicable method of going from a pointer program to an arithmetic program.

2 Interacting with THOR

THOR is a command-line program written in OCaml. It takes as input a C program and a function name. It then runs a shape analysis that proceeds by exploring the state-space of the program, symbolically executing all paths through the code. For loops, a join operator similar to that in [10] is used to ensure that the symbolic description of the program state converges to an overapproximation of the reachable states. If the program can be proved memory safe with only shape reasoning, then no further processing is required. However, if safety of the program depends on arithmetic information, as in Figure 1, then the second phase of THOR's combined analysis will be invoked.

This second phase translates the results of the shape analysis into a purely arithmetic program. This translation is such that if the arithmetic program can

be shown to be safe (where safety means non-reachability of a designated "error" location), then the original program is guaranteed to be memory safe. Viewed another way, the shape analysis implemented in THOR translates memory safety of heap-manipulating code to assertion safety of purely stack-based code. Details are given in [9], but we mention here two of the most interesting cases.

The first involves variables present only in the analysis, such as variables representing the lengths of lists. If the source program contains a loop that modifies a list during each iteration, the translation will insert a variable k into the generated program along with initialization and update statements for k that ensure this variable always tracks the length of the list. This allows the arithmetic analysis tool to discover relationships involving these length variables. We can see this occuring in the first loop in Figure 2 with the variable k.

The second interesting case involves branches on length variables. Sometimes a command is memory safe only if length variable k is positive, indicating that a certain list is non-empty. In such cases, a branch is inserted in the analysis and in the generated program that tests whether $k = 0$. In the *true* branch, we go to the error state, as this case corresponds to a memory fault. In the *false* branch, we continue exploring the state space, as this case is memory safe. This is the role of the `if` statement in the second loop in Figure 2.

By default, the arithmetic program is output as C code, so any analysis tool capable of handling C can be used to check the arithmetic program and thus prove memory safety of the original code. We have tried both BLAST [8] and ARMC [11] on the C code generated by THOR. We have also implemented an option to output the program in the FAST file format for use with that tool [1].

In addition to the arithmetic program, the tool also outputs a representation of the execution tree that was constructed while proving the program. The tree is represented as C-style source code. Branches in the analysis are represented as *if* statements and inductive invariants are represented by *gotos*.[3] Annotations are inserted between every command, which is helpful both when analyzing counter-example paths and when debugging the tool itself.

The tool can also generate graphical depictions of the program state. This is accomplished by using the Graphviz library [7] to render "box and pointer" diagrams that provide an intuitive view of the contents of memory. These graphical renderings are linked to the textual descriptions via Javascript such that by clicking on a memory state, the user can toggle between the text-based separation logic description and the Graphviz rendering. Figure 3 shows an example with the first three states described using a separation logic formula and the last two rendered as diagrams.

3 Conclusion

We have described THOR, our **T**ool for **H**eap-**O**riented **R**easoning. THOR is an implementation of the combination procedure given in [9]. It provides a means

[3] If I is an inductive invariant for the code c, then there will be a path in the output of the form `label: {I} c; goto label;`

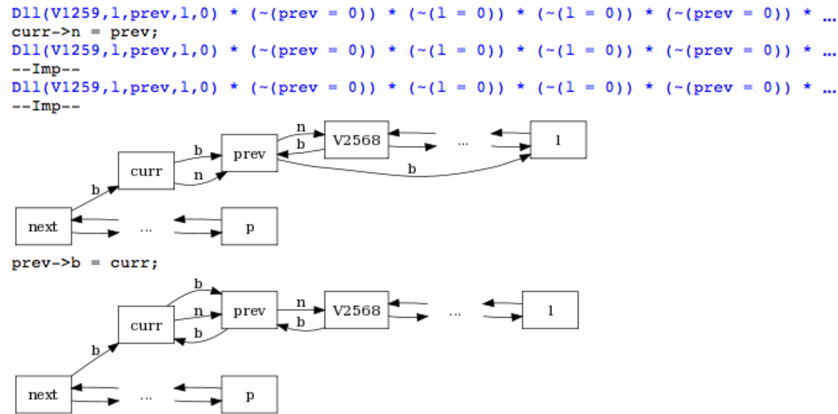

Fig. 3. Fragment of THOR's HTML output

of analyzing programs that require both shape and arithmetic reasoning and also serves as a source of interesting test programs for arithmetic tools. THOR is available for download at `http://www.cs.cmu.edu/~smagill/thor`. The distribution also includes a number of example programs.

References

1. S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. Fast: Fast acceleration of symbolic transition systems. In *CAV*, 2003.
2. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, 2007.
3. J. Berdine, B. Cook, D. Distefano, and P. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *CAV*, 2006.
4. A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with lists are counter automata. In *CAV*, 2006.
5. B.-Y. E. Chang, X. Rival, and G. C. Necula. Shape analysis with structural invariant checkers. In *SAS*, 2007.
6. D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, 2006.
7. E. Gansner and S. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11), 2000.
8. T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL'2002: Principles of Programming Languages*, pages 58–70. ACM Press, 2002.
9. S. Magill, J. Berdine, E. Clarke, and B. Cook. Arithmetic strengthening for shape analysis. In *SAS*, 2007.
10. S. Magill, A. Nanevski, E. Clarke, and P. Lee. Inferring invariants in separation logic for imperative list-processing programs. In *SPACE*, 2006.
11. A. Podelski and A. Rybalchenko. ARMC: the logical choice for software model checking with abstraction refinement. In *PADL*, 2007.
12. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. Technical report, 2008.