Tracking Heaps that Hop with Heap-Hop

Jules Villard[1] Étienne Lozes[1] Cristiano Calcagno[2,3]

[1] LSV, ENS Cachan, CNRS [2] Monoidics Ltd [3] Imperial College, London

Abstract. Heap-Hop is a program prover for concurrent heap-manipulating programs that use Hoare monitors and message-passing synchronization. Programs are annotated with pre and post-conditions and loop invariants, written in a fragment of separation logic. Communications are governed by a form of session types called *contracts*. Heap-Hop can prove safety and race-freedom and, thanks to contracts, absence of memory leaks and deadlock-freedom. It has been used in several case studies, including concurrent programs for copyless list transfer, service provider protocols, and load-balancing parallel tree disposal.

1 Introduction

Copyless message passing is an alternative to lock-based concurrency. Unlike message-passing in a context of distributed memory, copyless message passing programs can lead to efficient implementations, as only pointers to the contents of the message in memory are transferred. To avoid bugs, and in particular races, the programmer has to make sure that the ownership of the heap region representing the content of a message is lost upon sending.

Heap-Hop [1] is a program prover that checks concurrent programs that manipulate the heap, in particular list and tree structures, and synchronize using Hoare monitors and copyless message passing. Heap-Hop supports asynchronous communications on *channels*, each consisting of two endpoints which are dynamically allocated on the heap. Each endpoint can send to, and receive from, the other endpoint (its *peer*). Endpoints can be passed around as any other heap objects, and channels can be explicitly closed. Upon closure, no message should be pending for either peer, as this would result in a memory leak. Heap-Hop is based on verification conditions generation and checking, so the user only has to provide pre and post-conditions and loop invariants.

The proof system used by Heap-Hop [9] is based on separation logic [7], a logic that provides a local and modular analysis: the specification of a program p is *small*, in that it focuses on the resources actually needed by p to execute correctly, hopefully leading to concise proofs. The locality principle of separation logic is usually a strength, but for message passing it is also a weakness, since memory leaks and progress properties need to be checked on a global view of the program. To ensure these global properties, we rely on *contracts*. Contracts are a form of session types [8], or communicating finite state machines, that dictate which sequences of messages are admissible on a channel.

Heap-Hop provides strong guarantees: memory safety, meaning that the program does not fault on memory accesses; race freedom; contract obedience; and compliance

⋆ The first two authors are partially supported by the french Agence Nationale de la Recherche project PANDA, grant NR-09-BLAN-0169.

with user specifications (pre- and postconditions). Moreover, depending on the contract, Heap-Hop can also ensure deadlock-freedom and absence of memory leaks. We tested Heap-Hop on several case studies, including concurrent programs for copyless list transfer, service providers, communication protocols, and parallel tree disposal.

We first introduce the programming language and annotations with a few examples of increasing complexity, and then give some insights on Heap-Hop's internals. We conclude with some related works.

2 Heap-Hop

Programming Language. In our setting, channels are bidirectional FIFO and always consist of exactly two endpoints (e and f in the examples below). Communications are asynchronous, sending never fails, and receiving may block until the right message has arrived. The first argument of `send`/`receive` instructions is a *message identifier* which indicates what kind of message is communicated, and the second one is the endpoint that is used. Other arguments are optional depending on the number of parameters of the message. `open` and `close` respectively allocate and deallocate a channel and its two endpoints.[1] `open` takes one parameter: the contract identifier explained below.

The following program, with logical annotations in square brackets, exchanges a memory cell between two threads `put` and `get` by passing a message `cell`.

```
main() { local x,e,f; x=new(); (e,f)=open(C); put(e,x) || get(f); }
put(e,x) [e|->C{a} * x|->] { send(cell,e,x); } [e|->C{a}]
get(f) [f|->~C{a}] { y = receive(cell,f); } [f|->~C{a} * y|->]
```

The logical annotations are spatial conjunctions ($*$) of "points to" predicates that denote ownership of a cell ($x \mapsto$) or of an endpoint ($e \mapsto C\{a\}$ for contract C in state a). Notice how the ownership of the cell is transferred from the precondition of `put` to the postcondition of `get`. For Heap-Hop to accept this example, we will annotate the `cell` message with the formula $val \mapsto$,[2] to specify that the transmitted values corresponds indeed to a cell, and we will define the *contract* C for the channel (e, f).

Contracts are finite state machines that describe the protocol followed by the channel, *i.e.* which sequences of sends and receives are admissible on the channel. A contract C is written from one of the endpoints' point-of-view, the other one following the dual contract $\bar{C}$ (`~C` in source code), where sends ! and receives ? have been swapped.

Before giving a contract for the previous example, we make the example more interesting by sending e over itself after sending x, so that `get` can then close the channel (e, f). We need a second message `close_me`, whose invariant uses the special src variable, which refers to the sending endpoint, just as val refers to the sent value.

```
message cell [val|->]
message close_me [val|->C{b} * val==src]
contract C { initial state a { !cell -> a; !close_me -> b; }
             final state b {} }
```

[1] We have chosen a `close` primitive where both ends of a channel are closed together.

[2] A message can have several parameters, in which case they are referred to as $val0$, $val1$, ...

```
put(e,x) [e|->C{a} * x|->] {
  send(cell,e,x);
  send(close_me,e,e); } [emp]
```

```
get(f) [f|->~C{a}] {
  y = receive(cell,f);
  ee = receive(close_me,f);
  close(ee,f); } [y|->]
```

Notice how the postcondition of `put` is now emp (the empty heap). After the receive of `close_me`, and with the help of its invariant, Heap-Hop can prove that e and f form a channel and that they are both in the same final state, which permits the closing of (e, f). This would not be the case had we omitted $val = src$ in the invariant.

Let us give a final version of the program that sends a whole linked list starting at x (denoted by $\mathrm{list}(x)$ in the annotations) cell-by-cell through the channel. Our contract `C` already allows this: we can send an unbounded number of cells before we leave the state a. `get` cannot know anymore when the `close_me` message will come, so a `switch receive` between messages `cell` and `close_me` is used, which in general selects either message from the receive queue, whichever comes first.

```
put(e,x) [e|->C{a} * list(x)] {
  local t;
  while(x != 0)
  [e|->C{a} * list(x)] {
    t = x->tl;
    send(cell,e,x);
    x = t; }
  send(close_me,e,e); } [emp]
```

```
get(f) [f|->~C{a}] {
  local x, ee = 0;
  while(ee == 0) [(†)] {
    switch receive {
    x=receive(cell,f): {dispose(x);}
    ee=receive(close_me,f): {}
  }}
  close(ee,f); } [emp]
```

(†) $\triangleq$ `if ee==0 then f|->~C{a} else (ee|->C{b} * f|->~C{b},pr:ee)`[3]

A particularity of the copyless message passing setting is that doing the sending of the cell before dereferencing x in the example above (i.e. placing the `send(cell,e,x);` one line earlier) would result in a fault, as the cell pointed to by x is not owned by this thread anymore after it has been sent.

Usage. Heap-Hop takes annotated programs as input, and outputs a diagnosis for every function of the program: either a successful check, or an error report showing the incriminated program lines and formulas where the check failed. It also outputs a graphical representation of the contracts declared in the file.

Contracts play a fundamental role in the analysis. Heap-Hop checks whether the following three conditions hold:

Deterministic From every state of the contract, there should be at most one transition labeled by a given message name and a given direction.

Positional Every state of the contract must allow either only sends or only receives.

Synch All cycles in the contract that go through a final state must contain at least one send and one receive.

These conditions are sufficient to ensure the absence of memory leak on channel closure [9]; Heap-Hop will issue a warning if they are not met. If moreover there is only one channel used in the whole program, without Hoare monitors, and if all `switch receive` statements are exhaustive with respect to the contract, then the program is also guaranteed to be deadlock-free. Currently, Heap-Hop does not report on deadlock-freedom since we expect simpler proofs of it to be available using other methods [6].

[3] In `f|->~C{b},pr:ee`, `pr:ee` means that ee is the peer of f.

3 Internals

Heap-Hop is an extension of a previous tool called Smallfoot [2], and uses the same principles: it first converts the annotated program into verification conditions, then checks each condition by applying symbolic forward execution, eventually checking that the computed symbolic heap entails the targeted one. However, in case of non-deterministic contracts, fundamental changes are needed in the symbolic execution mechanism. Consider the following example:

```
contract ND { initial state a { !m -> b; !m -> c; }
              state b {} final state c {} }
foo() { (e,f) = open(ND); send(m,e); receive(m,f); close(e,f); }
```

Starting from state a, symbolic execution could then proceed to state b or c. Notice that only the choice of state c, which is the final state, allows to close the channel in the end, and this choice is not evident when the send is executed. For this reason, our symbolic execution mechanism explores all the possibilities in parallel, by operating on sets of symbolic heaps and pruning wrong choices along the way.

4 Related Work & Conclusion

As already mentioned, Heap-Hop is a Smallfoot extension based on a fully formalized proof theory [9]. Another extension of Smallfoot is SmallfootRG [3], that combines Separation Logic with Rely-Guarantee reasoning. Typical case studies of SmallfootRG are non-blocking concurrent algorithms, but it does not support message passing. Chalice [6] is a program prover that has been recently extended to support copyless message passing, and allows to prove deadlock-freedom using credits and lock levels. Both SmallfootRG and Chalice could encode our contracts and `switch receive` constructs, but this encoding, as well as being tedious, would be incomplete for non-deterministic contracts. SessionJ [5] and Sing# [4] are realistic programming languages that rely on contracts. The Sing# compiler uses a static analyzer to check some restricted form of copyless message-passing, but seemingly does not support ownership transfer of recursive data structures.

References

1. `http://www.lsv.ens-cachan.fr/~villard/heaphop/`.
2. J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *FMCO 2005*, volume 4111 of *LNCS*, pages 115–137, 2005.
3. C. Calcagno, M. Parkinson, and V. Vafeiadis. Modular Safety Checking for Fine-Grained Concurrency. *LNCS*, 4634:233, 2007.
4. M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language Support for Fast and Reliable Message-Based Communication in Singularity OS. In *EuroSys*, 2006.
5. Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-Based Distributed Programming in Java. In *ECOOP 2008*, pages 516–541.

6. K. Rustan M. Leino, Peter Müller, and Jan Smans. Deadlock-free channels and locks. In *ESOP*, volume 6012 of *LNCS*, pages 407–426. Springer, 2010.
7. J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS 2002*.
8. K. Takeuchi, K. Honda, and M. Kubo. An Interaction-Based Language and Its Typing System. *LNCS*, pages 398–398, 1994.
9. Jules Villard, Étienne Lozes, and Cristiano Calcagno. Proving Copyless Message Passing. In *APLAS'09*, volume 5904 of *LNCS*, pages 194–209, Seoul, Korea, 2009. Springer.