Verifying the Safety of a Practical Concurrent Garbage Collector

Georges Gonthier

INRIA Rocquencourt
78153 LE CHESNAY CEDEX
FRANCE

Abstract. We describe our experience in the mechanical verification of the safety invariants of an asynchronous garbage-collection algorithm [1], using the TLP system [2]. We only give a cursory overview of the algorithm and its formalisation. Our main focus is on the lessons learned from carrying a sizeable (22,000+ lines) formal proof through an off-the-shelf prover. In particular, we found the TLP style of structured proofs to be particularly effective for organising, writing, and managing proof scripts.

1 Motivation

If there is one kind of algorithm that warrants a mechanical proof, it probably is concurrent garbage collection. One of the main motivations for automatic garbage collection is safety from devastating, hard-to-detect memory management errors. This requires a very high degree of safety of the collection algorithm; however, such a degree is unattainable by simple testing for a concurrent collector, where errors are hard to trigger.

Therefore the prudent practice in concurrent garbage collection has been to use an array of interlocks to limit the asynchrony, thus enforcing a tamer model of concurrency in which a simple, robust algorithm has been proven (e.g., a sequential one). This caution has a price, though: the extra synchronisation is costly in terms of performance and/or of portability.

In [1] we showed that another tradeoff was possible: using rigourous formal methods, one can design a concurrent garbage collection algorithm that will perform efficiently under realistic concurrency assumptions. However, this demonstration was somewhat incomplete, since it rested only on a manual proof involving 2898 cases... Mechanical verification thus seemed the only way of making the tradeoff of algorithmic versus engineering safety worthwhile.

Furthermore, it appeared that this example could be used to exercise theorem provers in a particular manner. Mechanised proofs tend to fall in two categories: "mathematical proofs" and "certifications". In the former the object is a fragment of mathematics or an abstract algorithm (e.g., [4, 7]), the definitions are dense and layered, the proofs are heavily guided, and the main output is a better understanding of the theory at hand. In the latter, the object is a hardware/software system (e.g, [5, 8]), the definitions are very long and "flat", the proofs are highly repetitive, hence automated, and the main output is certification of the system. Of course both kinds of proofs occur in a large verification, and recently "hybrid" provers have received considerable attention.

Our collector example, however, is by itself a "hybrid" proof. It is "mathematical" in that it pertains to a short (100-line) algorithm, that it seeks to establish a simple result (for garbage collector safety, simple type correctness is convincing enough), that it involves abstract mathematics (reachability in graphs), and that a better understanding of the algorithm and its invariants was an expected output. On the other hand, our collector proof is also a "certification", because the combinatorics of concurrency blow up the proof size, and because validation of the algorithm was also an important output.

In section 2 we present the development cycle that lead us to the TLP proof. In section 3 we summarise the lessons learned during the proof itself. Section 4 discusses the directions in which this effort could be pursued.

2 The development cycle

The precise description of our algorithm and its formalisation can be found in [1]. Here we will only describe how these were developed.

The development of our algorithm can be cast in the standard "waterfall" model: requirements and architecture, then algorithm design and coding, then abstract formalisation, then formal proof, and finally mechanical verification. Note that formal methods appear here as an expansion of the "testing" stage; we did indeed use them as a debugging tool.

The best evidence that this development plan was sound is provided by the error trace. Each stage caused several major revisions of its immediate predecessor, and a few minor revisions of its grandfather, but changes never propagated more than two levels up:

- Writing down a formal model of the algorithm revealed a major synchronisation error in the algorithm design, and helped to clarify and strengthen the requirements.
- Writing down and manually checking the safety invariants revealed many errors in the model, as well a few secondary synchronisation errors in the algorithm.
- The mechanical verification uncovered a serious omission in the main collector invariant, and a few minor errors in the formal model, none of which reflected errors in the actual implementation (the model being more general).

In addition, there were some simple pragmatic facts supporting our plan:

- Since the whole point was to trade simplicity for performance, it would have been ludicrous to do a full formal analysis before implementing to check the efficiency.
- Since the invariants are about as long (100 lines), but much harder to understand than the program itself, it would have been needlessly hard to try to develop the program from the invariants.

- The invariants themselves are based on 33 definitions which in effect create an abstract view of the algorithm. These definitions involve sets, relations, and transitive closures. It is highly unlikely that they would have emerged naturally from the blind interaction with a prover.

The most crucial step was selecting the level of formalisation. The first attempt was too abstract and did not detect the error in the initial algorithm [1]; on the other hand, it was necessary to abstract from control flow and list management details to have a manageable proof. The transition-predicate approach of TLA [6] provided a convenient framework for making these tradeoffs.

3 The verification

Engberg's TLP [2] is a front-end for the LP prover [3]. It provides support for the TLA logic, for Lamport's formula list syntax, and his structured proofs, as well as a modest macro facility. The TLA support was largely irrelevant for us: the safety proof did not involve any significant temporal reasoning, so we only used the "prime" notation for next state variables.

On the other hand, the apparently trivial support for Lamport's structured proofs turned out to be crucial for the success of our effort. A TLP proof script is a sequence of prover commands and *steps*; a step is simply a formula together with its proof script, which may recursively contain substeps. A step may also introduce hypotheses which will be discharged upon exit; TLP also keeps track of any needed skolemisation. Each (sub)step is proved in its context; in particular, all previous steps and hypotheses, as well as any fact derived from them by forward inference prover commands, are available for the proof. A simple depth-based indexing scheme makes references to these local facts short and convenient.

This structure makes TLP proof scripts especially readable and robust, because they consist of a sequence of true statements, interspersed with forward inference commands (the few TLP backward inference commands turned out to be impractical). This is very close to a hand proof, so much so that we were able to write most of our 22,000 line script *off-line*, using the proof-checker only as a (sluggish) debugger!

The robustness stems from the fact that the validity of the substeps is generally independent from the prover deduction strategy and from superficial changes of the problem definition. Hence it is very easy to cut and paste pieces of scripts, or to adapt a script to a new context (a similar case has been made for Nqthm [5]). This feature turned out to be a lifesaver when we discovered a serious omission in the main collector invariant, during the proof of the last of 64 main lemmas, four months into the proof. The invariant and several definitions had to be reworked, but hardly any of the script needed any change; most substeps were still valid (albeit sometimes for different reasons!).

Many advocate the decomposition of a proof in a succession of small lemmas; however very few proof systems provide features for organising the resulting lemmata army. Our proof involves over 3,500 substeps (some of which would not have been needed with a more powerful prover than TLP, with support for typechecking, arithmetic, etc). It would have been next to impossible to spell out each of these as a separately named lemma with an explicit context. The very simple TLP indexing scheme was invaluable here.

Not everything in TLP was great, though. We had to develop some ingenuity to compensate for TLP's limitations in typechecking, higher-order rewriting, and quantifications. However, ultimately, it was the control provided by the proof structure that was decisive and that enabled us to go through with the proof.

4 Going further

Proving the termination of the collector cycle would appear to be our next logical step. However, this would give us fairly little return – livelock is rarely a problem for a concurrent garbage collector – for a proof that would be just as complex, and probably more given the weakness of LP in arithmetic.

A more ambitious project would be to layer the specification, using the invariant definitions as a refinement mapping from program (concrete) to invariant (abstract) variables. Currently the safety proof for each action consists of three parts: a proof that the action maintains some representation invariants, a proof the the program action (on concrete variables) implements a certain *abstract action* (on abstract variables), and finally a proof that the abstract action satisfies the algorithm's invariants. Layering the specification would yield a clean separation between the three parts, and would open the way for proving the correctness of an even more detailed version of the algorithm. This may become practical if a TLP-like interface is built on more powerful prover.

References

1. Doligez, D., Gonthier, G.: Portable, unobtrusive garbage collection for multiprocessor systems. ACM POPL (1994) 70–83
2. Engberg, U., Gr/onning, P., Lamport, L.: Mechanical verification of concurrent systems with TLP. LNCS 663 (CAV 1992) 44–55
3. Garland, S. J., Guttag, J. V.: An overview of LP, the Larch prover. LNCS 355 (RTA 1989) 137–151
4. Huet, G.: Residual theory in λ-calculus: a formal development. J. Func. Prog. **4** (1994) 371–394
5. Hunt, W. A. Jr., Brock, B.: A formal HDL and its use in the FM9001. Proc. Royal Soc. (1992)
6. Lamport, L.: The temporal logic of actions. ACM TOPLAS **16** (1994) 872–923
7. Lincoln, P., Rushby, J.: Formal verification of an algorithm for interactive consistency under a hybrid fault model. CAV 1993
8. Miller, S. P., Srivas, M.: Formal verification of the AAMP5 microprocessor. IEEE Workshop on Industrial-Strength Formal Spec. Techniques (1995)