Proposal for ACDC2011 Doctoral Symposium

Allan Tabilog

December 16, 2010

Student Details

- Full Name: Allan Tabilog
- Affiliation: School of Engineering and Computer Science, Victoria University of Wellington
- Address: Cotton Building, Kelburn Parade, Wellington, New Zealand
- Email Address: amtabilog@ecs.vuw.ac.nz
- Phone Number: 64272225531
- PhD Enrollment Date: March 2009 (Half-time student)

Title of thesis  Mechanising Local Rely-Guarantee Reasoning

Keywords  program verification, concurrency, theorem-proving, program logic

Research Problem  The project aims to mechanise local rely-guarantee reasoning (LRG) [3] by embedding it in the Isabelle/HOL theorem prover [6] and using the resulting formalisation to interactively verify a number of nonblocking concurrent algorithms. Nonblocking algorithms are those which do not use mutual-exclusion-based mechanisms such as locks to prevent process or thread interference, but instead allow interference in controlled ways. These are increasingly being recognised as alternatives to traditional lock-based ones that can be implemented to exploit advances in computer architectures e.g. the rise of multicore CPU’s. In principle and in practice these algorithms can perform better than lock-based ones (see e.g. [4]). However because such algorithms allow for more complex inter-process interactions, they are more difficult to understand and to verify.

The solution we propose is through deductive verification with a program logic supported by practical theorem-proving tools. We begin our study by considering LRG, which is a program logic that can be used to reason about shared-variable concurrent algorithms, and for which there is no theorem-proving support at present. LRG is a program logic that merges the features of two well-known program logics for concurrency: rely-guarantee reasoning [15] and separation logic [8, 7]. It is comparable to other formalisms in its approach, notably to a formalism called RGSep [11]. However, while the underlying theory and formal semantics of LRG appears to have been well-developed by its author, its usefulness and applicability has not been tested against a range of algorithms (nonblocking or otherwise). The only example in the literature is a verification of a simple concurrent GCD algorithm [3]. On the other hand, RGSep has been used in the verification of a wide range of algorithms, both manually on paper [10] and with the help of a tool called SmallfootRG [2], which is an extension of a tool called Smallfoot [1]. The Smallfoot verifications however only show that algorithms satisfy data structure invariants or shape invariants instead of full functional correctness.
Thus in our study we aim to evaluate the applicability of the LRG logic to the verification of full functional correctness of some nonblocking algorithms. We also aim to mechanise the logic because we believe that this is one way to make it more practical and usable. The mechanisation effort forces one to be explicit about assumptions, justifications, and proof strategies, and leads to a much greater understanding of the program logic. It also gives one a greater confidence in the soundness of the program logic, especially if one is able to construct a mechanised proof of its soundness. Moreover, the mechanised logic can be developed into a tool that can be used for interactively and to some degree, automatically, verifying the correctness of particular algorithms.

LRG has some novel features that we aim to exploit in the study:

1. It supports separating conjunction of rely/guarantee conditions. In concurrent separation logic one can only conjoin assertions over states. In RGSep one can conjoin assertions over local states or assertions over shared states, but cannot conjoin actions.

2. It supports a frame rule over rely and guarantee conditions, hence the sharing of resources no longer needs to be globally known. In classic rely-guarantee, one has to consider all possible thread interactions in the rely and guarantee conditions.

3. It supports a rule for hiding shared resources from the environment - hence resources can be shared among a subset of threads (instead of among all threads, which is the approach taken by RGSep).

4. Program states are split only conceptually into private and shared parts. Hence there is no need to make syntactic distinctions between assertions about shared and private states (as in RGSep); or semantic distinctions.

LRG provides a simple programming language with support for variables (program and logical variables), heap operations (allocation, lookup, update, and deallocation), the usual procedural constructs (skip, sequential composition, conditional, and while loop), and concurrency constructs (parallel composition and guarded atomic block). The assertion language of LRG extends separation logic and adds the notion of action (state transitions). Both the programming and assertion languages have a formal semantics. It would be interesting to embed these semantics in Isabelle, prove some properties of the logical connectives and other lemmas presented in the LRG paper, and use the formalisation to prove individual programs.

Feng [3] has hinted that LRG contains concurrent separation logic as a special case. In our project we could prove mechanically those example proofs that O’Hearn presented in his paper on concurrent separation logic (CSL) [7]. Also we could express in LRG all those predicates that are defined in CSL and hard-wired in Smallfoot - i.e. predicates for singly- and doubly-linked lists and trees, and see if it is possible to provide a way to define arbitrary inductive definitions of data structures (a limitation of the Smallfoot tools). The Smallfoot authors have also said that their tool is unable to prove more complex graph algorithms that have been proven manually with separation logic on paper - the proofs of these algorithms would constitute a significant case study.

At the time of writing, there is no known Isabelle/HOL embedding of a program logic that merges rely-guarantee and separation logic; however there are Isabelle/HOL formalisations of these logics separately in the literature. Rely-guarantee has been formalised in Isabelle by Nieto [5]. In [12] Weber encodes separation logic in Isabelle. In [9] Tuerk wrote a general framework for separation logic inside the HOL theorem prover based on Abstract Separation Logic. It has been instantiated to build a tool that can parse Smallfoot specifications that can automatically reason about shape invariants and interactively reason about the contents of data structures. However at the time of writing his formalisation does not have support for reasoning about synchronisation mechanisms.

Methodology The research methodology can be classified broadly into three kinds of activities: detailed literature review, study of existing tools, study of theorem-proving techniques, embedding of the main programming logic in the theorem prover, and the use of the resulting embedding to verify a range of algorithms.
A. Detailed literature review  This activity involves reading all available publications about separation logic, rely-guarantee reasoning, and formalisms that try to merge the two. The aim of the activity is to understand their motivations, key insights, differences in approach, semantic foundations, and applicability to the deductive verification of concurrent algorithms. This will also aim to reinforce understanding by following any manual proofs that have been done with these formalisms and working out by hand some chosen examples.

B. Study of existing tools  Investigate existing tools that support reasoning with these formalisms, e.g. Smallfoot and SmallfootRG. Install these tools and experiment with them. Understand the input languages, the verification methods used, and the kinds of algorithms and properties that these can verify.

C. Study of theorem-proving techniques  Understand how to use the Isabelle/HOL proof assistant, and in particular, find out the basic techniques for formalising the semantics of simple programming languages and assertion languages. Study how to embed basic program logics in it and construct proofs demonstrating some properties of program logics (e.g. soundness and completeness) as well as proofs of correctness of particular programs within the logic. Learn about basic techniques such as inductively-defined data types, sets, and predicates, and proofs by structural induction. Learn how to to write proofs using the Isar proof language [14, 13], which allows for more readable proofs compared with the traditional tactic-oriented proof construction.

D. Embedding the main program logic in Isabelle  Construct an embedding of LRG in Isabelle/HOL: define the programming language and assertion language as inductive data types and model their operational semantics; construct models of the heap as partial functions or records and model heap operations such as allocation, lookup, update, and deallocation; mechanically prove some properties of the logical connectives and other lemmas presented in the paper [3]; encode the inference rules as inductive sets (or predicates); and construct a mechanical proof of soundness of the logic.

E. Verification of a range of concurrent algorithms  Construct proofs of full functional correctness of a range of algorithms to demonstrate the utility of the program logic and its formalisation in the proof assistant. Such algorithms would include semaphore-based algorithms such as those treated by O’Hearn in [7], and algorithms involving concurrent lists, stacks, and queues.

Expected contributions

- An evaluation of the effectiveness of LRG as a program logic, as measured by the range of algorithms that can be verified by it.
- A mechanisation of LRG in Isabelle/HOL and a mechanised proof of its soundness.
- A collection of non-blocking algorithms that are verified using our mechanised logic.

Preliminary results  Over the past 1.5 years studying part-time, I have done a review of the relevant literature and discussed some of the papers in detail with my supervisors. This review covered separation logic, rely-guarantee, RGSeq, and LRG. We have seen that separation logic’s strength is local reasoning, which allows for more modular proofs and eliminates the need for reasoning about global program states, as has been the case with traditional Hoare logic and its variants. On the other hand, rely-guarantee reasoning is very good for reasoning about process or thread interference and would be ideal for verifying concurrent algorithms which use non-blocking synchronisation instead of locks or other mutual-exclusion-based mechanisms, as the main feature of these algorithms is that they allow controlled interference rather than eliminating it altogether. I have also installed and experimented with Smallfoot and SmallfootRG and seen how these tools can verify data structure invariants. Also I have learned how to use Isabelle/HOL with Isar and have learned how to define inductive types and carry out proofs by structural induction and rule induction, using both the traditional tactic-style proof method and the high-level and structured Isar proof language. I have constructed some basic formalisations, including the following:
• operational semantics (big- and small-step semantics) of a simple procedural language extended with pointers
• a Hoare logic for partial correctness, together with a proof of soundness and completeness
• A shallow (semantic) and an incomplete deep (syntactic) embedding of the assertion language of separation logic

Key References


