Formalizing a System for Deadlock Checking by Data Race Detection in Coq

Peter Brottveit Bock

Department of Informatics, University of Oslo

Creating concurrent programs is considered more difficult than single threaded programs. The two most common classes of errors are *deadlocks* and *race conditions*, where deadlocks occur when several threads are waiting in a cyclic manner for resources that other holds, whereas race conditions happen when several threads access a shared variable without protection with where at least one thread modifying the variable.

It is therefore desirable to statically analyze programs to check if they contain deadlocks. The systems which do this are not trivial; they usually contain many definitions and tedious proofs. If such a system is presented, is it therefore of interest to formally check that the system has the properties it has been claimed to have.

This abstract describes the formalization of a set of type and effect system related to deadlock checking, using the tools Ott[1] and Coq.

1 Background

In the technical report *Deadlock Checking by Data Race Detection* [2], a novel idea is introduced: a reduction from deadlock checking to data race checking. The idea is that given a program with locks, it can be analyzed to approximate how the locks are used, and then fresh variables can be inserted into the code at strategic locations. If there is a deadlock in the original program, the inserted variables will cause a race condition. One can therefore leverage current and future tools and research about data race detection, into deadlock checking. For race condition detection, there exists powerful static analysis tools, such at Goblint for C and Chord for Java.

The language of study is a functional language, with dynamic thread creation, dynamic lock creation, and reentrant lock acquiring.

The semantics is specified by a small-step operational semantics, divided into two parts: *local* steps and *global* steps. The local steps are syntax-directed and concern how a single thread operates independent of other concurrent threads. The global steps describe spawning of threads, manipulation of locks, together with interleaving of the local steps for a finite set of threads.

The analysis centers around estimating an upper bound for how many times a lock is acquired by a thread, and how this is changed by executing an expression. The latter is called an *effect* of the expression. An estimation is denoted by Δ, which can be considered a mapping from lock set variables to $\mathbb{R} \cup \{\pm\infty\}$. Finally, $t :: \Delta \to \Delta'$ means that executing t from the initial, abstract state Δ, will lead to the state Δ' (as a partial correctness property).

Also function types are annotated with effectw, as in $\hat{T} \xrightarrow{\Delta \to \Delta'} \hat{T}'$, which means that the body of the function has the effect $\Delta \to \Delta'$. Additionally, lock types are annotated with lock sets, indicating where in the program a lock may have been created.

The first type system is the specification, and the purpose of it is to state the desired system in a clear, elegant and declarative manner. It has judgments on the form

$$C; \Gamma \vdash t : \hat{S} :: \Delta_1 \to \Delta_2,$$

where C is a set of constraints, Γ is the context, mapping variables to type schemes, t is a single thread, and $\hat{S}$ is an annotated type scheme.

As the purpose of this system is to get a clear understanding of what is wanted, the system is not syntax-directed, and non-deterministic. The downside is that it does not make it clear whether the analysis can be done efficiently. This leads us to the next type system.

The algorithm is a syntax-directed type system without non-determinism. The most important difference from the specification, is that the algorithm *generates* a set of constraints. The form of a judgment is therefore $\Gamma \vdash t : \hat{T} :: \Delta_1 \rightarrow \Delta_2; C$.

Another thing to note is that the judgment has a type in the conclusion, not a type scheme. The reason is that, while the specification freely allows instantiation and generalization of a type at any point, the algorithm does this only at the places where it is necessary (instantiating when referencing a variable, generalizing in let-expressions).

Several properties are proved informally: subject reduction between the syntax-directed system and the semantics; soundness of the algorithm with regards to the specification; and completeness of the algorithm with regards to the syntax-directed system.

2 The Formalization

My work consists of a formalization of the syntax, semantics, type systems, and the proposition and proof of soundness. The goal of the formalization is to make the definitions and properties machine checkable. The proofs have been done top-down, and are cut off when they reach "obvious" truths.

The tools used to achieve this are Ott, a tool specialized for programming language theory researchers, and Coq, a very powerful, general purpose theorem prover assistant.

The grammar and most judgments have been formalized in Ott. Ott naturally lends itself to a deep embedding, but it is possible to bypass Ott's generation of Coq-code for the abstract syntax tree, and define manually how a grammar should be translated to Coq. Bypassing Ott's manual generation of Coq-code make the embedding shallower, and one can leverage existing theorems.

Ott can generate substitution and free variable functions from the grammar. Unfortunately, using some of the more advanced features of Ott makes the generated functions useless and they have to be hand coded in Coq. Additionally, the substitution functions generated are not capture avoiding.

The judgments from the paper can fairly directly be translated to Ott. Properties which are not given as syntactical rules, are coded by hand in Coq.

One initial challenge was the syntax of a program, which casually states that the $||$-operator should be associative and commutative, with $\emptyset$ as the identity. This property should, presumably not be used in the algorithm, as this would cause non-determinism in how to apply the rules. The current proofs have not needed this property yet, but it could be needed in either a sub-proof or in a later proof.

One of the major issues was how to handle *freshness*-claims in derivations, which are usually given as a premise X *is fresh*. Such a claim is not local; to check if it holds, the whole derivation must be inspected. The workaround needed to solve this are not pretty. Two ways it can be solved are:

1. Let the claim trivially be solved by a constructor, and then define a proposition on derivation which checks that the freshness-claims actually hold.

2. Modify the rules, so that information about all variables used “flow” through the system, so that freshness can be checked locally.

The first solution has the nice property that the rules will be similar to those presented in the paper, but it also forces the derivation to live in Coq’s *Type*, and provability then becomes both inhabitance and the truth of a predicate on derivations.

To implement the second solution, the rules have to be littered with variables which makes it possible to generate fresh variables, and to enforce that these generated variables are not used. Since variables are indexed by numbers, it is natural to make each rule take a number as its “input”, which can be incremented to generate fresh variables, and “return” the new maximum number. This is the chosen solution.

Formalization of the soundness proof has been reduced to simpler lemmas. During the process, errors of varying degrees of importance was found. Some related to weaknesses in the formalization, others to properties not mentioned in the paper, and finally technical errors. No error found has been grave enough to dismiss the general idea, but especially the typing rules for recursive functions are problematic.

3 Experience

Using Ott alone is useful, as it can detect ambiguity, and since it generates both LaTeX and Coq code, one never ends up with inconsistencies between the definitions in a paper and in the theorem prover. Also, the possibility to print out the grammars and judgments in an accessible notation makes it possible to discuss the work with people not familiar with the syntax of Coq.

The downsides with Ott is that the code generation can feel unreliable, and that the generated function for free variables and substitutions often become useless.

As someone with some experience with Coq, Ott forces a perspective where syntax and judgments are first class citizens, while functions and general propositions are second class. When a Ott-project grows larger, the idiosyncrasies of Ott come forth, and problems for which one has a straightforward Coq-solution need to be worked into language of Ott.

References

[1] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. *Ott: Effective Tool Support for the Working Semanticist.* ICFP, 2007.

[2] Ka I Pun, Martin Steffen, Volker Stolz, *Deadlock Checking by Data Race Detection.* Preprint submitted to Elsevier, 2013.