Improving Permission-Based Verification of Concurrent Programs with Chalice

Problem Description

Stefan Heule
stheule@student.ethz.ch

September 29, 2010

1 Background

Chalice is an experimental programming language and a verifier for concurrent programs written in this language. Its methodology centres around permissions and permission transfers, where permissions currently are represented as a pair (p, n) with $p \in \mathbb{N}_0$ and $n \in \mathbb{Z}$. Such a permission can be interpreted as a percentage $p + n \cdot \varepsilon$, where ε is an infinitesimal small number that cannot be further split. The set of legal permissions is restricted by $0 \leq p + n \cdot \varepsilon \leq 100$.

2 Main task

Permissions in the current encoding cannot be split forever, as an ε-permission is indivisible. That is, the current encoding essentially implements counting permissions. However, there are cases where it is desirable to have fractional permissions, i.e. permissions that can be split infinitely often. A fractional permission corresponds to a rational number $r \in \mathbb{Q}$ with $0 \leq r \leq 1$. This implies that read permissions can be split as often as needed, and in fact, the permission `rd(x.f)` in this new model stands for any fraction greater than 0, as opposed to a fixed number ε.

Current theorem provers perform rather bad when rational numbers are used. For this reason, permissions are stored as an integer $n \in \mathbb{N}_0$, which stands for the permission $\frac{n}{d}$, where d is a suitable natural number. This means that the theorem prover only needs to deal with natural numbers as permissions, while this still allows the full implementation of fractional permissions. d is a constant, such that all permissions necessary for the verification of a particular program can be represented in this way. The value of d is not actually used in the verification, the mere fact of its existence is enough to proof the desired properties. A full permission can then be represented by the constant d, and any value below d stands for a read permission.

The core tasks of this Bachelor thesis are the following:

- **Understand**. First of all, the current permission model, its implementation in the Chalice verifier and the interaction with Boogie has to be understood.
- **Implement**. The current version of Chalice should be adopted to the new fractional permissions model with the integer encoding described above. Chalice as a programming language should in this main task not be modified, only the implementation of the verifier and the semantics of read permissions.
- **Evaluation**. The new permission model and its implementation should be critically tested and evaluated. Of special interest is the completeness aspect of the new versions of the verifier compared to the old one. Are there examples that currently can be verified, but cannot with the new model, and/or vice versa? The evaluation should also include considerations of performance and usability.

3 Further work

Depending on the time left, progress of the project and personal preferences, there are several extensions possible to the project, including the following:

- Formalization of this new encoding, including a soundness proof.
- The current implementation suffers from a soundness issue related to folded predicates. A sketch solution to this issue exists, which could be made fully precise and implemented to fix the problem.
- The new permission model does not yet provide true fractional permissions, as it is in general not possible to relate two read permissions `rd(o.f)`. For instance, it may be desirable to state that a method returns (through its postcondition) half the permission it received, and uses the other half in some other way. Further work could define a language extension for Chalice to be able to make such statements, and implement the necessary changes to allow their verification.
- The current permission model is far more close to counting permissions than the new model. The new fractional permissions are very useful, though there seem to exist examples for which counting permissions are more natural. A possible extension could investigate this further, and possibly extend the model to allow counting permissions as well.
- The new implementation will leak some fraction of read permissions stored in a monitor when acquiring said monitor and releasing it afterwards again. This means, that reconstructing a full write permission is not possible after the monitor has been acquired by a thread. At the moment this does not seem to be an issue, as write permissions are not affected. However, if there turn out to be non-artificial examples where correct statements cannot be proven, an extension could improve the handling of read permissions in monitors by trying to avoid this leaking.

It is not necessary to decide at this point which of these extensions should be considered, as this can be best determined after most of the main task has been completed.