Translating Chalice into SIL

Problem Description

Christian Klauser
klauserc@student.ethz.ch

November 16, 2011

1 Background

Chalice is an experimental programming language designed for specifying and verifying concurrent programs. It is centred on the idea of passing permissions back and forth between methods, threads, monitors and channels. Currently, the Chalice verifier works by encoding its input program and Chalice's semantics into a Boogie program and having that program verified by the Boogie verifier. SIL is the intermediate language used in Semper, a long term project aiming to create an extensible, symbolic-execution-based verifier for Scala.

2 Main task

The goal of this Bachelor's thesis is to design and implement a new back end for the Chalice compiler, targeting SIL instead of Boogie. In addition to interfacing Chalice with a new verifier backend, the project helps flesh out SIL and test the tool chain of the Semper project.

The resulting program should

- use the existing Chalice front end with as few changes as possible. This ensures that the language does not get fragmented.
- be able to translate basic language constructs like assignments, conditions and loops so that they can be verified.
- be able to translate method calls, thread forks and joins, making sure that permissions are passed around correctly.
- be able to translate monitor-based locks, including transfer of permissions into and out of the lock. Chalice's deadlock-avoidance mechanism can be considered as an extension.
- track locations in the original source code in order to generate meaningful error messages in case an input program fails verification. The use of an intermediate representation should be as transparent as possible.

3 Extensions

Depending on the progress of the main task, several extensions to the project are possible.

- Chalice comes with support for *predicates* and *functions*, features that allow for information hiding and code reuse in invariants, pre- and postconditions. *Functions* in particular offset Chalice's lack of *pure* methods. *Predicates* and *functions* need to be translated into the target language in order to verify Chalice programs using these features.
- Deadlock-avoidance is a very important feature for a language that is designed to enable the verification of concurrent programs. Chalice already provides a mechanism for preventing deadlocks on *monitors*, it would just have to be translated into the target language.
- Many of the more interesting Chalice programs make use of *channels* to simplify communication between threads (Actor model). Like with *monitors*, there is a mechanism for avoiding deadlocks on `receive` operations. Again, these would have to be translated into the target language.