Support for Closures in an automated modular verification setting

Abstract

Delegation is an important concept of executing an unknown task by calling a method which is chosen at runtime. It's used in several common design patterns like Visitor, Command and Chain of Responsibility. Unfortunately, this dynamic aspect makes delegates hard to verify, since the specification of a client may depend on the behavior of statically unknown delegates. This raises many well known concerns like the recursion through the store phenomenon, where a contract depends on itself. The Paper "Modular Specification and Verification of Delegation with SMT Solvers" (Kassios, Müller 2013) shows that such issues can be solved with first-order logic, which is believed to be applicable to fractional permission based languages like Chalice.

Chalice is a fractional permission-based experimental language used to verify concurrent programs. ETH researchers developed a verifier for Chalice called SILICON, which supports a permission-style verification with Symbolic Execution. SILICON operates on an intermediate language called SIL, into which Chalice can be translated by the "Chalice to SIL" translator. At present the verification of closures is not supported. The main goal of this project is to add a way to specify closures in Chalice and extend the already existing translator with support for these closures such that certain properties of these closures can be verified with SILICON. Many contracts used to verify code are trivial, thus the translation should be able to infer some contracts automatically, such that the programmer doesn't have to specify them. Some of these inferable contracts are treated as extensions for the project.

Core: Part 1

The first part is finding a modular methodology for verifying closures using a combination of first-order logic and linear permission based logic. In particular for the following issues a solution must be found:

- Procedure contracts may refer to the specification of other methods, for example a closure factory must describe the behavior of the delegate it creates. In particular statements like `pre(c, ·)` and `post(c, ·)` for a closure `c` and `old(·)` expressions must be supported.
- Closures for arguments or as return values must be possible. Background knowledge used to verify such closures could be inferred by the verifier, but as a first step it can be assumed that you don't need any additional knowledge except what is present in the contracts.
- A closure may refer to itself through statically unknown pointers to other delegates. There must be mechanisms to prevent such circular dependencies in the contracts of closures (recursion through the store) because recursive definitions of specifications may be inconsistent.
- Capturing of state is a property which is hard to specify in the current setup, thus it can be ignored in the core part of the project. This is no constraint since you can still allow C#-like delegates which refer to non-anonymous methods of specific objects for most purposes.

- Custom control flow creation like a `while(c, b)` (where `c` and `b` are closures) or a Chain of Responsibility must be verifiable, but ghost parameter closures needed for verifying don't have to cover multiple states (like `old(·)` expressions).

There are several examples prepared by the supervisor which cover these issues. The new methodology should be efficient and sound in regard to the translation from Chalice to SIL and contracts should be verifiable by a SILICON-like verifier.

Core: Part 2

The second part of the core project is an actual implementation of the newly found methodology. Main steps are adding an user friendly way to define Closures in Chalice and extending the Chalice to SIL translator with support for these closures such that certain properties of these closures can be verified with the SIL verifier. Another focus is to reduce the specification overhead of trivially inferable contracts for the user wherever easily possible.

The new system should not conflict with anything already existing and examples for all the above issues must be verifiable by the SIL verifier using the new translation.

Extensions

Possible extensions of the project are the following:

- Tackling reasonably large examples from the literature, and tweaking the prover accordingly to achieve scalability.
- Add support for nested closures (anonymous methods in C#) and therefore state-capturing (anonymous closures can capture variables which are out of scope when the delegate is executed). Magic Wands can probably be used for this purpose.
- Automatically include background knowledge (state that is not given as arguments) needed to verify nested closures into the contracts.
- Extend custom control flow creation with ghost parameter closures which cover multiple states.
- Extending and unifying the specification language, which for the time being is minimal.
- Using the approach to support behavioral subtyping with traits or mixins in Scala.