Master Thesis Proposal: Adding Closure Support to Chalice

Prateek Agarwal

Introduction

Motivation

In programming methodologies research, special languages are created with minimal features to focus on core concepts of the language in isolation. These languages contribute as proof of concept to demonstrate efficacy of certain programming methodology. However to realize their potential the languages have to be developed on par with languages used in industry such as C#, Java etc. One of the biggest hurdle in this path is for different language features to work together.

Introduction To Chalice

Chalice[3], an experimental object based programming language provides methodology to guarantee freedom from concurrency issues such as deadlocks and race conditions. Fractional Permissions [5] are used to express access permissions on heap locations shared between concurrent threads. It uses Implicit Dynamic Frames [4], a methodology used for verification of non concurrent programs. Chalice programs are translated into Boogie programs which can be verified automatically using the Boogie program verifier [6] and an SMT solver like Z3 [7] without any human assistance.

Adding Closure support to Chalice

Chalice has minimal support for object oriented features. One of such features heavily used in OO languages is closure. Closures are functions which can capture their lexical environment. Using closures one can write concise programs and utilize newer programming patterns. With their introduction in C# [8] and Scala [9] and dynamic languages such as Python [10] and Ruby [11] closures have gained popularity in main stream industrial usage. Recently a first order formalism has been developed for verification of sequential programs with closures [1].

The goal of current project is to use ideas from [1] to add closure support in Chalice. The methodology from [1] uses Dynamic Frames[2] whereas Chalice uses Implicit Dynamic Frames[3] such a work presents challenges

Challenges in adding closure support to Chalice

a) Implementation of closure verification using abstraction mechanism of Chalice poses significant challenges

b) Two closures can share state by capturing their lexical environments. Since there is no known way to refer to and lock the captured state, this easily leads to race conditions. A formalism is needed to address this problem.

c) Chalice uses assertions like **acc(n)** which when used in a method's precondition indicates the method requires permission to modify the field **n** of **this** object. Such assertions can be combined with boolean operators e.g.

$$acc(n) \wedge (n * n \geq 0 \Rightarrow n \geq 0)$$

but such expressions are not allowed to be present in antecedent of implications which seems to be an important part of the methdology of [1]. A theoretical framework to define semantics of such expressions is still under development.

Scope

The scope of the project is limited to language extension. Closure is the programming language feature chosen to be added to Chalice. The project work includes the design and implementation of the extended language. Primary focus of the work is to combine the two formalisms of verifying closures and verifying concurrent programs.

Goals

Goal	**Requirements**
Addition of closures in Chalice	Document Lexical & Parsing (CFG) changes [1]
Development of test suite	Comprehensive set of programs
Modification of Chalice compiler	No issues with previous test suite and works well for new tests
Writing Report	Report highlighting key issues and findings of project

Development is to be done with aim at contribution to Chalice open source code repository. An exceptionally successful project is one that makes it to the Chalice open source repository as an official extension of the language.

Plan

Overall Plan

- Understanding existing work

  Getting familiar with Chalice language features

  Understanding dynamic and implicit dynamic frames

  Understanding formalism for closure verification

- Language Design

  Syntax of new language extension

  Informal document giving generated Boogie code for Chalice program elements

- Development of test cases

- Framework Design Changes

  Identifying additional issues & challenges with verification of closures within Chalice framework

  Identifying Chalice framework elements which conflict with the verification of closures

[1] An informal document is required which lists generated Boogie code for relevant Chalice programming constructs

- Extension of Chalice compiler to include the closure feature

  Extension of language syntax and semantics in Chalice compiler

  Addition of closure verification in Chalice verification framework

  Integration of closure verification in Chalice verification framework

- Writing Report

References

[1] Ioannis T. Kassios & Peter Mueller, Specification and Verification of Delegates in First Order Logic.

[2] Ioannis T. Kassios, Dynamic frames: Support for framing, dependencies and sharing without restrictions, FM06, volume 4085 of Lecture Notes In Computer Science, pages 378-393. Springer-Verlag, 2009

[3] Leino and Muller, A basis for verifying multi-threaded programs. Programming Languages and Systems, volume 5502 of Lecture Notes In Computer Science, pages 268-283. Springer-Verlag, 2006

[4] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames. In FTfJP 2008, pages 1-12,2008. Technical Report ICIS-R08013, Radboud University

[5] J. Boyland. Checking interference with fractional permissions. In SAS 2003, volume 2694 of Lecture Notes in Computer Science, pages 55-72. Springer, 2003

[6] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In FMCO 2005, volume 4111 of Lecture Notes in Computer Science, pages 364-387. Springer, 2006

[7] Leonardo de Moura, Nikolaj Bjørner Z3: An efficient SMT solver. In TACAS 2008, volume 4963 of Lecture Notes in Computer Science, pages 337-340. Springer, 2008.

[8] ECMA. C# language specification. Technical Report Standard 334, ECMA, 2006.

[9] M. Odersky, L. Spoon, and B. Veners. Programming in Scala. Artima, 2007

[10] D. M. Beazly. Python Essential Reference. SAMS, 3rd edition, 2006.

[11] D. Flanagan and Y. Matsumoto. The Ruby Programming Language. OReilly, 2008