Modular Reasoning about Concurrent Higher-Order Imperative Programs

Lars Birkedal
Department of Computer Science, Aarhus University
birkedal@cs.au.dk

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory – semantics; D.2.4 [*Software Engineering*]: Software/Program Verification

Keywords contextual equivalence; data abstraction; program logics

1. Introduction

Modern mainstream programming languages permit a powerful combination of language features: concurrency, higher-order functions, and mutable shared data structures. These features are all very important for programming in practice. However, it is well-known that the combination of them makes it difficult to write correct and secure programs, and it is therefore important to develop mathematically-based techniques for formal reasoning about correctness and security of programs with these features.

To get scalable methods that apply to realistic programs, it is crucial that the mathematical models and logics support *modular* reasoning, meaning that (1) specifications and proofs are compositional wrt. the program structure; and (2) specifications and proofs can concentrate on the resources that a program actually acts upon, instead of its entire state.

In this talk I will give an overview of some of the central developments that my group has worked on when generalizing reasoning techniques for simpler programming languages to concurrent, higher-order, imperative languages. I will consider both relational models and program logics based on higher-order separation logic.

2. Kripke Logical Relations

One of the most fundamental notions in formal reasoning about programs is contextual refinement, which expresses when one program fragment may be replaced by another one, without changing the overall behaviour of a complete program. This is the notion of correctness that one is interested in when, e.g., reasoning about data abstraction or proving compiler optimizations correct. It is difficult to reason directly about contextual refinement of programs and thus there has been a lot of work on extending the techniques of (bi)simulations and logical relations from simply-typed purely functional programming languages to programming languages with recursive types, polymorphism, dynamically allocated mutable references, and, lately, also concurrency. For the work on logical relations, the main challenges have been: (1) to develop mathematical techniques that are expressive enough to construct the interpretation of all the types – this is difficult when the programming language includes recursive types and general references, which preclude a definition of the relation by direct induction over the types; (2) to develop models that are sufficiently expressive for reasoning about uses of local state, i.e., to define the logical relations in such a way that one can show that two programs that use local state in very different ways to implement the same functionality are indeed related; and (3) to develop logical relations for concurrent languages which allow one to reason about one thread at a time, while still accounting for the possible interference of other running threads. To address these challenges we have developed new kinds of Kripke logical relations using guarded recursion. The resulting models have been used, e.g., (1) to prove correctness of a parallelization theorem, which expresses when it is sound to evaluate two expressions in parallel instead of sequentially, by giving a relational model for a type-and-effect system; and (2) to show correctness of fine-grained concurrent data structures by showing that they are contextual refinements of their coarse-grained counterparts, so that clients can reason about fine-grained concurrent data structures as if all accessses to them were sequentialized.

3. Higher-Order Separation Logics

Instead of specifying a program by relating it to another program, we can specify programs using program logics. Separation logic allows specifications and proofs to concentrate on the resources that a program actually acts upon, instead of its entire state. To allow specifications and proofs to be compositional wrt. the program structure we developed higher-order separation logic. To reason about higher-order imperative programs, we introduced nested triples. These features facilitate modular reasoning about sequential higher-order programs with limited forms of sharing. To also cater for concurrency and sharing, we make use of earlier work that extend rely-guarantee versions of separation logic with protocols governing access to shared mutable state. For modularity, we use higher-order quantification over so-called impredicative protocols. To show soundness of the logic we build a non-trivial model using guarded recursion. The resulting logic can be used, e.g., to give modular specifications of layered and recursive higher-order abstractions, as used ubiquitously in GUI and network applications for implementing asynchronous I/O.

POPL '14, January 22–24, 2014, San Diego, CA, USA.
Copyright is held by the owner/author(s).
ACM 978-1-4503-2544-8/14/01.
http://dx.doi.org/10.1145/2535838.2537849