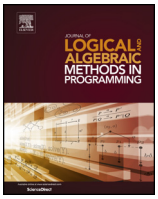
Journal of Logical and Algebraic Methods in Programming 100 (2018) 130–151

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp



Bestow and atomic: Concurrent programming using isolation, delegation and grouping ☆

Elias Castegren *, Joel Wallin, Tobias Wrigstad

Uppsala University, Sweden

ARTICLE INFO

Article history:
Received 22 August 2017
Received in revised form 6 April 2018
Accepted 27 June 2018
Available online 30 June 2018

ABSTRACT

Any non-trivial concurrent system warrants synchronisation, regardless of the concurrency model. Actor-based concurrency serialises *all* computations in an actor through asynchronous message passing. In contrast, lock-based concurrency serialises *some* computations by following a lock–unlock protocol for accessing certain data.

Both systems require sound reasoning about pointers and aliasing to exclude data-races. If actor isolation is broken, so is the single-thread-of-control abstraction. Similarly for locks, if a datum is accessible outside of the scope of the lock, the datum is not governed by the lock.

In this paper we discuss how to balance aliasing and synchronisation. In previous work, we defined a type system that guarantees data-race freedom of actor-based concurrency and lock-based concurrency. This paper extends this work by the introduction of two programming constructs; one for decoupling isolation and synchronisation and one for constructing higher-level atomicity guarantees from lower-level synchronisation. We focus predominantly on actors, and in particular the Encore programming language, but our ultimate goal is to define our constructs in such a way that they can be used both with locks and actors, given that combinations of both models occur frequently in actual systems.

We discuss the design space, provide several formalisations of different semantics and discuss their properties, and connect them to case studies showing how our proposed constructs can be useful. We also report on an on-going implementation of our proposed constructs in Encore.

1. Introduction

Concurrency can be defined as coordinating access to shared resources. Synchronisation is naturally a key aspect of concurrent programs and different concurrency models handle synchronisation differently. Pessimistic models, like locks or the actor model [1,2] serialise computation *within certain encapsulated units*, allowing sequential reasoning about internal behaviour at the cost of sometimes pruning possible parallel performance gains. In contrast, optimistic models, like lock-free programming [3] or software transactional memory [4] allow concurrent operations on the same data, but require that all

☆ This work is sponsored by the FP7 project "UPSCALE" (Project number 612985), and the Swedish Research Council through the UPMARC center of excellence and the project "Structured Aliasing" (Project number 2012-4967).

* Corresponding author.
E-mail addresses: elias.castegren@it.uu.se (E. Castegren), joel.wallin.3149@student.uu.se (J. Wallin), tobias.wrigstad@it.uu.se (T. Wrigstad).

https://doi.org/10.1016/j.jlamp.2018.06.007
2352-2208/

operations follow some protocol to exclude unwanted behaviour, or avoid side-effects that cannot be rolled-back in the event of conflicts between threads. In this paper, we focus on pessimistic models.

In the case of the actor model, if a reference to an actor *A*'s internal state is accessible outside of *A*, operations inside of *A* are subject to data-races and sequential reasoning is lost. The same holds true for operations on an aggregate object behind a lock, if a sub-object is leaked and becomes accessible where the appropriate lock is not held.

In previous work, we designed Kappa [5], a type system in which the boundary of a unit of encapsulation can be statically identified. An entire encapsulated unit can be wrapped inside some synchronisation mechanism, *e.g.,* a lock or an asynchronous actor interface, and consequently all operations inside the boundary are guaranteed to be data-race free. An important goal of this work is facilitating object-oriented reuse in concurrent programming: internal objects are oblivious to how their data-race freedom is guaranteed, and the building blocks can be reused without change regardless of their external synchronisation. Further, making synchronisation tractable simplifies concurrent programming as the portions of a system that are accessed concurrently will be identified, and compilers can verify that the program behaves in accordance with the programmer's intention, with respect to concurrent accesses.

This paper explores two extensions to the Kappa system, which we explain in the context of the actor model (although they are equally applicable to a system using locks). The first extension, *bestow*, allows references to an object to escape its *unit of encapsulation* without escaping its *unit of synchronisation*; all external operations on private state will be implicitly delegated to the owner of that state, either via message passing, or by acquiring a lock specific to the owner of the private state. This enables several useful programming patterns without relaxing Kappa's static data-race freedom guarantee. For example, in the context of an actor system, actors may safely leak references to state, effectively allowing many objects to cooperate in constructing the actor's interface. Similarly, in the context of a lock, it will be possible to hold on to references deep inside a structure, even when the lock is not held, with a guarantee from the type system that these will not be used until the lock is re-acquired.

When encapsulation and synchronisation are decoupled, another extension becomes necessary to group operations together, enabling *atomicity* of multiple operations on a data structure that potentially has several different entry-points. To this end, we introduce an *atomic* block scoped construct. In the context of an actor system, this allows *e.g.,* grouping several messages to prune unwanted interleavings. In the context of a lock-based system, it allows performing several distinct operations without releasing the lock in-between.

This paper makes the following contributions:

1. We discuss (Section 3) the difference between actor systems where data-races are avoided through *isolation* of an actor's passive objects and actor systems where data-races are avoided through *delegation* of operations on passive objects to the actors that own them, and show how both are supported in the Encore programming language [6]. Encore supports delegation through *bestowed references*, which were introduced in Encore as part of this work. In contrast to systems like E [7] and AmbientTalk [8], Encore's delegation is purposely *not* transparent, allowing programmers to reason about the performance and latency of operations and distinguish between operations on local and remote objects (Section 4).
2. We extend the delegation concept with a notion of movement of objects between actors, *i.e.,* implicit and/or automated transfer of ownership. This enables a form of load-balancing of passive objects between actors (Section 4.3, formalised in Section 7).
3. We introduce a block-scoped construct for grouping operations to be performed as an atomic unit that can be applied to both actor-based systems and lock-based systems, as well as systems that combine the two models (Section 5).
4. We explore and formalise three variations of the semantics of bestowed references and atomic blocks, and show that the resulting systems are free from data-races. For simplicity, our formalisations are based on simple λ-calculus models with actors. We provide mechanised versions in Coq for others to build on (Sections 6–7).
5. We report on a number of small case studies using our proposed constructs (Section 8) and on the on-going implementation of bestowed references and atomic blocks in Encore (Section 9).

This paper extends earlier work [9] by adding the two variations of the semantics (Section 7), mechanising the semantics and their proofs [10], providing case studies (Section 8), reporting on the current state of implementation (Section 9), and adding more in-depth discussions about the work throughout all sections of the paper.

2. Background: Kappa and Encore

This section covers the background needed to understand the context of this work. It explains the basics of Kappa, a type system which guarantees data-race freedom in concurrent programs, and Encore, an actor-based language which uses Kappa to facilitate safe sharing between actors.

Kappa is a type system for concurrent object-oriented programming [5]. Its most important guarantee is that no two concurrent operations will access the same memory address, unless both operations only use this address for reading. Kappa achieves this by preventing the creation of aliases which could be used to cause data-races. A reference can only be shared between threads if all accesses are synchronised (*e.g.,* by using locks) or if all operations available through the reference are non-mutating.