Reference Capabilities for Concurrency & Scalability: an Experience Report

ELIAS CASTEGREN, Uppsala University
TOBIAS WRIGSTAD, Uppsala University

In this presentation, we report on our work on Kappa, a capability-based type system for parallel and concurrent programming. We have used variations of Kappa to achieve data-race freedom for object-oriented programs and safety for lock-free algorithms, and are currently using it to allow safe sharing of data in the actor-based language Encore.

1 KAPPA: TYPES FOR CONCURRENCY AND SCALABILITY

In previous work we introduced Kappa, a capability-based type system for parallel, object-oriented programming [2]. Our design goals was to give *static guarantees of data-race freedom* for highly concurrent programs without sacrificing *scalability*. Furthermore, we want to encourage programming in an *object-oriented* style by supporting features like encapsulation, code-reuse and subtype polymorphism.

In Kappa, data-race freedom is achieved through reference capabilities, which ensure that all interactions with an object are safe (*i.e.,* free from data-races): a newly created object is safe by construction, and the type system makes sure that all aliases may also be safely accessed. A (reference) capability specifies a set of allowed operations through its type. Additionally, the means of concurrency control of a capability is specified through a **mode** annotation. Kappa defines a taxonomy of modes which all specify *how* data-race freedom is achieved. For example, a **linear** capability is the only reference to an object, a **read** capability will never be used to mutate or observe mutation of the underlying object, and a **local** capability will never leave the creating thread.

An important concept in object-oriented programming is the ability to separate the different concerns of an object into different reusable modules, *e.g.,* through inheritance or by using traits. By defining the type and mode of a capability separately, Kappa allows first defining the business logic of an object together with its type, and then specifying the means of concurrency control (the mode) when the type is used. Notably, the implementation of an object can be agnostic to the concurrency control where the object is used, meaning that the same type can be reused with minimal code duplication across different concurrency scenarios: a type `List` can be used in a thread-local capability **local** `List` in one place, and in a unique, transferable capability **linear** `List` in another.

Additionally, capabilities can be composed to form new capabilities, analogous to composition of traits or mixins in object-oriented languages. Kappa defines a small calculus for composing capabilities which also describes how a composite capability may be decomposed into its constituents. When a capability is composed of two or more sub-capabilities which do not provide operations which could cause races, the type system allows splitting the capability up into its constituent parts, which are safe to use in parallel. If desired, the original capability can be restored through a merging operation. This enables parallelism patterns that go beyond what can be achieved using regions and effects (*e.g.,* by allowing unstructured parallelism), and without any additional annotations on methods [1].

In our first instantiation of Kappa, we achieved data-race freedom by banning shared mutable state all together; any object with mutable state that is reachable by more than one thread must be protected by a lock. Relying on locks makes reasoning simple, but hurts scalability as contention on shared resources increases. It is also too

This work is sponsored by the UPMARC center of excellence, the FP7 project “UPSCALE” and the project “Structured Aliasing” financed by the Swedish Research Council.
.

restrictive for many fine-grained concurrency patterns, such as lock-free algorithms, which require concurrent access to mutable state. To allow these patterns, we extended Kappa with a more fundamental notion of capabilities which are flexible enough to express several lock-free algorithms (and even implement locks natively), but still powerful enough to prevent (uncontrolled) races on mutable state [3].

2 KAPPA AND ENCORE: APPLYING THE THEORY TO A REAL LANGUAGE

We have implemented Kappa as the type system for Encore, a programming language which uses active objects[1] as its main mechanism for achieving concurrency. Letting the development of Encore drive the development of Kappa has been a fruitful way for us to discover new concepts and theory to explore. To adapt Kappa to a concurrency model based on active objects and asynchronous message passing, we extend the system with an **active** mode which describes a capability governing access to an active object, and which may only be interacted with asynchronously. Kappa ensures that no data shared between active objects is subject to data-races. Data-race freedom is not only a useful property for programmers, but is also a requirement for the Encore runtime, specifically the Orca garbage collection protocol [5]. Encore shares its runtime with Pony, another capability-based actor language [4]. Pony also uses reference capabilities to achieve data-race freedom, but Kappa is a more expressive system in terms of what kind of aliasing patterns it can capture.

As most actor languages, Encore relies on encapsulation to be able to reason about the behaviour of an active object as if there was a single thread of control. Encapsulation simplifies programming, but also requires an active object to extend its interface with all public operations available on its internal objects (analogous to a list having to provide the entire interface of its iterator). To alleviate this, we have extended Encore (and Kappa) with a mechanism for sharing private data between active objects and implicitly delegating operations to the active object which "owns" the data being operated on [6]. This allows programmers to switch from relying on isolation (as in *e.g.,* Akka) to relying on delegation (as in *e.g.,* E) to avoid data-races.

With a language implementation comes actual programming, and with actual programming comes programming conveniences that we did not yet address in our theoretical treatise of Kappa. For example, when implementing method overriding, care must be taken so that the new method does not break the assumptions of the original method so that we *e.g.,* accidentally provide a **read** capability with a method that performs mutation. A situation that comes up frequently in libraries is the reuse of functions and methods across different concurrency scenarios. Mode polymorphism must be carefully designed both from the inside and the outside—*e.g.,* to avoid the duplication of linear capabilities in a polymorphic context, and to avoid code that accidentally consumes a linear capability, losing the value. This gives rise to a notion of *polymorphic concurrency control*; a function may use a value knowing that it will not cause data-races, but may be agnostic to how this safety is achieved [1].

3 CONCLUDING REMARKS

In our presentation, we will explain the design of Kappa, the calculus of capabilities, the taxonomy of modes, and report on our experience with an actual implementation in terms of new challenges and positive feedback to the theory. Additionally, we will actively solicit input for upcoming design decisions, such as the semantics of certain combinations of modes, and the relation to more dynamic capability systems that allow *e.g.,* dynamic, unstructured revocation.

REFERENCES

[1] E. Castegren and T. Wrigstad. Kappa: Insights, Current Status and Future Work. In *IWACO*, 2016.
[2] E. Castegren and T. Wrigstad. Reference Capabilities for Concurrency Control. In *ECOOP*, 2016.
[3] E. Castegren and T. Wrigstad. Relaxed Linear References for Lock-free Data Structures. In *ECOOP'17*, 2017.
[4] S. Clebsch, S. Drossopoulou, S. Blessing, and A. McNeil. Deny Capabilities for Safe, Fast Actors. In *AGERE*, 2015.

[1] Active objects can be thought of as state carrying actors

[5] S. Clebsch, J. Franco, A. M. Yang, S. Drossopoulou, T. Wrigstad, and J. Vitekn. Orca: Leveraging Types and Messaging for Fully Concurrent GC. In *OOPSLA*, 2017. To appear.
[6] E. Castegren and T. Wrigstad. Actors without borders: Amnesty for imprisoned state. *PLACES*, 2017.