Abstract. An important application of unique object references is safe and ef- cient message passing in concurrent object-oriented programming. However, to prevent the ill effects of aliasing, practical systems often severely restrict the shape of messages passed by reference. Moreover, the problematic interplay between destructive reads—often used to implement unique references—and tem- porary aliasing through “borrowed” references is exacerbated in a concurrent setting, increasing the potential for unpredictable run-time errors.

This paper introduces a new approach to uniqueness. The idea is to use capa- bilities for enforcing both at-most-once consumption of unique references, and a flexible notion of uniqueness. The main novelty of our approach is a model of uniqueness and borrowing based on simple, unstructured capabilities. The ad- vantages are: first, it provides simple foundations for uniqueness and borrowing. Second, it can be formalized using a relatively simple type system, for which we provide a complete soundness proof. Third, it avoids common problems involving borrowing and destructive reads, since unique references subsume borrowed references.

We have implemented our type system as an extension to Scala. Practical ex- perience suggests that our system allows type checking real-world actor-based concurrent programs with only a small number of additional type annotations.

1 Introduction

Message-based concurrency provides robust programming models that scale from multi- core processors to distributed systems, web applications, and cloud computing. Seam- less scalability requires that local and remote message send operations should behave the same. A good candidate for such a uniform semantics is that a sent message gets moved from the memory region of the sender to the (possibly disjoint) memory region of the receiver. Thus, a message is no longer accessible to its sender after it has been sent. This semantics also avoids data races if concurrent processes running on the same computer communicate only by passing messages.

However, moving messages physically requires expensive marshaling (i.e., copy- ing). This would prohibit the use of message-passing altogether in performance-critical code that deals with large messages, such as image processing pipelines or network protocol stacks [19] [20]. To achieve the necessary performance in these applications, the underlying implementation must pass messages between processes running on the same shared-memory computer by reference. But reference passing makes it challenging to enforce race freedom, especially in the context of imperative, object-oriented languages, where aliasing is common. The two main approaches to address this problem are:
Immutable messages. Only allow passing objects of immutable type. Examples are Java-style primitive types (e.g., int, boolean), immutable strings, and tree-shaped data, such as XML.

Alias-free messages. Only a single, unique reference may point to each message; upon transfer, the unique reference becomes unusable [20,39,40].

Immutable messages are used, for instance, in Erlang [3], a programming language created by Ericsson that was used at first in telecommunication systems, but is now also finding applications in Internet commerce (e.g., Amazon’s SimpleDB1).

The second approach usually imposes constraints on the shape of messages (e.g., trees [40]). Even though messages are passed by reference, message shape constraints may lead indirectly to copying overheads; data stored in an object graph that does not satisfy the shape constraints must first be serialized into a permitted form before it can be sent within a message.

Scala [35] provides Erlang-style concurrent processes as part of its standard library in the actors package [25]. Scala’s actors run on the standard Java platform [31]; they are gaining rapidly support in industry, with applications in the Kestrel message queue system2 powering the popular Twitter micro-blogging service, and others.

In Scala actors, messages can be any kind of data, mutable as well as immutable. When sending messages between actors operating on the same computer, the message state is not copied; instead, messages are transferred by reference only. This makes the system flexible and guarantees high performance. However, race safety has previously neither been enforced by the language, nor by the run-time library.

This paper proposes a new type-based approach to statically enforce race safety in Scala’s actors. Our main goal is to ensure race safety with a type system that’s simple and expressive enough to be deployed in production systems by normal users. Our system removes important limitations of existing approaches concerning permitted message shapes. At the same time it allows interesting programming idioms to be expressed with fewer annotations than previous work, while providing equally strong safety guarantees.

1.1 Background

There exists a large number of proposals for unique object references. A comprehensive survey is beyond the scope of this paper; Clarke and Wrigstad [12] provide a good overview of earlier work, where unique references are not allowed to point to internally-aliased objects, such as doubly-linked lists. Aliases that are strictly internal to a unique object are not observable by external clients and are therefore harmless [48]. Importantly, “external” uniqueness enables many interesting programming patterns, such as merging of data structures and abstraction of object creation (through factory methods [23]). In the following we consider two kinds of alias encapsulation policies:

- **Deep encapsulation**: [33] the only access (transitively) to the internal state of an object is through a single entry point. References to external state are allowed.

---

1 See http://aws.amazon.com/simpledb/
2 See http://github.com/robey/kestrel/