Software ENgineering

Minimal ownership for active objects

D.G. Clarke, T. Wrigstad, J. Östlund, E.B. Johnsen

REPORT SEN-R0803 JUNE 2008

ISSN 1386-369X

Minimal ownership for active objects

ABSTRACT
Active objects offer a structured approach to concurrency, encapsulating both unshared state and a thread of control. For efficient data transfer, data should be passed by reference whenever possible, but this introduces aliasing and undermines the validity of the active objects. This paper proposes a minimal variant of ownership types that preserves the required race freedom invariant yet enables data transfer by reference between active objects (that is, without copying) in many cases, and a cheap clone operation where copying is necessary. Our approach is general and should be adaptable to several existing active object systems.

2000 Mathematics Subject Classification: none
1998 ACM Computing Classification System: D1.3, D3.3
Keywords and Phrases: ownership types; active objects; data races
Note: This work was carried out under project SEN3 EU-FP6 CREDO.

Minimal Ownership for Active Objects[⋆]

Dave Clarke[1], Tobias Wrigstad[2], Johan Östlund[2], and Einar Broch Johnsen[3]

[1]CWI, Amsterdam, The Netherlands
[2]Purdue University, USA
[3]University of Oslo, Norway

Abstract Active objects offer a structured approach to concurrency, encapsulating both *unshared* state and a thread of control. For efficient data transfer, data should be passed by reference whenever possible, but this introduces aliasing and undermines the validity of the active objects. This paper proposes a minimal variant of ownership types that preserves the required *race freedom* invariant yet enables data transfer by reference between active objects (that is, without copying) in many cases, and a cheap clone operation where copying is necessary. Our approach is general and should be adaptable to several existing active object systems.

1 Introduction

Active objects have been proposed as an approach to concurrency that gels naturally with object-oriented programming [1,43,70]. They are used for concurrency in, for example, Symbian OS [50], though threads are also allowed. Active objects encapsulate not only their state and methods, but also a single (active) thread of control. Additional mechanisms, such as *asynchronous method calls* and *futures*, reduce the temporal coupling between the caller and callee of a method. Together, these mechanisms make a large degree of potential concurrency exploitable for deployment on multi-core or distributed architectures.

Internal data structures of active objects, used to store or transfer local data, do not need independent threads of control. In contrast to the active objects, these *passive objects* resemble ordinary (Java) objects. An immediate benefit of distinguishing active and passive objects is that all the concurrency control is handled by the active objects, and locking (via *synchronised methods*) becomes redundant in the passive objects. This significantly simplifies programming and enables the (re-)use of standard collection APIs without additional concurrency considerations.

Unfortunately, introducing passive objects into the model gives rise to aliasing problems. Objects belonging to the state of one active object could be passed to another active object, enabling concurrent modification and/or observation of changes to the passive data objects via aliases. Specifically, two ‘threads’ can access the same passive data; if at least one thread modifies the data, then different access orders may produce different results. Unless passive objects have

[⋆] This work is in the context of the EU project IST-33826 CREDO: Modeling and analysis of evolutionary structures for distributed services (`http://credo.cwi.nl`).

locks (which is what the active objects model tries to avoid), concurrent access could occur at arbitrary points in the code. Thus, we are immediately recast into the setting of unconstrained shared variable concurrency. This problem can be addressed several ways:

immutable data only Active objects are mutable, but field values belong to immutable data types; e.g., integers or booleans, immutable objects such as Java-style strings or XML, or Erlang- and Haskell-style datatypes [5, 41].

cloning Locally, active objects can arbitrarily access passive objects, but when data is passed between active objects, the data must be deeply cloned. This approach is taken for distributed active objects (e.g., [7, 17]).

unique references Only one reference to any passive object allowed at any time. Passive objects can be safely transferred between active objects.

However, none of these approaches is entirely satisfactory.

Emerald [38, 60] partly addresses this problem using the first approach. Objects can be declared immutable to simplify sharing and for compiler optimisation, but immutability is an unchecked annotation which may be violated. Immutability is optional, as adopting pure immutability means that programs cannot be implemented in an imperative object-oriented style.

ProActive [7] uses the second approach and copies all message parameters. The programmer gets a simple and straightforward programming model, but the overhead may be massive in message-intensive applications.

Last, using uniqueness requires a radical change in programming style and may result in fragile code in situations not easily modelled without aliasing.

Contributions This paper investigates the application of ownership types in the context of active object-based concurrency. We develop a type system that statically identifies the boundaries of active objects and hence the opportunities for reference semantics instead of copying semantics in method calls. Our ownership types system requires very few annotations and seems practical to integrate with systems like SCOOP [48] and ProActive [7].

Concurrency is often mentioned as a natural albeit under-investigated application area for ownership types. In previous work, we combined ownership types with effects to facilitate reasoning about disjointness [19] and with uniqueness for ownership transfer [22]. Recently, we coalesced these to realise flexible forms of immutability and read-only references [58]. In this paper we tune these systems to the active objects concurrent setting, and extend the resulting system with the *arg* reference mode from Flexible Alias Protection [56]. Furthermore, our specific choices for ownership defaulting make the proposed language design very concise in terms of additional type annotations. This paper's main contributions are:

Type System A synthesised minimal type system with little syntactic overhead that identifies active object boundaries. This type system enables *static checking* and automatic inference of safe practices that programmers today must do manually (framework permitting), such as:

  – Replace deep copying with reference passing for immutable (parts of) objects, or objects which are invalidated at the source (unique references).
  – Replace remote references by copying for immutable (parts of) objects for more efficient local access in a distributed setting.

Generality Our results apply to any active object or actor based concurrency model. Active object systems such as ProActive [7], Emerald [38,60], and Scoop [48] use unchecked immutability or active annotations. Integrating our type system with these approaches for static checking seems straightforward.

Organisation Section 2 surveys the alias control mechanisms on which we build our proposal. Section 3 further details the problem we address and Section 4 presents our solution to it. Section 5 formalises our proposal in the language Joëlle, and shows its type system. Section 6 presents Joëlle's dynamic semantics and meta-theory. Section 7 compares our work with related work, and Section 8 concludes.

2 Ownership and Other Alias Control Mechanisms

This section surveys the alias control mechanisms used in this paper. These mechanisms address the problem of reasoning about *shared mutable state* [36, 56], which is problematic as a shared object's state can change *unexpectedly*, potentially violating a sharer's invariants or a client's expectations. There are three main approaches to this problem:

ownership: encapsulate all references to an object within some *box*; such as another object, a stack frame, a thread, a package, a class, or an active object [3,4,10,12,18,23,35,52,56].

uniqueness: eliminate sharing so that there is only one active reference to an object [3,10,13,22,35,49].

immutability: eliminate or restrict mutability so an object cannot change, or so that changes to it cannot be observed [9,14,56,63,66,71].

2.1 Ownership

Ownership types [23] initially formalised the core of Flexible Alias Protection [56]; variants have later been devised for a range of applications [3,10,12,18,23,52,56]. In general, object graphs form an unstructured "soup" of objects. Ownership types impose structure on these graphs by first *putting objects into boxes* [27], then imposing a topology [2,18] on the boxes, and finally restricting the way objects in different boxes can access each other, either prohibiting certain references or limiting how the references can be used [51,52].

Ownership types record the box in which an object resides, called the *owner*, in the object's type. The type system syntactically ensures that fields and methods with types containing the name of a private box are encapsulated (thus only accessible by `this`). This encapsulation ensures that the contents of private boxes cannot be exported outside their owner. For this to work, the owner information must be retained in the type. Consider the following code fragment:[1]

```
class Engine {}
class Car { this::Engine e; }
```

In class `Car`, the owner of the `Engine` object is `this`, which indicates that the object in the field `e` is owned by the current instance of `Car` (or, in other words, that every car has its own engine). The type system ensures that the field `e` is accessible only by `this`, the owning object.

Ownership types enforce a constraint on the structure of object graphs called *owners-as-dominators*. This property ensures that access to an object's internal state goes through the object's interface: the only way for a client of a `Car` object to manipulate the `Car`'s `Engine` is via some method exposed in the `Car`'s public interface. Some ownership types proposals [2, 3, 11, 52] weaken this property.

All classes, such as `Engine` above, have an implicit parameter `owner` which refers to the owner of each instance of the class. Thus, arbitrary and extensible linked data structures may be encapsulated in an object. Contrast this with Eiffel's expanded types [47] and C++'s value objects [65], which enable an object to be encapsulated in another object, but require a fixed sized object. In the following class

```
class Link { owner::Link next; int data; }
```

the `next` object has the same owner as the present object. This is a common idiom, and we call such objects *siblings*. (The Universes system [52] uses the keyword `peer` instead of `owner`.)

2.2 External Uniqueness

Object sharing can be avoided using unique or linear references [3, 10, 13, 22, 35, 49]: at any point in the execution of a program, only one *accessible* reference to an object exists. Clarke and Wrigstad introduced the notion of *external uniqueness* [22,68] which gels nicely with ownership types and permits unique references to aggregate objects that are inherently aliased, such as circularly linked lists. In external uniqueness, unique references must be (temporarily) made non-unique to access or call methods on fields. The single external reference is thus the only active reference making the aggregate effectively unique. External uniqueness enables ownership transfer in ownership types systems.

External uniqueness is effectively equivalent to introducing an owner for the field or variable holding a reference into the data structure, such that the only occurrence of that owner is in the type of the field or variable. In the code below, `first` holds the only pointer to the (sibling) link objects.

```
class List { unique::Link first; }
```

External uniqueness can be maintained by destructive reads, or techniques such as Alias Burying [13].

[1] In this section, code uses syntax from Joe-like languages [19, 22, 58].

2.3 Immutability and 'Safe' Methods

Immutable objects can never change after they are created. An *immutable reference* prevents the holder from calling methods that mutate the target object. Furthermore, references to representation objects returned from a method call via an immutable reference are also immutable—or immutability would be lost. *Observational exposure* [14] occurs when an immutable reference can be used to observe changes to an object, which is possible if non-immutable aliases exist to the object or its representation. Fortunately, strong encapsulation, uniqueness, and *read-only* methods make the (staged) creation of "truly immutable" objects straightforward [58]. This is similar to Fähndrich and Xia's recently proposed Delayed Types [28].

In Flexible Alias Protection [56], '*arg*' or *safe references* (our preferred terminology) to an object may only access immutable parts of the object; i.e., the parts which do not change after initialisation. Thus, clients accessing an object via a safe reference can only depend on the object's immutable state, which is safe as it cannot change unexpectedly. Safe references can refer to *any* object, even one which is being mutated by a different active object, without any risk for observational exposure.

2.4 Owner-Polymorphic Methods

Owner-polymorphism is crucial for code reuse and flexibility in the ownership types setting [18,68]. *Owner-polymorphic methods* are parametrised with owners to give the receiver temporary permission to reference an argument object. For example, the following method accepts an owner parameter `foo` in order to enable a list owned by any other object to be passed as an argument:

```
<foo> int sum(foo::List values) { ... }
```

Clarke [18] established that owner-polymorphic methods can express a notion of *borrowing*: an object may be passed to another object, which does not own it, without the latter being able to capture a reference to the former. (For further details, see [68].) Owner-polymorphic methods are reminiscent of region-polymorphic procedures in Cyclone [32].

3 Active Objects and Data Sharing

Active objects interacting through *asynchronous method calls* (e.g., [17,40,48]) have been proposed as a concurrency model that gels better with object-oriented programming. Such active object proposals aim to hide much of the complexity of concurrent and distributed programming by using object-internal threads and single-thread-per-object invariants. Several slightly differently flavoured active objects systems exist for languages such as Java [7], Eiffel [16,53] and C++ [50].

The primary goal of this work is to design an ownership types system for active objects to control data races and data sharing while avoiding cloning where possible. We also want to provide a sensible default semantics for completely unannotated classes to facilitate reuse of existing classes as passive objects in the setting of active objects.

3.1 Motivating Controlled Data Sharing

In *Concurrent programming in Java* [44], Doug Lea writes:

> *To guarantee safety in a concurrent system, you must ensure that* all *objects accessible from multiple threads are either immutable or employ appropriate synchronization, and also must ensure that* no *other object ever becomes concurrently accessible by leaking out of its ownership domain.*

The simple ways to guarantee the above are the first two we listed in Section 2: making everything immutable or use deep copying semantics. While efficient, the first is both severely restrictive and requires careful inspection of the code to determine that the messages are truly immutable. The second is easier to check, just check for clone operations on arguments, but has the downside of adding potentially massive copying overhead.

We argue that the most effective approach is the pragmatic combination: using ownership transfer or immutable objects where possible, and deep copying only when there is no other way. To enable this in a statically and modularly checkable fashion, we use a few extra annotations on interfaces of active object classes. We believe that inserting these extra annotations will be helpful for maintenance and possibly also refactoring. Most importantly, we believe that the static checking enabled by the annotations will save time, both programmer-time and run-time.

Back in the active object setting, assume that there are two active objects `Client` and `Broker` and let `Client` create a complex data structure `Request`, which `Broker` shall match with an `Offer`. If `Request` is cloned when passed from `Client` to `Broker`, two potential problems arise. Firstly, there is the cost of cloning it (and potentially the cost of passing it back via cloning). Secondly, the two copies of the request must be synchronised if changed concurrently. The situation becomes even worse for complex networks of active objects, leading to additional copying.

4 Active Ownership

This section describes our ownership types system for controlled data sharing in active objects. Our active objects model is shared with ProActive [7] (modulo minor things unimportant in this context), which is an implementation of ASP active objects [17] as a Java framework. Our system should however be relatively straightforward to adapt to, for example, SCOOP [48] or Emerald [38]. We phrase our contribution directly in terms of a Java-like language formalised on top of FJ [39] for generality and to simplify the presentation. To give a flavour for how

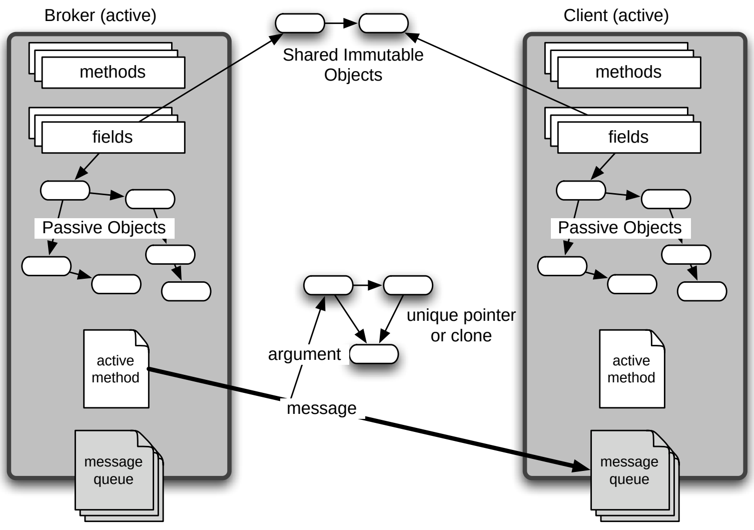

Figure 1. Active Ownership. Safe references are not depicted. Broker and Client are active objects from the code example in Figure 2.

unobtrusive the extension would be, code examples use annotations for owners to completely fit in the Java syntax, and we design default annotations to minimize the syntactic overhead.

Our alias control mechanisms uphold the invariant that no two 'threads' concurrently change or observe changes to an object, which is the invariant obtained by the deep copying of message arguments in ProActive (with minor exceptions unimportant to us here).

We now present the features of our type system and the constraints it imposes, along with justification for these constraints. The elements of our proposal are shown in Figure 1.

4.1 Active and Passive Classes

Active and passive objects are instantiated from active and passive classes respectively. These are declared as follows:

Active classes are annotated with a `@active` annotation and have an optional `run` method which is activated immediately after the active object is constructed.

Passive classes are *not* annotated with `@active`. Thus, classes are passive by default; e.g., the classes in the Java libraries are all considered passive without annotating them.

Active objects primarily interact via asynchronous method calls. These are default for active objects and return a future. A future is a placeholder for a value which need not be currently available. For asynchronous calls, the future is the placeholder for the methods' actual return values. Thus, the caller need not wait for the call to complete. A future's value is accessed by the future's `get` method,

```
@active class Client {
   void run() {
      Request rm = ...; // formulate Request
      @future Offer offer = myBroker!book(rm.clone());
      ... // evaluate offer
  offer.getProvider().accept(offer.clone());
   }
}
@active class Broker {
   void run() { ... } // go into reactive mode
   // book returns first Offer that responds to the request
   Offer book(Request request) { ... }
}
@active class Provider {
   void run() { ... } // go into reactive mode
   Offer query(Request request) { ... }
   boolean accept(Offer offer) { ... }
}
class Request {
   Request(String desc) { ... }
   void markAccepted() { ... }
}
class Offer {
   Offer(Details d, Provider p, Request trackback) { ... }
   Provider getProvider() { ... }
}
```

Figure 2. Example of active objects exchanging arguments by copying. Here `@future Offer` denotes a future of type `Offer`. For clarity, we use a `!`-notation on asynchronous method calls, e.g., `myBroker!book(rm)`.

which blocks until the future has a value. Synchronous calls may be encoded by calling `get` directly after asynchronous method calls. Method calls to passive objects are always synchronous; i.e., they are similar to standard method calls as found in Java.

Figure 2 shows a use of active objects that deliberately copy arguments. Later we show how this copying is avoided, see Figure 3. For brevity, we focus simply on the interfaces, which suffices for the type annotations. The figure shows the following scenario:

1. Client sends request to broker
2. Broker forwards request to provider(s) and negotiates a deal
3. Broker returns resulting offer to client
4. If client accepts offer, client sends acceptance to provider

The client, the broker, and all providers are represented as active objects and execute concurrently. In contrast, requests and offers are passive objects, passed between active ones by copying to avoid data races between the active objects.

4.2 Language Constructs for Active Ownership

This section describes our language features leading up to an encoding of the example from Figure 2 that avoids copying.

Ownership structure The type system presented has the following owners:

`@active`	globally accessible owner of all active objects
`@owner`	the object owning the current `this`
`@this`	owner denoting the current `this`
`@unique`	owner denoting the current field or variable
`@immutable`	globally accessible owner of all immutable objs
`@safe`	globally accessible owner allowing safe access

The owners `@active`, `@unique`, `@immutable`, and `@safe` are available in any context, and denote the global owner of active objects, unique references, immutable references, and safe references, respectively. Nested inside each active object is a collection of passive objects, owned by the active object with owner `@this`. The owner `@owner` is available only in passive classes only for referring to the owner of the current instance, and is used to create linked data structures within an active object.

Note that the ownership hierarchy is very flat, as there is no owner `@this` inside a passive class. Ownership encapsulates passive objects inside an active object. Consequently, there is no need to keep track of nesting or other relationships such as links between owners [2]. In addition, the classes in this system take no owner parameters, in contrast to the original ownership types system [23]. Therefore no run-time representation of ownership is required [69].

Immutable and Safe References Immutable types have owner `@immutable`, which is valid everywhere. In our system, only passive objects can have immutable type. Fields or variables containing immutable references are not final unless explicitly declared final or if the container enclosing the field is immutable. Only *read-only* and *safe* methods (see below) can be called on immutable objects, which preserves immutability.

Safe references (called *argument* references in Flexible Alias Protection [56]) have owner `@safe` and can be used only to access the final fields of an object, and the final fields of the values returned from methods, and so forth. These parts of an object cannot be changed underfoot. Methods that object these conditions are called safe methods, denoted by a `@safe` annotation. Any non-active type can be subsumed into a `@safe` type.

Our immutable references must be created from unique objects. Essentially, immutability is achieved through invalidating all references to an object with a mutating capability. This is powerful and flexible as it allows a single class to be used both as a template for both mutable and immutable objects (see Clarke and Wrigstad's original external uniqueness paper [22]) and staged construction. Effectively, immutability becomes a property of the object, rather than of the class or references.

In contrast to immutables, a safe reference does not preclude the existence of references with a mutating capability to the same object. A safe reference merely limits what operations can be performed on the referenced object via the reference in terms of mutation and observing an object's state. Both kinds of references avoid observational exposure.

Read-only and Safe Methods Following previous read-only proposals, e.g., [9,14,35,63], a read-only method preserves the immutability of objects, and does not return non-immutable references to otherwise immutable objects. Read-only methods cannot update any object with owner `@owner`, which notably includes the receiver. They are not, however, purely functional: they can be used to modify unique references passed in as arguments or objects freshly created within the method itself and they can call mutating methods on active objects.

By virtue of immutability encoded in the owners, a return value from a read-only method that has owner `@owner` will (automatically) have the owner `@immutable` when the read-only method is called on an immutable reference, and hence will not provide a means for violating the immutability of the original reference [58, 71]. In order to allow modular checking, read-only methods are annotated with `@read`.

A safe method, annotated `@safe`, is an additionally restricted read-only method that may only access immutable parts of the receiver's state, i.e., final fields containing safe or immutable references. Conceptually, a read-only method prevents mutation whereas a safe method also prevents the observation of mutation.

4.3 Data Transfer and Minimal Cloning

To ensure that the data race freedom invariant is preserved, care is needed when passing data between active objects. How data is passed, will depend on the owner of the data:

`@active`	pass by reference
`@this`	object must be cloned
`@owner` (in passive object)	object must be cloned
method owner parameter	object must be cloned
`@unique`	pass by reference (destructively)
`@immutable` or `@safe`	pass by reference

Active objects are safe to pass by reference as external threads of control never enter them by virtue of asynchronous methods calls and futures. Immutable and safe objects are obviously safe to pass by reference as their accessible parts cannot be changed. Uniques are safe to pass by reference as they are effectively transferred to the target.

Other objects must be cloned. Cloning returns a unique reference which can be moved across regardless of the owner of the expected parameter type.

Using the owner annotations, it is possible to statically infer what parts of an object that must be further cloned to be safely passed to another active object. The inferred (generated or generic reflexive) clone operation does not differ significantly from standard serialisation algorithms in that it preserves the underlying object graph structure, and is similar to the *sheep clone* described by Noble et al. [18, 55] for ownership types and Nienaltowski's object import [53]. The clone operation behaves as follows on references:

`@active`	reference is copied
`@this`	object is cloned
`@owner` (in passive object)	object is cloned
method owner parameter	object is cloned
`@unique`	object is cloned
`@immutable` or `@safe`	reference is copied

Reducing Syntactic Baggage We adopt a number of reasonable defaults for owner annotations to reduce the amount of annotations required in a program, and to use legacy code immediately in a sensible way.

Passive Classes (including all library code) have one implicit owner parameter `@owner`, which is the default owner of all fields and all method arguments. Note that this means that library code, in general, requires no annotations.

Active Classes have the implicit owner `@active`. In an active class, the default owner is `@this` for all fields and `unique` for all method arguments.

Together these defaults imply that all passive objects reachable from an active object's fields are encapsulated inside the active object, in the absence of immutable and safe references. By default, all method parameters in the public interface of active objects are `@unique`. This is the only way to guarantee that *mutable* objects are not shared between active objects. References passed between active objects must be unique, either originally or as a result of performing a clone. Note that this default annotation as `@unique` does not apply to active class types appearing in the interface, as these can only be `@active`. This choice of defaults is supported by results of Potanin and Noble [59] and Ma and Foster [46], which indicate that many arguments between objects could well be unique references.

Benchmarks of non-trivial programs performed by Carlsson et al. [15] for a related system, albeit in the context of Erlang, show that avoiding unnecessary copying of arguments in message passing can have a measurable effect on the overall performance of an application. We suspect that our story here is even slightly stronger, since we would not need any dynamic checks to determine correctness at run-time. Importantly, both systems guaranteed non-sharing of message objects at compile-time.

4.4 Revisiting the example

Figure 3 adds active ownership annotations to Figure 2 in order to avoid *all* copying. Only six annotations are needed to express the intended semantics of Figure 2. This might seem excessive for a 20-line program, but remember that

```
@active class Client {
   void run() ‡ {
      †Request rm = ...; // formulate Request
      @future @immutable Offer offer = myBroker!book(rm); // (1)
      ... // evaluate offer
      offer.getProvider().accept(offer); // (2)
   }
}
@active class Broker {
   void run() ‡ { ... } // go into reactive mode
   // book returns first provider that responds to the request
   Offer book(@safe Request request) ‡ { ... } // (3)
}
@active class Provider {
   @immutable Offer query(@safe Request rq) ‡ { ... } // (4)
   boolean accept(@immutable Offer offer) ‡ { ... } // (5)
}
class Request {
   Request(†String desc) ‡ { ... }
   void markAccepted() ‡ { ... }
}
class Offer {
   Offer(†Details d, †Provider p, @safe Request r) ‡ ... // (6)
   Provider getProvider() @read { ... } // (7)
}
```

Figure 3. The active objects example with active ownership. † indicates an implicit use of a default, `@owner` in passive classes and `@this` in active. ‡ indicates an implicit use of the `@write` default for methods. These are not part of the actual code.

this is just the interface, and that the actual code for this program would be much longer. Furthermore, no annotations are required for library code used by this program.

The offer is made immutable (4), which allows it to be safely shared between concurrently executing clients, brokers and providers. This propagates to the type of the future variable (1) and formal parameter (5). The request is received as a safe reference (3), so the broker may only access its immutable parts which precludes both races and observational exposure. This constraint is reasonable, since changing crucial parts of a request under foot might lead to invalid offers. Parts of the request can still be updated by the client (but not by the broker or any provider), e.g., to store a handle to the accepted offer in it. The safe annotation propagates to (6). Read-only methods are annotated `read` (7). Reading the provider from an immutable offer (2) returns an immutable reference, since `offer` is immutable (1).

4.5 Other Relevant Features

Here we briefly discuss other language features that can be bundled with our type system to increase expressiveness. For space reasons, we omit a discussion of exceptions, which is a straightforward addition (they would be immutable).

Owner-polymorphic methods (and their problems) The previous discussion ignored owner-polymorphic methods. An owner-polymorphic method of an active object enables the active object to borrow passive objects from another active object, with the guarantee that it will not keep any references to the borrowed objects. Such methods require care, as they are problematic in the presence of asynchronous method calls. It is easy to see that an asynchronous call could easily lead to a situation where two active objects (`Client` and `Broker`) have access to the same passive objects (`Request`):

1. `Client` lends `Request` to `Broker` via an asynchronous method call.
2. `Client` continues executing on `Request`.
3. `Broker` concurrently operates on `Request`.

We choose the simplest solution to avoid this problem by banning asynchronous calls to owner-polymorphic methods. Alternative approaches would require preventing `Client` from accessing `Request`—or more precisely, objects with the same owner as `Request`—until the method call to `Broker` returned. This can be achieved using an `await` on the future returned by the asynchronous call to the method on `Broker`.

Dealing with Globals The presence of globally accessible fields and methods can be problematic in the active object model as they introduce opportunities for sharing passive objects across active ones. If not treated carefully, they can result in potential race conditions. Our solution is reminiscent of that of Scala [57] where globally accessible singleton objects can be declared directly through the `object` keyword. For a Java program the static fields of non-primitive type and static methods of every class needs to be moved to an active singleton object. Thus, they remain globally accessible, but protected from race conditions as they are encapsulated in an active object. For Java, this translation is straightforward: All accesses to static fields and methods need to be modified to use the newly created object. Our treatment of global variables is comparable to an approach in Lea's book in the context of Java [44, page 85].

5 The Language and its Type System

This section introduces Joëlle, a kernel language reminiscent of ProActive. It extends Featherweight Java (FJ) [39], combining active objects, asynchronous method calls and futures, with ownership, uniqueness, and immutability. Joëlle-objects encapsulate their state—the fields of an object are accessible only via the

$$
\begin{array}{rcl}
\mathit{Prg} & ::= & \overline{L}\ \{\ \overline{T}\ \overline{x}\texttt{;}\ \mathit{sr}\ \} \\
L & ::= & [\texttt{active}]\ \texttt{class}\ C\ \texttt{extends}\ C\ \{\ \overline{T}\ \overline{f}\texttt{;}\ \overline{\texttt{final}}\ \overline{T'}\ \overline{f'}\texttt{;}\ \overline{M}\ \} \\
M & ::= & \texttt{<}\overline{a}\texttt{>}\ T\ m\ (\overline{T}\ \overline{x})\ P\ \{\ \overline{T'}\ \overline{y}\texttt{;}\ \mathit{sr}\ \} \\
P & ::= & \texttt{write} \mid \texttt{read} \mid \texttt{safe} \\
\mathit{lval} & ::= & f \mid x \\
e & ::= & \mathit{lval} \mid e\texttt{.get} \mid e\texttt{.uget} \mid e!m\,(\overline{e}) \mid e.\texttt{<}\overline{o}\texttt{>}m\,(\overline{e}) \\
 & & \mid \texttt{new}\ C\,() \mid \texttt{null} \mid \mathit{lval}\texttt{--} \mid \texttt{clone}\ e \\
s & ::= & \mathit{lval} := e \mid s\texttt{;}\ s \mid \texttt{borrow}\ \mathit{lval}\ \texttt{as}\ x, a\ \{\ s\ \} \\
\mathit{sr} & ::= & s\texttt{;}\ \texttt{return}\ e \\
T & ::= & o :: C \mid !T \\
o & ::= & \texttt{active} \mid \texttt{owner} \mid \texttt{this} \mid \texttt{unique} \mid \texttt{immutable} \mid \texttt{safe} \mid a
\end{array}
$$

Figure 4. The language syntax. Variables *lval* are fields (f) or local variables (x), and C is a class name. Owner variables are denoted a. Overline, $\overline{x}$, indicates tuples of features, following Featherweight Java, and brackets [*opt*] indicate optional features.

object's methods. Objects execute concurrently and each active object has its own thread. The type system is a synthesis of the type systems of FJ and ClassicJava [31,39], Creol [25], External Uniqueness [22,68], and Joe$_3$ [58]. To simplify the presentation of the formalism, we assume that all default annotations on types and methods have been elaborated.

Just like in ProActive and SCOOP, we restrict our concurrency model so that only one thread may execute be active in an active object at a time; other threads in the object are *suspended.*[2] We distinguish between *blocking* a thread and *releasing* a thread. Method calls to active objects are asynchronous and the result is stored in a *future.* Forcing the caller to *wait* for the call to return is unsatisfactory in a distributed setting where communications may disappear and block the caller's process permanently. Execution is blocked only when attempting to read from a future, using `get`, which does not yet have an assigned value.

5.1 Syntax

The language syntax is given in Figure 4. A program *Prg* is a list of class definitions, followed by a method body which acts like the `main` method. A class L inherits from a superclass, which may be `Object`, extending it with additional fields f and methods M. For æsthetic and spatial reasons, we lose the @-sign on the annotations, so active classes are indicated by the keyword `active`, etc.

Methods M can be owner-polymorphic. For simplicity, we require owner parameters to be expressly declared using $<\overline{a}>$, although this is not strictly needed since owners don not have a nesting relation.

A method annotated with `read` or `safe` does not modify its target but may modify its arguments, depending upon their types. Additionally, `safe` methods

[2] ProActive has a couple of exceptions to this; e.g., for methods that throw checked exceptions. This seems to be purely due to limitations in the underlying Java language.

may depend on immutable state. Only `read` and `safe` methods may be called on `immutable` references, and only `safe` methods may be called on `safe` references. A method body may have a number of local variables, as expected.

Expressions e are mostly standard. Asynchronous method calls are denoted by $e!m\,(\overline{e})$ and the (blocking) read operations by $e.\texttt{get}$ and $e.\texttt{uget}$ (the latter destructively reads the future). Although our work on a compiler for Joe$_3$ has shown us that parameters to owner-polymorphic methods in general can be inferred, we require synchronous local method calls $e.\texttt{<}\overline{o}\texttt{>}m\,(\overline{e})$ to take owner explicit arguments $\overline{o}$ whenever the method is owner-polymorphic to simplify the formalism. *lval*-- destructively reads a unique field or variable, and `clone` e clones the result of an expression. Both result in unique references.

Statements s include borrowing blocks, `borrow` *lval* `as` x, $a :: C$ { s }, which enable a unique reference to an aggregate object to be temporarily treated non-unique for the duration of statement `s`. This allows manipulation of the aggregate object without violating uniqueness [22, 68].

Futures are given a value once (under the hood) and this value remains for its lifetime.

Types consist of either an annotated class type or a future type, denoted **fut**(T), a future of type T. Type annotations indicate owner or access mode. An annotation on a future type determines how the future is treated. For example, a variable typed **fut**(`unique` $:: C$) must be treated as a unique reference. The owners is described in Section 4.2.

5.2 Static Semantics

We now briefly present the static semantics of Joëlle using the auxiliary functions and judgements surveyed in Figure 5.

The auxiliary definitions, shown in Figure 6, are slightly adapted from FJ, extended with owner and permission annotations on methods. Functions *callok*, *writeok*, and *readok* check that a method call is correct with respect to access modes, that member fields are only written in write methods, and that accessed fields are final in safe methods, respectively. For brevity we omit trivial functions, such as *final* for testing finality of a field.

Figure 7 shows the rules for good environments, owners, types, and subtyping. The typing environment Γ binds variables to their type and records the owners in scope (with 'type' $*$). P denotes the permissions of the current context: `read`, `write`, or `safe`. Figures 8 and 9 show the static semantics. The standard rules should be straightforward. Statements and expressions that mutate the heap need a mutable context; i.e., $P =$ `write`. Fields accessed in a `safe` method must be final. This is ensured by predicates *writeok* and *readok*, respectively.

(Stat-Borrow) introduces an owner a, allowing the unique l-value to be stored in x as a normal reference for the scope of the block. (Expr-Exact) uses the `exact` annotation on types (in type rules only) so that subsumption does

$\mathit{fields}(C) = \overline{T}\ \overline{f}$	the fields of class C and their types
$\mathit{mtype}(m, C) = \cdots$	return/argument types, permission and owner parameters of method m in class C
$\mathit{mbody}(m, C) = \cdots$	the permission and body of method m in class C
$\mathit{override}(m, D, \cdots)$	governs OK overriding (= preserve types)
$\mathit{callok}(P, P', o)$	calling a method on o with permission P' is ok, given permission P
$\mathit{writeok}(P, \mathit{lval})$	reading *lval* is ok, given permissions P
$\mathit{readok}(P, \mathit{lval})$	writing to *lval* is ok, given permissions P
$\Gamma \vdash \diamond$	good environment Γ
$\Gamma \vdash o : *$	good owner o
$\Gamma \vdash T$	good type T
$\Gamma \vdash T \preceq T'$	T is a subtype of T'
$\vdash \mathit{Prg}$	good program *Prg*
$\vdash L$ `ok`	good class L
$\Gamma \vdash M$ `ok` $\in C$	good method M in class C
$\Gamma \vdash \mathit{lval} : T$ `exact`	T is exact type *lval*
$\Gamma; P \vdash s$ `ok`	good statements s, given permissions P
$\Gamma; P \vdash e : T$	expression e has type T, given permissions P

Figure 5. Judgements for Joëlle's semantics. The permissions P govern the necessary permission to call a certain method or write to or read from a field. Note that exact types are required for type soundness when assigning to *lval*s and in borrowing blocks, which do a read and an assign to an *lval*.

not change the type. Ultimately, this is to ensure that assignment to an *lval* is type correct and to ensure type soundness when using borrowing blocks, as the borrowed value in a borrowing block is read from and written to the same *lval*. Distinguishing between *lval*s as l-values and expressions is required to preserve uniqueness (see [22, 68]). Asynchronous calls are only allowed on active objects by (Expr-Call-Async) and synchronous calls only on passive objects by (Expr-Call-Sync). In (Expr-Call-Sync), *callok* prevents calling mutating methods on immutable references, etc. Adding an additional type rule to allow synchronous calls of `safe` methods on active objects is straightforward.

By (Expr-New), instantiation returns an externally unique reference. Thus, creating an immutable object is simply storing the result of an instantiation in a variable or field with immutable owner. (Expr-Dread) destructively reads a unique *lval* into a free value. Destructively reading a field is only allowed in a write method. (Expr-Clone) clones an object and returns a (externally) unique reference to the clone.

As dealing with finals properly is orthogonal to our proposal we simply treat reading uninitialised final fields and reassigning final fields as errors.

Auxiliary Functions ($\mathit{fields}(\texttt{Object}) = \epsilon$)

$$\frac{CT(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \texttt{\{}\ \overline{T\ f\texttt{;}}\ \overline{\texttt{final}\ T'\ f'\texttt{;}}\ \overline{M}\ \texttt{\}}}{\mathit{fields}(C) = \mathit{fields}(D), \overline{T\ f}, \overline{T'\ f'}}$$

$$\frac{\begin{array}{c}CT(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \texttt{\{}\ \cdots\texttt{;}\ \overline{M}\ \texttt{\}}\\ \texttt{<}\overline{a}\texttt{>}\ T\ m\,\texttt{(}\overline{T\ x}\texttt{)}\ P\ \texttt{\{}\ \cdots\ \texttt{\}} \in \overline{M}\end{array}}{\mathit{mtype}(m, C) = (\overline{a}, P, \overline{T} \to T)} \qquad \frac{\begin{array}{c}CT(C) = \texttt{class}\ C\,\texttt{ext.}\ D\,\texttt{\{}\ \cdots\texttt{;}\ \overline{M}\ \texttt{\}}\\ m \text{ is not defined in } \overline{M}\end{array}}{\mathit{mtype}(m, C) = \mathit{mtype}(m, D)}$$

$$\frac{\begin{array}{c}CT(C) = \texttt{class}\ C\ \cdots\ \texttt{\{}\ \cdots\texttt{;}\ \overline{M}\ \texttt{\}}\\ \texttt{<}\overline{a}\texttt{>}\ T\ m\,\texttt{(}\overline{T\ x}\texttt{)}\ P\ \texttt{\{}\ \overline{U\ y}; sr\ \texttt{\}} \in \overline{M}\end{array}}{\mathit{mbody}(m, C) = (P, \overline{T\ x}, \overline{U\ y}, sr : T)} \qquad \frac{\begin{array}{c}CT(C) = \texttt{class}\ C\,\texttt{ext.}\ D\,\texttt{\{}\ \cdots\texttt{;}\ \overline{M}\ \texttt{\}}\\ m \text{ is not defined in } \overline{M}\end{array}}{\mathit{mbody}(m, C) = \mathit{mbody}(m, D)}$$

$$\frac{\begin{array}{c}\mathit{mtype}(m, D) = (\overline{a}, P, \overline{U} \to U)\\ \Rightarrow (\ \overline{a} = \overline{a'}\ \wedge\ P = P'\ \wedge\ \overline{T} = \overline{U}\ \wedge\ T = U\ )\end{array}}{\mathit{override}(m, D, \overline{a'}, P', \overline{T} \to T)}$$

$$\frac{}{\mathit{callok}(P, \texttt{safe}, \texttt{safe})} \qquad \frac{P' \leq \texttt{read}}{\mathit{callok}(P, P', \texttt{immutable})} \qquad \frac{o \in \{\texttt{this}, \texttt{owner}\} \quad P' \leq P}{\mathit{callok}(P, P', o)}$$

$$\frac{}{\mathit{callok}(P, P', a)} \qquad \frac{\neg\mathit{final}(f)}{\mathit{writeok}(\texttt{write}, f)} \qquad \frac{}{\mathit{writeok}(P, x)} \qquad \frac{P = \texttt{safe} \Rightarrow \mathit{final}(f)}{\mathit{readok}(P, f)} \qquad \frac{}{\mathit{readok}(P, x)}$$

Figure 6. Lookup functions adopted (and slightly modified) from Featherweight Java. We assume the following ordering on P: $\texttt{safe} < \texttt{read} < \texttt{write}$. $\Rightarrow$ is implication.

6 Dynamic Semantics

This section presents Joëlle's dynamic semantics, based on our work on the active object language Creol [25], and on the formal semantics of ownership and external uniqueness due to Wrigstad [68].

6.1 Configurations and Reduction Rules

The semantics is a small-step reduction relation on *configurations* of objects, futures, and owner-bindings (see Fig. 10). An *object* has an id, an indication of whether it is active or passive, a type consisting of an owner and a class, some fields, and a collection of running and sleeping *processes* for the method invocations on the object. We extend the syntax of statements s (and hence sr) to include the stack as part of the expression in order to track the value of local variables:

$$s ::= \cdots \mid T\ x\ v;\ s$$

This states that the statement s has stack variable x of type T storing value v. We also add the usual booleans.

A *future* captures the state of an asynchronous method call: initially *sleeping*, the call later becomes *active*, and finally, when *completed*, it stores its result in the future. The value $\mathit{mode} \in \{\mathsf{s}, \mathsf{a}, \mathsf{c}\}$ represents these future states. Types get

Good Environment ($\Gamma ::= \epsilon \mid \Gamma, \mathit{lval} : T \mid \Gamma, o : *$)

$$\overset{(\textsc{Env-}\epsilon)}{\frac{}{\epsilon \vdash \diamond}} \qquad \overset{(\textsc{Env-lval})}{\frac{\Gamma \vdash T \quad \mathit{lval} \notin \mathrm{dom}(\Gamma)}{\Gamma, \mathit{lval} : T \vdash \diamond}} \qquad \overset{(\textsc{Env-Owner})}{\frac{\Gamma \vdash \diamond \quad o \notin \mathrm{dom}(\Gamma) \quad o \in \{a, \mathtt{owner}, \mathtt{this}\}}{\Gamma, o : * \vdash \diamond}}$$

Good Owner

$$\overset{(\textsc{Owner})}{\frac{\Gamma \vdash \diamond \quad o : * \in \Gamma}{\Gamma \vdash o : *}} \qquad \overset{(\textsc{Owner-Other})}{\frac{\Gamma \vdash \diamond \quad o \in \left\{\begin{matrix}\mathtt{immutable}, \mathtt{safe}, \\ \mathtt{unique}, \mathtt{active}\end{matrix}\right\}}{\Gamma \vdash o : *}}$$

Good Type

$$\overset{(\textsc{Type-Owned})}{\frac{\Gamma \vdash o : * \quad [\mathtt{active}]\ \mathtt{class}\ C\ \cdots \in CT}{\Gamma \vdash o :: C}} \qquad \overset{(\textsc{Type-Future})}{\frac{\Gamma \vdash T}{\Gamma \vdash !T}}$$

Good Subtyping

$$\overset{(\textsc{Sub-Class})}{\frac{\Gamma \vdash o : * \quad [\mathtt{active}]\ \mathtt{class}\ C\ \mathtt{extends}\ D \cdots \in CT}{\Gamma \vdash o :: C \preceq o :: D}} \qquad \overset{(\textsc{Sub-Unique})}{\frac{\Gamma \vdash o :: C \quad o \neq \mathtt{active}}{\Gamma \vdash \mathtt{unique} :: C \preceq o :: C}}$$

$$\overset{(\textsc{Sub-Safe})}{\frac{\Gamma \vdash o :: C \quad o \neq \mathtt{active}}{\Gamma \vdash o :: C \preceq \mathtt{safe} :: C}} \qquad \overset{(\textsc{Sub-Refl})}{\frac{\Gamma \vdash T}{\Gamma \vdash T \preceq T}} \qquad \overset{(\textsc{Sub-Trans})}{\frac{\Gamma \vdash T \preceq T' \quad \Gamma \vdash T' \preceq T''}{\Gamma \vdash T \preceq T''}} \qquad \overset{(\textsc{Sub-Future})}{\frac{\Gamma \vdash T \preceq T'}{\Gamma \vdash !T \preceq !T'}}$$

Figure 7. Well-formed Environments, Owners, Types and Subtyping.

default values by the *default* function (e.g., $\mathit{default}(C) = \mathtt{null}$, $\mathit{default}(\mathtt{bool}) = \mathtt{false}$, and $\mathit{default}(\mathtt{fut}(T)) = \mathtt{null}$). The *initial configuration* of a program $\overline{L}\ \{\overline{T}\ \overline{x}; \mathit{sr}\}$ has one object $(o, \emptyset, \emptyset, \overline{T}\ \overline{x}\ \mathit{default}(\overline{T}); \mathit{sr} : T)$. The default mode, κ, for final fields is u, indicating that they are uninitialised, and for other fields the mode is v, virginal.

The modes κ serve a dual purpose. Firstly, they are used to dynamically track proper (one-time) intialisation of final fields are initialised before use, as in Tribe [20]. Secondly, κ tracks whether a non-final field has been read or written.

An object is virginal if all its (non-final) fields are marked virginal. A configuration is virginal if all of its constituents are.

Reduction takes the form of a relation $\mathit{config} \rightarrow \mathit{config}'$. Rules apply to partial configurations and may be applied in parallel. This differs from the semantics of object-oriented languages with a global store [31], but it allows true concurrency and enables reasoning about data races. The key rules are given in Fig. 11. The context reduction semantics decomposes a statement into a reduction context and a redex, and reduces the redex [29]. *Reduction contexts* are process sets Q, method bodies M, statements S, and expressions E *with* a single hole denoted by $\bullet$:

Good Program

$$\frac{Prg = \overline{L}\ \{\overline{T}\ \overline{x}; s; \texttt{return}\ e\} \quad \vdash \overline{L}\ \texttt{ok} \qquad \overline{x} : \overline{T}; \texttt{write} \vdash s\ \texttt{ok} \quad \Gamma'; \texttt{write} \vdash e : T}{\vdash Prg}$$

Good Class

$$\frac{\neg \mathit{active}(D) \quad \mathit{fields}(C) = \overline{U}\ \overline{g} \qquad \Gamma = \overline{g} : \overline{U}, \texttt{owner} : *, \texttt{this} : \texttt{owner} :: C \quad \Gamma \vdash \overline{M}\ \texttt{ok} \in C}{\vdash \texttt{class}\ C\ \texttt{extends}\ D\ \{\ \overline{T}\ \overline{f}\texttt{;}\ \overline{\texttt{final}}\ \overline{T'}\ \overline{f'}\texttt{;}\ \overline{M}\ \}\ \texttt{ok}}$$

$$\frac{\mathit{active}(D) \quad \mathit{fields}(C) = \overline{U}\ \overline{g} \qquad \Gamma = \overline{g} : \overline{U}, \texttt{this} : *, \texttt{this} : \texttt{active} :: C \quad \Gamma \vdash \overline{M}\ \texttt{ok} \in C \qquad \texttt{<>}\ T\ \texttt{run()}\,\texttt{write\{}\ \cdots\ \texttt{\}} \in \overline{M} \text{ for some } T}{\vdash \texttt{active class}\ C\ \texttt{extends}\ D\ \{\ \overline{T}\ \overline{f}\texttt{;}\ \overline{\texttt{final}}\ \overline{T'}\ \overline{f'}\texttt{;}\ \overline{M}\ \}\ \texttt{ok}}$$

Good Method

$$\frac{P = \texttt{safe} \implies |\overline{a}| = 0 \quad \Gamma' = \Gamma, \overline{a} : *, \overline{x} : \overline{T}, \overline{y} : \overline{U} \qquad \Gamma'; P \vdash s\ \texttt{ok} \quad \Gamma'; P \vdash e : T \qquad CT(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\ \ldots\ \} \qquad \mathit{override}(m, D, \overline{a}, P, \overline{T} \to T)}{\Gamma \vdash \texttt{<}\overline{a}\texttt{>}\ T\ m\,\texttt{(}\overline{T}\ \overline{x}\texttt{)}\ \ P\ \{\ \overline{U}\ \overline{y}; s; \texttt{return}\ e\ \}\ \texttt{ok} \in C}$$

$$\frac{\Gamma' = \Gamma, \overline{x} : \overline{T}, \overline{y} : \overline{U} \quad \Gamma'; P \vdash s\ \texttt{ok} \quad \Gamma'; P \vdash e : T \qquad CT(C) = \texttt{active class}\ C\ \texttt{extends}\ D\ \{\ \ldots\ \} \qquad \mathit{override}(m, D, \epsilon, P, \overline{T} \to T)}{\Gamma \vdash T\ m\,\texttt{(}\overline{T}\ \overline{x}\texttt{)}\ \ P\ \{\ \overline{U}\ \overline{y}; s; \texttt{return}\ e\ \}\ \texttt{ok} \in C}$$

Figure 8. Good Declarations.

$$\begin{aligned}
Q &::= (P, \text{run}, M)\ \mathit{processes}\\
M &::= \bullet \mid S; \texttt{return}\ e \mid \texttt{return}\ E\\
S &::= \bullet \mid T\ x\ v; S \mid \mathit{lval} := E \mid S; s \mid \mathit{BB}(\mathit{lvar})\{S\}\\
E &::= \bullet \mid E.\texttt{get} \mid E.\texttt{uget} \mid E!m(\overline{e}) \mid v!m(\overline{v}, E, \overline{e})
\end{aligned}$$

Although our reduction context notion is standard, Q deserves explaining. It is used to denote the location of a redex in some method body within the set of *processes*.

As statements are evolving, we have incorporated stack frames $T\ x\ v; S$ and a run-time representation of the borrowing construct $\mathit{BB}(\mathit{lvar})\{S\}$ to represent an active borrowing block [68].

Redexes reduce in their respective contexts; i.e., body-redexes in M, stat-redexes in S, and expr-redexes in E. Redexes are defined thus:

$$\begin{aligned}
\mathit{body\text{-}redexes} &::= \texttt{return}\ v\\
\mathit{stat\text{-}redexes} &::= \mathit{lval} := v \mid \texttt{borrow}\ \mathit{lval}\ \texttt{as}\ T\ x\{\ s\ \}\\
&\mid\ \ \mathit{BB}(\mathit{lvar})\{\overline{T}\ \overline{x}\ \overline{v}; \texttt{skip}\}\\
\mathit{expr\text{-}redexes} &::= x \mid f \mid v.\texttt{get} \mid v.m!(\overline{v}) \mid \texttt{new}\ C() \mid \mathit{lval}\text{--}
\end{aligned}$$

Good Statement

$$\textsc{(Lval-Assign)}\quad \dfrac{\mathit{lval} : o :: C \in \Gamma \quad \Gamma; P \vdash e : o :: C \quad \mathit{writeok}(P, \mathit{lval})}{\Gamma; P \vdash \mathit{lval} := e \ \mathsf{ok}}$$

$$\textsc{(Stat-Sequence)}\quad \dfrac{\Gamma; P \vdash s \ \mathsf{ok} \qquad \Gamma; P \vdash s' \ \mathsf{ok}}{\Gamma; P \vdash s; s' \ \mathsf{ok}}$$

$$\textsc{(Stat-Borrow)}\quad \dfrac{\Gamma \vdash \mathit{lval} : \texttt{unique} :: C \ \texttt{exact} \qquad \Gamma, a : *, x : a :: C \vdash s \ \mathsf{ok} \qquad \mathit{writeok}(P, \mathit{lval})}{\Gamma; P \vdash \texttt{borrow} \ \mathit{lval} \ \texttt{as} \ a :: C \ x \ \{s\} \ \mathsf{ok}}$$

Good Expression

$$\textsc{(Expr-Exact)}\quad \dfrac{\mathit{lval} : T \in \Gamma}{\Gamma \vdash \mathit{lval} : T \ \texttt{exact}}$$

$$\textsc{(Expr-Lval)}\quad \dfrac{\Gamma \vdash \mathit{lval} : o :: C \ \texttt{exact} \qquad o \neq \texttt{unique} \quad \mathit{readok}(P, \mathit{lval})}{\Gamma; P \vdash \mathit{lval} : o :: C}$$

$$\textsc{(Expr-Dread)}\quad \dfrac{\Gamma \vdash \mathit{lval} : \texttt{unique} :: C \ \texttt{exact} \qquad \mathit{writeok}(P, \mathit{lval})}{\Gamma; P \vdash \mathit{lval}\texttt{--} : \texttt{unique} :: C}$$

$$\textsc{(Expr-Get)}\quad \dfrac{\Gamma; P \vdash e : !T \qquad \neg\texttt{unique}(T)}{\Gamma; P \vdash e.\texttt{get} : T}$$

$$\textsc{(Expr-Unique-Get)}\quad \dfrac{\Gamma; P \vdash e : !T \quad \texttt{unique}(T)}{\Gamma; P \vdash e.\texttt{uget} : T}$$

$$\textsc{(Expr-Call-Async)}\quad \dfrac{\Gamma; P \vdash e :: \texttt{active} : C \qquad \mathit{mtype}(m, C) = (\epsilon, P', \overline{T} \rightarrow T) \qquad \texttt{this} \in \mathit{owners}(\overline{T}) \Rightarrow e = \texttt{this} \qquad \Gamma; P \vdash \overline{e} : \overline{T}}{\Gamma; P \vdash e!m\,(\overline{e}) \ : !T}$$

$$\textsc{(Expr-Call-Sync)}\quad \dfrac{\begin{array}{c}\Gamma; P \vdash e : o :: C \quad o \neq \texttt{active} \\ \mathit{mtype}(m, C) = (\overline{a}, P', \overline{T} \rightarrow T) \quad |\overline{a}| = |\overline{o}| \\ \texttt{this} \in \mathit{owners}(\overline{T}) \Rightarrow e = \texttt{this} \quad \sigma = [o/\texttt{owner}, \overline{o}/\overline{a}] \\ \overline{o} \subseteq \{a,\texttt{this},\texttt{owner}\} \quad \Gamma; P \vdash \overline{e} : \overline{T\sigma} \quad \mathit{callok}(P, P', o)\end{array}}{\Gamma; P \vdash e.\texttt{<}\overline{o}\texttt{>}m\,(\overline{e}) \ : T\sigma}$$

$$\textsc{(Expr-New)}\quad \dfrac{\Gamma; P \vdash \diamond \quad \neg\mathit{active}(C)}{\Gamma; P \vdash \texttt{new} \ C\texttt{()} \ : \texttt{unique} :: C}$$

$$\textsc{(Expr-New-Active)}\quad \dfrac{\Gamma; P \vdash \diamond \quad \mathit{active}(C)}{\Gamma; P \vdash \texttt{new} \ C\texttt{()} \ : \texttt{active} :: C}$$

$$\textsc{(Expr-Null)}\quad \dfrac{\Gamma \vdash T}{\Gamma; P \vdash \texttt{null} : T}$$

$$\textsc{(Expr-Clone)}\quad \dfrac{\Gamma; P \vdash e : o :: C \quad o \neq \texttt{active}}{\Gamma; P \vdash \texttt{clone} \ e : \texttt{unique} :: C}$$

$$\textsc{(Expr-Clone-Active)}\quad \dfrac{\Gamma; P \vdash e : \texttt{active} :: C}{\Gamma; P \vdash \texttt{clone} \ e : \texttt{active} :: C}$$

$$\textsc{(Expr-Subsumption)}\quad \dfrac{\Gamma; P \vdash e : T \qquad \Gamma; P \vdash T \preceq T'}{\Gamma; P \vdash e : T'}$$

Figure 9. Typing statements and expressions. $\mathit{owners}(\overline{T})$ gives the set of owners appearing in some type in $\overline{T}$.

$$
\begin{array}{rcl}
\mathit{config} & ::= & \epsilon \mid \mathit{object} \mid \mathit{future} \mid \mathit{obind} \mid \mathit{config}\ \mathit{config} \\
\mathit{object} & ::= & (\mathit{oid}, \alpha, \tau, \mathit{fds}, \mathit{processes}) \\
\alpha & ::= & \texttt{active} \mid \texttt{passive} \\
\tau & ::= & \mathit{ovar} :: C \\
\mathit{fds} & ::= & \overline{\kappa}\ \overline{f}\ \overline{v} \\
v & ::= & \mathit{oid} \mid \mathit{mid} \mid \texttt{null} \mid b \\
\kappa & ::= & \texttt{v} \mid \texttt{r} \mid \texttt{w} \mid \texttt{u} \mid \texttt{i} \\
\mathit{future} & ::= & (\mathit{mid}, \mathit{mc}, \mathit{mode}, v) \\
\mathit{mc} & ::= & \mathit{oid}.m(\overline{v}) \\
\mathit{process} & ::= & (P, \mathit{flag}, \mathit{sr} : T) \\
\mathit{processes} & ::= & \epsilon \mid \mathit{process} \mid \mathit{processes}\ \mathit{processes} \\
\mathit{flag} & ::= & \texttt{run} \mid \texttt{sleep} \\
\mathit{obind} & ::= & (\mathit{ovar}, \mathit{ovar})
\end{array}
$$

Figure 10. The syntax for run-time configurations. Here, *oid* and *mid* denote identifiers for objects and futures, respectively. Processes correspond to method invocations, and include P to indicate whether the method is `safe`, `read`, or `write`, and a *flag* to indicate whether or not it is running or sleeping. κ denotes an annotation on fields used to record whether a final field is uninitialised (`u`) or initialised (`i`), and whether a non-final field is virginal (`v`), has been only read (`r`), or has been at least written (`w`). *obind* records a mapping from owner placeholders to actual owners—a level of indirection which enables transfer of ownership without requiring a global substitution of owners.

Filling the hole of a context Q with an expression r is denoted $Q[r]$.

Recall that l-values ranges over fields and variables. Fields are in component *fds* of a configuration, whereas variables form part of *sr*. To find the value of x when in some context Q, the innermost x must be selected from the stack of the selected process in Q. This is relatively simple to define. We abstract the operations of looking up and assign *lval*s as the following operations, defined over a tuple of the fields *fds* and a context Q. We assume that these operations update the mode, κ, for any field accesses and updates:

- $(v, \mathit{fds}') = \textsc{access}(\mathit{fds}, Q)(\mathit{lval})$: v is the value stored in l-value *lval*. If *lval* is a field, then its mode is updated to indicate that a read has occurred. This function fails if a final field is uninitialised with an error we are not aiming to trap. The resulting fields fds' reflect the updated mode.
- $(\mathit{fds}', Q') = \textsc{update}(\mathit{fds}, Q)(\mathit{lval}, v)$: updates the l-value *lval* with new value v, giving a new set of fields or new context. This function also updates the mode to indicate that a write has occurred, in the case that *lval* is a field, and it marks uninitialised final fields as initialised and fails when a *lval* is an already initialised final field.

Next we define a function Oh-Lookup to find the value associated with a syntactic owner (`this`, `owner` or an owner parameter, though not `unique`, `safe`, `read`): $\textsc{Oh-Lookup}(o, \mathit{process}) = \mathit{ovar}$, whenever *ovar* is the owner associated with o in *proccess*.

A function collects the identifiers defined in a configuration:

$$\begin{aligned}
\mathrm{dom}(\epsilon) &= \emptyset\\
\mathrm{dom}(\mathit{config}\ \mathit{config}') &= \mathrm{dom}(\mathit{config}) \cup \mathrm{dom}(\mathit{config}')\\
\mathrm{dom}((\mathit{oid}, \_, \_, \_)) &= \{\mathit{oid}\}\\
\mathrm{dom}((\mathit{mid}, \_, \_, \_)) &= \{\mathit{mid}\}\\
\mathrm{dom}((\mathit{ovar}, \_)) &= \{\mathit{ovar}\}
\end{aligned}$$

(Red-Active-Call)

$$\frac{\mathit{mid}\text{ is fresh}}{(\mathit{oid}, \alpha, \tau, \mathit{fds}, Q[\mathit{oid}'!m(\overline{v})]) \to (\mathit{oid}, \alpha, \tau, \mathit{fds}, Q[\mathit{mid}])\ (\mathit{mid}, \mathit{oid}'.m(\overline{v}), \mathtt{s}, \mathtt{null})}$$

(Red-Get)

$$(\mathit{oid}, \alpha, \tau, \mathit{fds}, Q[\mathit{mid}.\mathtt{get}])\ (\mathit{mid}, \mathit{mc}, \mathtt{c}, v) \to (\mathit{oid}, \alpha, \tau, \mathit{fds}, Q[v])\ (\mathit{mid}, \mathit{mc}, \mathtt{c}, v)$$

(Red-Unique-Get)

$$(\mathit{oid}, \alpha, \tau, \mathit{fds}, Q[\mathit{mid}.\mathtt{uget}])\ (\mathit{mid}, \mathit{mc}, \mathtt{c}, v) \to (\mathit{oid}, \alpha, \tau, \mathit{fds}, Q[v])\ (\mathit{mid}, \mathit{mc}, \mathtt{c}, \mathtt{null})$$

(Red-New)

$$\frac{\begin{array}{c}\mathit{oid}'\text{ and }\mathit{ovar}'\text{ are fresh}\quad \mathit{fds}' = \mathit{defaults}(C)\\ \mathit{active}(C) \Rightarrow \alpha = \mathtt{active}\quad \neg\mathit{active}(C) \Rightarrow \alpha = \mathtt{passive}\end{array}}{\begin{array}{c}(\mathit{oid}, \alpha, \tau, \mathit{fds}, Q[\mathtt{new}\ C()]) \to\\ (\mathit{oid}, \alpha, \tau, \mathit{pq}, \mathit{fds}, Q[\mathit{oid}'])\ (\mathit{oid}', \alpha', \mathit{ovar}' :: C', \mathit{fds}', \epsilon)\ (\mathit{ovar}'\ \ \mathit{ovar}')\end{array}}$$

(Red-Bind)

$$\frac{\begin{array}{c}\tau = \_ :: C\quad \mathit{mbody}(m, C) = (P, \overline{T}\ \overline{x}, \overline{U}\ \overline{y}, \mathit{sr} : T)\\ p = (P, \mathtt{sleep}, \overline{T}\ \overline{x}\ \overline{v}; \overline{U}\ \overline{y}\ \mathit{default}(\overline{U}); \mathtt{fut}(T)\ \mathtt{destiny}\ \mathit{mid}; \mathit{sr} : T)\end{array}}{(\mathit{oid}, \alpha, \tau, \mathit{fds}, \mathit{pq})\ (\mathit{mid}, \mathit{oid}.m(\overline{v}), \mathtt{s}, \mathtt{null}) \to (\mathit{oid}, \alpha, \tau, \mathit{fds}, \mathit{pq} :: p)\ (\mathit{mid}, \mathit{oid}.m(\overline{v}), \mathtt{a}, \mathtt{null})}$$

(Red-Schedule)

$$\frac{\alpha = \mathtt{active} \Rightarrow \{(P, \mathit{flag}, \mathit{sr}) \in \mathit{pq} \mid P = \mathtt{write} \wedge \mathit{flag} = \mathtt{run}\} = \emptyset}{(\mathit{oid}, \alpha, \tau, \mathit{fds}, (P, \mathtt{sleep}, \mathit{sr}) :: \mathit{pq}) \to (\mathit{oid}, \alpha, \tau, \mathit{fds}, (P, \mathtt{run}, \mathit{sr}) :: \mathit{pq})}$$

(Red-Return)

$$\frac{l(\mathtt{destiny}) = \mathit{mid}}{\begin{array}{c}(\mathit{oid}, \alpha, \tau, \mathit{fds}, (P, \mathtt{run}, M[\mathtt{return}\ v : T]) :: \mathit{pq}))\ (\mathit{mid}, \mathit{oid}.m(\overline{v}), \mathtt{a}, \mathtt{null}) \to\\ (\mathit{oid}, \alpha, \tau, \mathit{fds}, \mathit{pq})\ (\mathit{mid}, \mathit{oid}.m(\overline{v}), \mathtt{c}, v)\end{array}}$$

(Red-Context)

$$\frac{\mathit{config} \to \mathit{config}'\quad \mathrm{dom}(\mathit{config}') \cap \mathrm{dom}(\mathit{config}'') = \emptyset}{\mathit{config}\ \mathit{config}'' \to \mathit{config}'\ \ \mathit{config}''}$$

(Red-Parallel)

$$\frac{\begin{array}{c}\mathit{config}_1\ \ \mathit{config}_s \to \mathit{config}_1'\ \ \mathit{config}_s^1\quad \mathit{config}_2\ \ \mathit{config}_s \to \mathit{config}_2\ \ \mathit{config}_s^2\quad \mathit{config}_s\text{ is virginal}\\ \mathrm{dom}(\mathit{config}_s) = \mathrm{dom}(\mathit{config}_s^1) = \mathrm{dom}(\mathit{config}_s^2)\quad \mathrm{dom}(\mathit{config}_1') \cap \mathrm{dom}(\mathit{config}_2') = \emptyset\end{array}}{\mathit{config}_1\ \ \mathit{config}_2\ \ \mathit{config}_s \to \mathit{config}_1'\ \ \mathit{config}_2'\ (\mathit{config}_s^1 \odot \mathit{config}_s^2)}$$

Figure 11. The context reduction semantics I.

(Red-Lval)
$$\frac{(v, \mathit{fds}') = \textsc{access}(\mathit{fds}, Q)(\mathit{lval})}{(\mathit{oid}, \alpha, \tau, \mathit{fds}, Q[\mathit{lval}]) \rightarrow (\mathit{oid}, \alpha, \tau, \mathit{fds}', Q[v])}$$

(Red-Lvar-Dread)
$$\frac{(v, \mathit{fds}') = \textsc{access}(\mathit{fds}, Q)(\mathit{lval}) \quad (\mathit{fds}'', Q') = \textsc{update}(\mathit{fds}', Q)(\mathit{lval}, v)}{(\mathit{oid}, \alpha, \tau, \mathit{fds}, Q[\mathit{lval}{-}{-}]) \rightarrow (\mathit{oid}, \alpha, \tau, \mathit{fds}'', Q'[v])}$$

(Red-Lval-Assign)
$$\frac{(\mathit{fds}', Q') = \textsc{update}(\mathit{fds}, Q)(\mathit{lval}, v)}{(\mathit{oid}, \alpha, \tau, \mathit{fds}, Q[\mathit{lval} := v]) \rightarrow (\mathit{oid}, \alpha, \tau, \mathit{fds}', Q'[\texttt{skip}])}$$

(Red-Owner-Cast)
$$\frac{\textsc{Oh-Lookup}(o, (\mathit{oid}, \alpha, \tau, \mathit{fds}, Q[(o)\,\mathit{oid}'])) = \mathit{ovar}''}{\begin{array}{c}(\mathit{oid}, \alpha, \tau, \mathit{fds}, Q[(o)\,\mathit{oid}'])\ (\mathit{ovar}''\ \ \mathit{ovar}''')\ (\mathit{oid}', \alpha', \mathit{ovar} :: C, \mathit{fds}', pq)\ (\mathit{ovar}\ \mathit{ovar}') \rightarrow \\ (\mathit{oid}, \alpha, \tau, \mathit{fds}', Q[\mathit{oid}]')\ (\mathit{ovar}''\ \ \mathit{ovar}''')\ (\mathit{oid}', \alpha', \mathit{ovar} :: C, \mathit{fds}', pq)\ (\mathit{ovar}\ \mathit{ovar}''')\end{array}}$$

(Red-Borrowing-Start)
$$\frac{\textsc{access}(\mathit{fds}, Q)(\mathit{lval}) = (v, \mathit{fds}')}{(\mathit{oid}, \alpha, \tau, \mathit{fds}, Q[\texttt{borrow}\ \mathit{lval}\ \texttt{as}\ T\ x\{\ s\ \}]) \rightarrow (\mathit{oid}, \alpha, \tau, \mathit{fds}', Q[BB(\mathit{lvar})\{T\ x\ v; s\}])}$$

(Red-Borrowing-Finish)
$$\frac{(\mathit{fds}', Q') = \textsc{update}(\mathit{fds}, Q)(\mathit{lval}, v)}{(\mathit{oid}, \alpha, \tau, \mathit{fds}, Q[BB(\mathit{lvar})\{\overline{T}\ \overline{x}\ \overline{v}; \texttt{skip}\}]) \rightarrow (\mathit{oid}, \alpha, \tau, \mathit{fds}', Q'[\texttt{skip}])}$$

Figure 12. The context reduction semantics II. (Red-Owner-Cast) is a coercion inserted automatically when owners change via subsumption [22].

Expressions. In (Red-Call), an asynchronous call adds a sleeping future to the configuration, returning the future's id to the caller—this provides a unique identifier for the invocation. In (Red-Get), a read operation on a future variable blocks the active process until the future is in completed mode. (Red-Unique-Get) is the variant for futures containing uniques—it destructively reads the future. Object creation in (Red-New) introduces a new instance of a class into the configuration, with default values for the new object's fields.

Method invocation and return. A method call results in an activation on the callee's process queue. As the call is asynchronous, there is a delay between the call and its activation, represented by the sleeping mode of a future. Subsequent to the call, (Red-Bind) creates a process to run the method. This process is added to the process queue, and the future changes its mode to active, thus preventing multiple activations of the method. (Red-Schedule) is the rule to select and run a method. It allows only one `write` call to be active in an active object. When process execution is completed, the return value is stored by (Red-Return) in the future identified by the `destiny` variable. This future changes its mode to completed, and the process is discarded.

Note that we assume synchronous calls have been encoded using an asynchronous call followed by a `get`.

Context and parallel reductions. A reduction applies to a subconfiguration by rule (Red-Context). Rule (Red-Parallel) allows concurrent reductions on potentially overlapping configurations. These overlapping configurations are then merged after the reduction has occurred, indicated by the operation $\odot$. This operation fails in the case that a write-write or read-write conflict occurs in the two concurrent reductions (or an error occurs with final initialisation). Merging also collects together futures that may be shared, relying on the fact that at most reduction could have updated it. We define $\mathit{config} \odot \mathit{config}'$ over two configurations that have the same domain. It works 'pointwise', that is, for pairs of elements with the same *id*, so we simply need to define how it operates for objects and futures:

objects

$$(\mathit{oid}, \alpha, \tau, \mathit{fds}, \mathit{pq}) \odot (\mathit{oid}, \alpha, \tau, \mathit{fds}', \mathit{pq}') = (\mathit{oid}, \alpha, \tau, \mathit{fds} \odot \mathit{fds}', \mathit{pq} \odot \mathit{pq}')$$

where $\mathit{pq} \odot \mathit{pq}'$ is the union of the two process queues provided that the `destiny` variables (containing a supposedly unique future id for each invocation) have distinct values, and that only one has an active process in the case of active objects.

Now we define $(\kappa\ f\ v) \odot (\kappa'\ f\ v') = (\kappa''\ f\ v'')$. Note that if the field f is not written by either branch, then we have that $\kappa, \kappa' \in \{\mathtt{u}, \mathtt{v}, \mathtt{r}\}$ and that $v = v'$ (as they both started from the same configuration).

- If $\kappa = \mathtt{i}$, then require $\kappa' = \mathtt{u}$ or $\kappa' = \mathtt{i}$ and $v = v'$. In that case, the resulting $\kappa'' = \mathtt{i}$ and $v'' = v$. And vice versa. Otherwise, the entire result is a final variable fault—which we do not care about.
- if $(\kappa, \kappa') \in \{(\mathtt{w}, \mathtt{w}), (\mathtt{r}, \mathtt{w}), (\mathtt{w}, \mathtt{r})\}$, then a *data race* is reported. This is the error we do care about.
- in all other cases, either one of κ and κ' is $\mathtt{w}$, in which case the corresponding v or v' is chosen as v'', *or*, $v = v'$ and hence define $v'' = v$.

futures Assuming that $\mathtt{s} < \mathtt{a} < \mathtt{c}$, $(\mathit{mid}, \mathit{mc}, \mathit{mode}, v) \odot (\mathit{mid}, \mathit{mc}, \mathit{mode}', v') = (\mathit{mid}, \mathit{mc}, \mathit{mode}'', v'')$, where $\mathit{mode}'' = \max(\mathit{mode}, \mathit{mode}')$, $\neg(\mathit{mode} = \mathit{mode}' = \mathtt{c})$, and $\mathtt{v}''$ is v if $v \neq \mathtt{null}$ and v' otherwise.

6.2 Meta-Theory Overview

The system presented here extends the formalisation of ownership types and external uniqueness presented in Wrigstad's thesis [68] in several ways. Firstly, object interaction is via asynchronous method calls and futures. Secondly, various mechanisms are present to limit which fields can be read and/or written in a method. Neither of these affect the soundness of the underlying class types nor the ownership and external uniqueness part of our system.

Our dynamic semantics is designed so that data races are detected exclusively in the rule (Red-Parallel), but the challenge comes in showing that this rule does not fail due to data races. The absence of data races comes down the the fact that the type system combined with method invocation scheduling (Red-Schedule) ensure either that:

- at most one `write` method can be running in an active object at a time; *and*

- due to the invariant imposed by ownership types—namely that only the normal references to a passive object are held by the owning active object or by other passive objects owned by the active object—we obtain that at most one `write` method can be running in a passive object at a time; *and*
- any number of `safe` methods can also be running at the same time.

OR, in the case that a unique reference to a passive object is converted into a `immutable` reference, and hence no normal reference to the object exists,

- any number of `read` methods can be running in a passive object; *and*
- any number of `safe` methods can be running at the same time.

It is relatively easy to show that `read` methods only read fields (of their target object) and that `safe` methods only read the final fields of their target object (and no fields apart from those in the target objects are accessible).

Together, these combine to show that no write-write or write-read conflicts occur within both active and passive objects.

7 Related Work

7.1 Ownership Types

This paper is not the first to apply ownership types to concurrency control. Some approaches introduce thread-local owners to encapsulate data within threads [6, 10], thus avoiding data races and, in the latter case, deadlocks. Guava [6] is presented as an informal collection of rules, instead of a type system, which would require a significantly more complex ownership types system that the one we present here. Boyapati et al.'s PRFJ [10] encodes a lock ordering in class headers, which imposes a rigid structure on the code which makes program evolution tedious. This system is more complex than ours, and forces programmers to handle threads explicitly. Another related approach uses Universes instead of ownership types for race safety [24]. In each case the underlying concurrency model is threads and not active objects.

More recently, X10 incorporates *place types* to statically describe where data resides to improve data locality [61]. Place types have complex structure. X10 also has futures and a shared space of immutable data. Remote data is remotely accessed, and no unique references are used to enable efficient data transfer between places. X10 has a hierarchical memory model, similar to a distributed system, so pointer transfer need not have the same advantages as in a shared memory system. X10 uses conditional atomic blocks (on local data), similar to software transactional memory. X10 is designed for high performance computing, whereas we aim for more general purpose programming.

StreamFlex [64] is a proposal which is close in spirit to ours. It uses a minimal notion of ownership, with little need for annotations, to handle garbage collection issues in a real-time setting. Objects may be passed between concurrent components without copying (as does the Singularity OS [37]). StreamFlex's notion of immutability is however more limited safe or unique references are not supported. In conclusion, the additional limitations of the stream programming approach (compared to our more general-purpose approach) allows StreamFlex to use even less syntactic baggage than Joëlle. Previous experiments (e.g., [21]) suggest that the massive syntactic overhead of full-blown ownership types systems can be very much lowered when adapted to more a specific programming model. We believe that Joëllehas a very elegant trade-off between expressiveness, generality, and syntactic overhead.

7.2 Actors and Active objects

In the actor model [1], concurrent threads (called *actors*) employ asynchronous message passing as their only means for communication. Thus, message passing does not transfer control between objects. Modern actor models/libraries such as those developed for Java and Scala [33,67], and Erlang's actor model [5], include features such as access to the sender of a message and message forwarding.

Erlang is relevant as a purely functional language with immutable data. This is a bad fit for OO and encoding of data structures that rely on sharing or object identifiers is difficult or impossible. In a proposal related to ours, Carlsson et al. [15] present an underapproximate *message analysis*, reminiscent of escape analysis, to detect when messages can be safely shared rather than copied for a mini-Erlang system. Their analysis has cubic worst-case time complexity. Our type-system based approach should fare better for the price of a few additional concepts.

Active objects interacting through *asynchronous method calls* have been proposed to better combine object-orientation with concurrent and distributed programming (e.g., [17,40,48]), due to looser coupling between callers and callees than with synchronous calls. Asynchronous calls introduce *wait-by-necessity* synchronisation: execution proceeds until the method return, encapsulated as a *future* [8,17,30,34,45,70], is accessed. In contrast to synchronous calls, no control is transferred with the call; asynchronous calls may be seen as triggers of concurrent activity [54]. Just like

Other active object languages such as the Emerald system [38,60] do not use asynchronous method calls. Emerald objects can be active or passive, but method calls are synchronous, so > 1 thread in a single object is possible. Objects have a monitored section which guarantees mutual exclusion and processes synchronise through system-defined condition objects. To simplify sharing, Emerald supports immutable objects through unchecked programmer annotation.

Active objects are implemented as a main concurrency model of Symbian OS [50] and in the ProActive Java framework [7]. Based on Eiffel, Eiffel// [16] is an active objects system with asynchronous method calls with futures, and SCOOP [48] uses preconditions for task scheduling. In SCOOP, active object boundaries are captured by `separate` annotations on variables, which are not statically enforced. (This is partially improved by Nienaltowski [53].) In the original SCOOP proposal, method arguments across active objects have deep copying semantics. Object migration through uniqueness is not supported. Later versions of SCOOP [53] integrate an eager locking mechanism to enable pass-by-reference arguments for non-value objects. An integration of our approach with SCOOP seems fairly straightforward.

The concurrency model of Creol [40] inspired the formalisation of our dynamic semantics. In Creol only one method may execute in an active object at any given time, similar to ASP [17]; however, the active object may yield its thread of control at specified *release* points in the code, so the active object may accept external method calls. Thus, an active object may have multiple pending method activations, some of which are midway through execution. These execute in some interleaved way, which allows different activities to be pursued within the object; in particular, active and reactive object behaviour are easily and dynamically combined [40].

Different components or active objects communicate data by cloning in, e.g., the coordination language ToolBus [26] and ASP [17]. In a distributed setting this is vindicated, but, as ToolBus developers [26] observe, copying data is a major source of performance problems. This is exactly the problem our approach aims to address, without introducing data-races, in a light-weight, statically checkable fashion.

7.3 Software Transactional Memory

Software Transactional Memory is a recent approach to avoiding data races [62]. Critical sections are protected by atomic blocks, which are executed in an optimistic, transactional way, without locking. Changes are logged and eventually committed atomically. If a conflict occurs, the atomic block is reversed and restarted. Software transactional memory is not alien to active objects. For example, one could imagine Software transactional memory under the hood of Emerald's mutually exclusive object regions [38, 60].

We view software transactional memory as a complementary approach. If our type system were introduced in a context with software transactional memory, we could statically infer methods for which transactions would not need to be monitored (safe and immutable methods) and could exploit uniqueness to optimise the software transactional memory implementation. In turn, the transactional memory could remove the need for programmer-inserted release points (which, however, are orthogonal to our proposal).

Ongoing work by Kotselidis et al. [42] adds software transactional memory to ProActive. Preliminary results are promising, but the system retains ProActive's deep-copying semantics even for inter-node computations.

8 Concluding Remarks

In this paper, we applied ownership types to a data sharing problem of active objects. Our solution involves a combination of ownership, external uniqueness, and immutability. Our formalisation is close to ASP [17], and to its framework realisation in Java, ProActive [7]. Our minimal ownership system was defined so that default annotations can be chosen to give a low syntactic overhead. Specifically, we expect that few changes would be necessary to code for passive objects if our system were implemented in ProActive.

Our system is close in spirit to several existing systems that lack our static checking for things like immutability. No existing active objects system is powerful enough to infer minimal safe cloning the way we outlined in Section 4.3. We believe that our system should be possible to incorporate in many existing systems. Potentially, Joëlle could even be compiled to a version of Proactive that avoids copying when safe to do so. Furthermore, we believe that our system captures existing programming practices (close inspection of Lea's concurrency book [44] supports this) with low syntactic overhead *in a statically checkable fashion.*

References

1. G. Agha and C. Hewitt. Actors: A conceptual foundation for concurrent object-oriented programming. In *Research Directions in Object-Oriented Programming*, pages 49–74. MIT Press, 1987.
2. J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *ECOOP*, LNCS 3086, pages 1–25. Springer, 2004.
3. J. Aldrich, V. Kostadinov, and C. Chambers. Alias Annotations for Program Understanding. In *OOPSLA*, 2002.
4. P. S. Almeida. Balloon Types: Controlling sharing of state in data types. In *ECOOP*, LNCS 1241. Springer, 1997.
5. J. Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, 2007.
6. D. F. Bacon, R. E. Strom, and A. Tarafdar. Guava: a dialect of Java without data races. In *OOPSLA*, 2000.
7. L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Composing, Deploying, for the Grid. Springer, 2006.
8. H. G. Baker Jr. and C. Hewitt. The incremental garbage collection of processes. In *Proc. Symposium on Artificial Intelligence Programming Languages*, ACM SIGPLAN Notices 12, 1977.
9. A. Birka and M. D. Ernst. A practical type system and language for reference immutability. In *OOPSLA*, 2004.
10. C. Boyapati, R. Lee, and M. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *OOPSLA*, 2002.
11. C. Boyapati, B. Liskov, and L. Shrira. Ownership Types for Object Encapsulation. In *POPL*, 2003.
12. C. Boyapati and M. Rinard. A Parameterized Type System for Race-Free Java Programs. In *OOPSLA*, 2001.
13. J. Boyland. Alias burying: Unique variables without destructive reads. *Software—Practice and Experience*, 31(6):533–553, 2001.
14. J. Boyland. Why we should not add readonly to Java (yet). *Journal of Object Technology*, 5(5):5–29, 2006.
15. R. Carlsson, K. F. Sagonas, and J. Wilhelmsson. Message analysis for concurrent programs using message passing. *ACM TOPLAS*, 28(4):715–746, 2006.

16. D. Caromel. Service, Asynchrony, and Wait-By-Necessity. *Journal of Object Oriented Programming*, pages 12–22, 1989.
17. D. Caromel and L. Henrio. *A Theory of Distributed Objects*. Springer, 2005.
18. D. Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, Sydney, 2001.
19. D. Clarke and S. Drossopolou. Ownership, Encapsulation and the Disjointness of Type and Effect. In *OOPSLA*, 2002.
20. D. Clarke, S. Drossopoulou, J. Noble, and T. Wrigstad. Tribe: A simple virtual class calculus. In *AOSD*. ACM, 2007.
21. D. Clarke, M. Richmond, and J. Noble. Saving the world from bad beans: Deployment-time confinement checking. In *OOPSLA*, 2003.
22. D. Clarke and T. Wrigstad. External uniqueness is unique enough. In *ECOOP*, LNCS 2473. Springer, 2003.
23. D. G. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64, 1998.
24. D. Cunningham, S. Drossopoulou, and S. Eisenbach. Universe Types for Race Safety. In *VAMP 07*, 2007.
25. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In *ESOP*, LNCS 4421. Springer, 2007.
26. H. de Jong. *Flexible Heterogeneous Software Systems*. PhD thesis, University of Amsterdam, 2007.
27. S. Drossopoulou, D. Clarke, and J. Noble. Types for hierarchic shapes. In *ESOP*, LNCS 3924. Springer, 2006.
28. M. Fahndrich and S. Xia. Establishing object invariants with delayed types. *SIGPLAN Not.*, 42(10):337–350, 2007.
29. M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.
30. C. Flanagan and M. Felleisen. The semantics of future and an application. *J. Funct. Program.*, 9(1):1–31, 1999.
31. M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, LNCS 1523. Springer, 1999.
32. D. Grossman, M. Hicks, T. Jim, , and G. Morrisett. Cyclone: A type-safe dialect of C. *C/C++ Users Journal*, 23(1), 2005.
33. P. Haller and M. Odersky. Actors that unify threads and events. In *COORDINATION*, LNCS 4467. Springer, 2007.
34. R. H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM TOPLAS*, 7(4):501–538, 1985.
35. J. Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA*, 1991.
36. J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. The Geneva Convention on the treatment of object aliasing. *OOPS Messenger*, 3(2):11–16, 1992.
37. G. Hunt and J. Larus. Singularity: Rethinking the software stack. *Operating Systems Review*, 40(2):37–49, 2007.
38. N. C. Hutchinson, R. K. Raj, A. P. Black, H. M. Levy, and E. Jul. The Emerald programming language report. Technical Report 87-10-07, Seattle, 1987. Revised 1997.
39. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
40. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, 2007.

41. S. P. Jones and J. H. (editors). Haskell 98: A non-strict, purely functional language. Technical report, 1999.
42. C. Kotselidis, M. Ansari, K. Jarvis, M. Luján, C. Kirkham, and I. Watson. Investigating software transactional memory on clusters. In *IWJPDC '08: 10th International Workshop on Java and Components for Parallelism, Distribution and Concurrency*. IEEE, 2008.
43. R. G. Lavender and D. C. Schmidt. Active object: an object behavioral pattern for concurrent programming. *Proc. Pattern Languages of Programs*, 1995.
44. D. Lea. *Concurrent Programming in Java*. Addison-Wesley, 2nd edition, 2000.
45. B. H. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In D. S. Wise, editor, *PLDI*, pages 260–267. ACM, 1988.
46. K.-K. Ma and J. S. Foster. Inferring aliasing and encapsulation properties for Java. In *OOPSLA*, 2007.
47. B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.
48. B. Meyer. Systematic concurrent object-oriented programming. *Commun. ACM*, 36(9):56–80, 1993.
49. N. H. Minsky. Towards alias-free pointers. In *ECOOP*, LNCS 1098, pages 189–209. Springer, 1996.
50. B. Morris. CActive and Friends. Symbian Developer Network, November 2007. `http://developer.symbian.com/main/downloads/papers/CActiveAndFriends/CActiveAndFriends.pdf`.
51. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, LNCS 2262. Springer, 2002.
52. P. Müller and A. Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming*, pages 131–140. Technical Report 263, Fernuniversität Hagen, 1999.
53. P. Nienaltowski. *Practical framework for contract-based concurrent object-oriented programming*. PhD thesis, ETH Zurich, 2007.
54. O. Nierstrasz. Composing active objects. In *Research directions in concurrent object-oriented programming*, pages 151–171. MIT Press, 1993.
55. J. Noble, D. Clarke, and J. Potter. Object ownership for dynamic alias protection. In *TOOLS Pacific*, 1999.
56. J. Noble, J. Vitek, and J. Potter. Flexible Alias Protection. In *ECOOP*, 1998.
57. M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL, 2004.
58. J. Östlund, T. Wrigstad, D. Clarke, and B. Åkerblom. Ownership, uniqueness and immutability. In *IWACO*, 2007.
59. A. Potanin and J. Noble. Checking ownership and confinement properties. In *Formal Techniques for Java-like Programs*, 2002.
60. R. K. Raj, E. Tempero, H. M. Levy, A. P. Black, N. C. Hutchinson, and E. Jul. Emerald: A general-purpose programming language. *Software: Practice and Experience*, 21(1):91–118, 1991.
61. V. A. Saraswat, V. Sarkar, and C. von Praun. X10: concurrent programming for modern architectures. In *Principles and Practice of Parallel Programming*, 2007.
62. N. Shavit and D. Touitou. Software transactional memory. In *PODC*, pages 204–213. ACM, 1995.
63. M. Skoglund and T. Wrigstad. Alias control with read-only references. In *Sixth Conference on Computer Science and Informatics*, Mar. 2002.

64. J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. StreamFlex: High-throughput Stream Programming in Java. In *OOPSLA*, 2007.
65. B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.
66. M. S. Tschantz and M. D. Ernst. Javari: Adding reference immutability to Java. In *OOPSLA*, pages 211–230, 2005.
67. A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *OOPSLA*, pages 439–453. ACM, 2005.
68. T. Wrigstad. *Ownership-Based Alias Management*. PhD thesis, Royal Institute of Technology, Stockholm, May 2006.
69. T. Wrigstad and D. Clarke. Existential owners for ownership types. *Journal of Object Technology*, 4(6):141–159, 2007.
70. A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. In OOPSLA'86. *SIGPLAN Notices*, 21(11):258–268, 1986.
71. Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kiezun, and M. D. Ernst. Object and reference immutability using Java generics. In *ESEC/SIGSOFT FSE*, pages 75–84. ACM, 2007.