Deny Capabilities for Safe, Fast Actors

Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, Andy McNeil
Causality Ltd., Imperial College London
{syvlan, sophia, sebastian, andy}@causality.io

Abstract
Combining the actor-model with shared memory for performance is efficient but can introduce data-races. Existing approaches to static data-race freedom are based on uniqueness and immutability, but lack flexibility and high performance implementations. Our approach, based on deny properties, allows reading, writing and traversing unique references, introduces a new form of write uniqueness, and guarantees atomic behaviours.

1. Introduction
A current trend in programming languages is to combine the actor-model [3] of concurrency with shared memory to eliminate the requirement to copy all messages between actors, as is done in languages such as Erlang [4]. This is done to improve performance, but it results in the possibility of data races.

Historically, programming languages have mostly relied on dynamic approaches to prevent data races, using explicit mechanisms, such as mutexes or semaphores, or implicit mechanisms, such as lock inference or lock-free algorithms. Ensuring data-race freedom statically [18] improves performance by doing at compile-time what must otherwise be done at run-time, and eliminates errors that can result from incorrectly implementing locking or lock-free algorithms.

We provide a type system that ensures data race freedom statically for an actor-model language while also providing a way to type actors themselves, in the mould of active objects [13], and without placing any restrictions on the structure of messages. In addition, the type system is amenable to efficient implementation, and we have implemented it for the Pony programming language.

Existing approaches to static data race freedom use reference capabilities to describe what a reference is allowed to do. In previous work, reference capabilities have been expressed as permissions [10], fractional permissions [9], uniqueness [12], immutability [28], and isolation [19] (a refinement of separate uniqueness [22], which is a refinement of external uniqueness [12]).

We have taken a different approach and use reference capabilities to describe what other aliases are denied [17] by the existence of a reference. We use a matrix of deny properties, with notions such as isolation, mutability, and immutability all being derived from these properties. What aliases to the object are allowed to do is explicit rather than implied, whereas what the reference is allowed is derived. This change in approach gives a derivation for properties previously considered intrinsic, and models a reduction in reference capabilities as a weaker guarantee.

We clarify our use of the term reference capability. Capabilities were introduced to support protection across processes [23], and have been adopted into several branches of computing since. The term object capabilities has been coined by Mark Miller [25, 26], to describe the set of operations an object is allowed to apply on some other object. Mark Miller proposes that in order to restrict this set, one should create a new object which only offers these capabilities, and which delegates to the original object. In our work, reference capabilities offer a partition of the operations into those which may read, or write the object, or pass the object on to a different actor. Moreover, our reference capabilities are transitive, e.g. a write capability to an object o grants write access to all its fields, but also to all the objects writeable from o. Pony is both an object capability and reference capability secure language.

Other approaches have combined actors with data-race freedom, such as minimal ownership for active objects [13], capabilities for uniqueness and borrowing in Scala [22], and Kilim [29]. However, various useful patterns have not been supported, e.g. traversing and modifying an isolated data structure, or updating an object and then sending it in a message while keeping read access to it. By taking a more fundamental view of reference capabilities, we were able to develop a more flexible type system that supports such pat-
terms. Moreover, we have developed a fast implementation, with performance comparable or superior to the fastest, unsafe systems.

The matrix of deny properties exposes two novel reference capability types, tag and trn (transition). A tag reference capability allows identity comparison and asynchronous method call, but does not allow reading from or writing to the reference. We type actors as tag, which allows them to be integrated into the object type system and passed in messages. A trn reference capability is a new form of uniqueness, write uniqueness, that describes objects that can only be written to through a single reference, but can be read from through many aliases.

We also extend viewpoint adaptation [16, 19] to apply to every reference capability and introduce the concept of safe to write, which, taken together, allow reading from and writing to both unique objects and unique fields. We treat the types of temporary identifiers differently from those of permanent paths, which allows us to traverse unique structures, something that is not possible using other approaches [13, 19, 22].

In our system, an alias may have a different reference capability from the initial reference. This addresses a key issue in reference capability systems, namely that sub-typing is not reflexive: an isolated type cannot be assigned to a field or local variable unless the source reference is eliminated with a technique such as destructive read or alias burying [8]. As a part of this, we introduce unaliased types, which provide static alias tracking without alias analysis.

Our reference capabilities also provide a static region system [21], requiring no additional annotation. The trn reference capability provides a new form of write region, in which a region boundary applies to write operations but not read operations. In addition, actor behaviours are guaranteed to be atomic, in the sense that an actor is guaranteed not to witness changes in state made by other actors during the execution of a behaviour, nor can a behaviour be interrupted to execute a different behaviour on the same actor, nor can a message be received by an actor while executing a behaviour.

Contributions In this work, we present:

- Deny properties as a fundamental basis for uniqueness and immutability.
- Combination with the actor paradigm.
- A new form of write uniqueness, trn.
- A reference capability, tag, that can be used to type actors.
- Viewpoint adaptation and safe-to-write semantics for reading and writing unique types.
- An alias operation in the type system to express non-reflexive sub-typing.
- Unaliased types for static alias tracking.
- Static regions, including a new form of write region.

Moreover, a native code compiler using LLVM, a runtime, and a standard library exist for the Pony programming language that uses this type system. We have used this implementation to demonstrate efficiency through a comparison to existing actor-model languages and libraries, as well as to MPI [20].

Outline We present our ideas in terms of a minimal actor-model, object-oriented language. We present reference capabilities as deny properties in sec. 2, a syntax in sec. 3, a formal type system in sec. 4, an operational semantics in sec. 5, related work in sec. 6, an implementation and benchmarks in sec. 7, and conclusions and further work in sec. 8.

2. Reference capabilities as deny properties

Rather than indicate which operations are allowed on a reference1, our reference capabilities indicate what operations are denied on other references to the same object (aliases). We distinguish what is denied to the actor that holds a reference (local aliases) from what is denied to all other actors (global aliases). Each reference capability stands for a pair of local and global deny properties. These are shown in table 1. For example, ref denies global aliases that can read from or write to the object, but it allows local aliases to both read from and write to it.

No reference capability can deny local aliases that it allows globally. Therefore, some cells in the matrix are empty. For example, there is no reference capability that denies local read and write aliases, but denies only write aliases globally.

These deny properties are used to derive the operations permitted on a reference. A reference that denies global read and write aliases is safe to both read and write, i.e. is mutable, since it guarantees that no other actor can read from or write to the object. A reference that denies only global write aliases is only safe to read, i.e. immutable, since it guarantees no other actor will write to the object, but does not guarantee no other actor will read from it. A reference that allows all global aliases is not safe to either read or write, i.e. it is opaque.

In addition, when the local deny properties and the global deny properties of a reference are the same, the reference can be safely sent as an argument to an asynchronous method call to another actor, i.e. it is sendable. In other words, when the local alias deny properties are the same as the global alias deny properties, it does not matter which actor holds the reference.

Short examples A ref reference to an object denies global read/write aliases. As a result, it is safe to mutate the object, since no other actor can read from it. This is effectively a traditional object-oriented reference type.

1 We use the term reference to mean the path currently being considered, and alias to mean any other path to the same object.
If an actor has a box reference to an object, no alias can be used by other actors to write to that object. This means that other actors may be able to read the object, and aliases in the same actor may be able to write to it (although not both: if the actor can write to the object, other actors cannot read from it). Using box for immutability allows a program to enforce read-only behaviour, similar to const in C/C++. For example:

```kotlin
class List
    fun box size1(): Int => ...
    fun val size2(): Int => ...
```

Note that the receiver reference capability is specified after the keyword fun. In `size1`, by indicating that the receiver has box reference capability, we can be certain that this will not be mutated when calculating its size (provided it has no mutable alias to itself). In addition, immutability is transitive, so no readable fields of this will be mutated either. Since box denies global write aliases but does not deny local write aliases, it is possible for this to be mutated through some alias if that alias is held by the same actor. The box reference functions as a black box: the underlying object may be mutable through an alias or it may be immutable through any alias.

In `size2`, by indicating that the receiver has val reference capability, we make a stronger guarantee: we deny both local and global write aliases. As a result, it is not possible for this (and all its readable fields) to be mutated, regardless of other aliases, nor will it be mutated at any time in the future.

Since a val reference has the same local and global deny properties, it is possible to send a val reference to another actor. A val reference is effectively a value type, similar to values in functional languages.

```kotlin
actor Dataflow
    be calculate1(list: List val) => ...
    be calculate2(list: List List box) // Not allowed
```

We use the keyword actor to indicate a class that can have behaviours (asynchronous methods), and we use the keyword be to define behaviours. A behaviour is executed asynchronously by the receiving actor, and a given actor executes only one behaviour at a time, making behaviours atomic. While executing a behaviour, the receiver sees itself (i.e. this in the behaviour) as ref, and is able to freely read from and write to its own fields. However, at the call-site, a behaviour does not read from or write to the receiver, and so a behaviour can be called on a tag receiver.

In `calculate1`, the `list` parameter is guaranteed to be deeply immutable, because a val is guaranteed to have no local or global write aliases. As a result, it is safe to share this object amongst actors. Denying global write aliases means no actor can write to the object, regardless of how many actors have an alias to list, making concurrent reads safe without copying, locks, or any other runtime safety mechanism. In `calculate2`, a parameter of type `List box` is rejected by the type system, as a box does not deny local write aliases, making it unsafe to send a box to another actor as the sending actor could retain a mutable alias.

A tag reference has no deny properties, but it can be used for asynchronous method calls, i.e. calling behaviours. A reference capability with no permissions has appeared in previous work [27], but without allowing asynchronous method calls.

```kotlin
actor Dataflow
    be step(list: List val, flow: Dataflow tag) => ...
```

Here, we can call behaviours on flow, but we cannot read or write the fields of flow. However, when flow executes those behaviours asynchronously, it will see itself as a ref, allowing it to mutate its own state. As such, tag allows us to type actors themselves, thus integrating them into our type system and allowing threads (in the form of actors) to be treated as first-class values. In contrast to existing systems [19], we formalise both dynamic thread creation (actor constructors) and communicating actor graphs of any shape (including cycles).

In order to pass mutable data between actors, we use iso references. All mutable reference capabilities deny global read/write aliases, allowing them to be written to because no other actor can read from the object. An iso reference also denies local read/write aliases, which means if the iso reference is sent to another actor, we are guaranteed that the sending actor no longer holds either read or write aliases to the object sent.

```kotlin
actor Dataflow
    be step(list: List iso, flow: Dataflow tag) => ...
```

Here, by passing an iso reference, a Dataflow actor can mutate the list before sending it to the flow actor. In order to do this, we must be certain the sending actor does not retain a read or write alias. To this end we use an aliasing type system wherein a newly created alias to an object cannot violate the deny properties of the reference being aliased. For example, a newly created alias of an iso reference must be neither readable nor writeable (i.e. a tag).
To move deny properties, we consume a reference or use a destructive read, both with the expected semantics.

```scala
class BookingManager
  var accountant: Accountant
  var all: Map[Date, Booking box]
  var future: Map[Date, Booking trn]
  fun ref close(date: Date) =>
    accountant.account(future.remove(date))
```

Our type system introduces the concept of unaliased types, annotated with $\circ$, in order to type values for which an alias has been removed. Here, the consume produces a List iso which is aliased as a List iso when the behaviour is called. The non-destructive read produces a List iso which is aliased as a List tag, which is rejected by the type system.

We distinguish between aliases which outline the execution of an expression, and temporary identifiers which do not. The use of temporary identifiers, combined with viewpoint adaptation, allows reading from and writing to isolated objects and isolated fields. Earlier work on isolation and external uniqueness systems [12, 19, 22] does not provide this.

```scala
class Dataflow
  be step(list: List iso, flow: Dataflow tag) =>
    flow.step(list, this) // Not allowed
    flow.step(consume list, this)
```

Here, we mutate list1 by assigning list2 to its next field, maintaining isolation for both list1 and list1.next. Similarly, we could read from or write to fields of list1.next, since path traversal is allowed. This also allows calling methods on isolated references and fields of any path depth.

Unsafe reads are prevented by viewpoint adaptation, and unsafe writes are prevented by safe-to-write rules. For example:

```scala
class Dataflow
  fun ref append(list: List iso, list2: List iso, flow: Dataflow tag) =>
    list1.next = consume list2
    flow.step(consume list1)
```

Even if list1.next had the type List iso, this assignment is rejected. As a result, isolated references form static regions, wherein mutable references reachable by the iso reference can only be reached via the iso reference and immutable references reachable by the iso reference are either globally immutable or can only be reached via the iso reference.

A trn reference makes a novel guarantee: write uniqueness without read uniqueness. By denying global read/write aliases, but only denying local write aliases, it allows an object to be written to only via the trn reference, but read from via other aliases held by the same actor. This allows the object to be mutable while still allowing it to transition to an immutable reference capability in the future, in order to share it with another actor.

```scala
class Accountant
  be account (booking: Booking val) => ...
```

Here we use a trn reference to model bookings that remain mutable until they are closed and sent for accounting. All bookings are in the all map, but only mappings that have not been closed out and are still mutable are in the future map. When a booking is closed, it is removed from the future map, returning a Booking trn, which is aliased as a Booking trn, which is a subtype of Booking val and can be shared with the Accountant actor. Without a write unique type, namely trn, this would require copying the Booking.

A trn reference also forms a static region, but with a looser guarantee than an iso reference. Mutable references reachable by the trn reference can only be reached via the trn reference, but immutable references, whether global or local, are not contained in the resulting write region.

### 3. Syntax

In fig. 1 we present the syntax, which is a subset of Pony. We support actors in the mould of active objects, introduced with the keyword actor. These can have both synchronous methods (functions, introduced through the keyword fun) and asynchronous methods (behaviours, introduced through the keyword be) as well as named constructors (introduced through the keyword new). Passive objects (introduced through the keyword class) have only synchron-
ous methods (functions) and constructors. We use the term method and identifier n to refer to constructors, functions, and behaviours. The syntax of expressions is standard with the exception of the recover keyword - more in sec. 4.

The novel element of the syntax is the inclusion of reference capability annotations κ on types and functions, where:

κ ∈ {iso, trn, ref, val, box, tag}

These reference capabilities are the foundation of our type system.

Types consist of a class or actor identifier S followed by a reference capability κ. In addition, extended types ET can be unaliased, . An unaliased type is created with constructors and destructive reads - more in sec. 4.

The over-bar notation indicates a sequence of elements such as F, with the convention that the nth element is referred to as F_n. Similarly, x : T indicates a pairwise sequence of identifiers and types. To reduce notation, we assume a fixed program P.

### 4. Type system

The type system has the format \( \Gamma \vdash e : ET \) and is defined in fig. 3. The following aspects required special attention:

1. The treatment of operations which discard aliases.
2. The distinction between operations which introduce stable aliases (i.e. paths that survive the execution of a term) vs. those which create only temporary aliases.
3. Reference capabilities when accessing fields.
4. Reference capability recovery.
5. The treatment of actors.

**Operations which discard aliases** Assignment operations discard aliases, as they return the previous value of the left-hand side (ASNLOCAL and ASNFIELD) after overwriting it. The fact that an alias has been discarded is important in the cases where the reference capability is unique (iso or trn).

We indicate this through the unaliased annotation , which expresses that there is no stable path to the corresponding object.

Because unaliasing is of importance only when the underlying reference capability is iso, trn or ref, we have defined the unaliasing operation U, which takes a type and returns an extended type, cf. def. 1. This operator is used whenever an alias is discarded (cf. T-ASNLOCAL, T-ASNFIELD).

Object constructors also introduce unaliased values, as indicated in the rule T-CCTOR. Also, null has no stable alias, and thus is unaliased, cf. T-NULL. While the full Pony language has no null, we use it here to simplify the modelling of consume x, which is treated as (x = null).

**Distinction between introducing stable or temporary aliases** Some operations introduce stable aliases (eg. assignment), while others introduce only unstable ones (eg. field read). We express the distinction in the type system through the difference between the type judgments \( \Gamma \vdash e : ET \) and the aliased type judgment \( \Gamma \vdash_A e : ET \). For example, when assigning an expression e to a variable x, the right-hand side is typed in the judgment \( \Gamma \vdash_A (\text{cf. T-ASNLOCAL}) \). The aliased judgement is also applied to the receiver and arguments of method calls and asynchronous behaviours (T-SYNC and T-ASYNC), the arguments to object and actor constructors (T-CCTOR and T-ATOR), and the right-hand side of a field assignment (T-ASNFIELD).

The aliased type judgement \( \Gamma \vdash_A e : ET \) is defined in terms of the unaliased type judgement \( \Gamma \vdash e : ET' \), where ET has to be a super-type of the aliased version of ET', i.e. \( \mathcal{A}(ET') \leq ET \). The operation \( \mathcal{A}(ET) \) gives the type that an alias of ET would have. When aliasing an unaliased type there is no previous alias to consider, and therefore \( \mathcal{A}(S \kappa) = S \kappa \). For other types, the result must be the minimal super-type of the underlying type which is locally compatible with it, i.e. \( \mathcal{A}(S \kappa) = S \kappa' \) where \( \kappa' \leq \mathcal{A}(\kappa') \) and \( \mathcal{A}(\kappa') \) does not locally deny \( \kappa' \).

**Definition 1.** Aliasing and unaliasing.

- \( \mathcal{A}(S \kappa) = S \kappa \)
- \( \mathcal{A}(S \kappa) = \begin{cases} S \text{tag} & \text{iff } \kappa = \text{iso} \\ S \text{box} & \text{iff } \kappa = \text{trn} \\ S \kappa & \text{otherwise} \end{cases} \)
- \( \mathcal{U}(S \kappa) = \begin{cases} S \kappa & \text{iff } \kappa \in \{\text{iso}, \text{trn}, \text{ref}\} \\ S \kappa & \text{otherwise} \end{cases} \)

Thus, through a combination of aliasing and unaliasing, we can obtain unique types when needed. For example, for x and y of type c trn, the assignment x = y is illegal, because the aliased type of y is c box and c box \( \not\leq c \text{trn} \). However, the assignment y = consume x is legal, because the type of consume y is c trn, and the alias of c trn is c trn.

**Reference capabilities at field read** When reading a field f from an object i we obtain a temporary. The reference capability of this temporary must be a combination of \( \kappa \), the reference capability of the path leading to \( \iota \), and \( \kappa' \), the reference capability with which \( \iota \) sees the field. We express

<table>
<thead>
<tr>
<th>( \kappa )</th>
<th>( \kappa' )</th>
</tr>
</thead>
<tbody>
<tr>
<td>iso</td>
<td>iso</td>
</tr>
<tr>
<td>trn</td>
<td>trn</td>
</tr>
<tr>
<td>ref</td>
<td>ref</td>
</tr>
<tr>
<td>val</td>
<td>val</td>
</tr>
<tr>
<td>box</td>
<td>box</td>
</tr>
<tr>
<td>tag</td>
<td>tag</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \kappa )</th>
<th>( \kappa' )</th>
</tr>
</thead>
<tbody>
<tr>
<td>iso</td>
<td>iso</td>
</tr>
<tr>
<td>trn</td>
<td>trn</td>
</tr>
<tr>
<td>box</td>
<td>box</td>
</tr>
<tr>
<td>tag</td>
<td>tag</td>
</tr>
<tr>
<td>box</td>
<td>tag</td>
</tr>
<tr>
<td>box</td>
<td>box</td>
</tr>
<tr>
<td>tag</td>
<td>tag</td>
</tr>
</tbody>
</table>

Table 2. Viewpoint adaptation.
this through the operator $\triangleright$, defined in table 2. When reading a field through an origin, the result must not violate the deny properties of either the origin or the field. For example, reading a ref field from an iso reference returns tag - thus we do not violate the deny properties of the origin or the field itself.

Storing a reference into a field of an object $\iota$ is legal if the type of the reference is both a subtype of the type of the field and also safe to write into the origin. The relation $\kappa \triangleleft \kappa'$, as defined in table 3, expresses which reference capabilities $\kappa'$ are safe to write into into origin $\kappa$. When writing to a field through an origin, no alias of the object being written may exist that would violate the deny properties of the origin. Therefore, all entries for val, box and tag are empty. Moreover, only iso, val or tag references may be stored into an iso origin; all other writes would violate the region introduced by the iso origin.

**Reference capability recovery** The evaluation of an expression which has access only to sendable variables (i.e. iso, val, and tag) will return a sendable type. This is an extension of previous work on recovery [19], which is related to work on borrowing [22]. We introduce such expressions through the recover keyword (T-REC). The return type of recover $e$ is the sendable version of the return type of $e$. For example, if $e$ has type ref, then recover $e$ has type iso, and if $e$ has type ref0, then recover $e$ has type iso0.

**Definition 2.** Reference capability recovery

$$\mathcal{R}(\mathcal{S} \mathcal{k} \phi) = \begin{cases} \text{Siso} & \text{iff } \kappa \in \{\text{iso, trn, ref}\} \\ \text{Sval} & \text{iff } \kappa \in \{\text{val, box}\} \\ \text{Stag} & \text{otherwise} \end{cases}$$

$\mathcal{R}(\mathcal{E} \mathcal{T})$ is the sendable reference capability that retains the same local read and/or write guarantee. In other words, a writeable reference capability can become iso and a readable reference capability can become val. In Pony, explicit recover expressions are used along with implicit recovery detected by the compiler.

**The treatment of actors** Actors introduce the question of who may read or update the actor’s fields, the possibility of synchronous calls on actors, and the type required for asynchronous calls.
Field read and write requires that the actor should see itself as a ref. As a result, any other actor will see it as tag. Therefore no other actor except the current one will be allowed to observe an actor’s fields - a nice consequence of the type system.

By a similar argument, because the actor sees itself as ref, any other paths that point to it will do so as box, ref, or tag, and this means that the actor may call synchronous methods on itself, provided that the receiver reference capability of the method declaration is ref, box, or tag. Interestingly, for asynchronous (behaviour) calls, the receiving actor only needs to be seen as a tag (T-ASYNC), even though the receiver reference capability in the behaviour is ref. This is in contrast to method calls, where the receiver object/actor has to be seen as a reference capability which is a subtype of the receiver reference capability in the method declaration. The looser requirement for actors is sound, because, as discussed above, no other actor may obtain access to the actor’s state.

5. Operational semantics

The operational semantics has the shape $\chi \rightarrow \chi'$, where $\chi, \chi'$ are heaps mapping object addresses $\omega$ to their class identifier and their fields, and actor addresses $\alpha$ to their actor identifier, their fields, their message queue, their stack, and the next expression to execute. Runtime entities are defined in fig. 5. We use some shorthand notation for clarity - more in app. sec. ??.

We use $x$ to indicate a source identifier, $\tau$ to indicate a temporary identifier, and $y$ and $z$ to indicate identifiers which may be either.

A call stack consists of an actor address $\alpha$ followed by a sequence of frames $\varphi$. A frame consists of the method identifier, a mapping of its parameters to values, and an expression hole. The latter is the continuation of the caller and will be executed by the previous frame when the current activation terminates.

The auxiliary judgement $\chi, \sigma, e \leadsto \chi', \sigma', e'$ expresses local execution within a single actor. $M$ and $F$ return method and field declarations. They are defined in app. sec. ??.

Local execution is defined in fig. 6. EXPRHOLE allows execution to propagate to the context. FLD, NULL, and SEQ are as expected.

ASNLOCAL and ASNFLD combine assignment with a destructive read, returning the previous value of the left-hand side. The resulting value is unaliased: while there may be other paths pointing to the value in the program, this one no longer does. In effect, one alias to the value has been discarded. The existence of unaliased values is used in the type system, where T-ASNLOCAL and T-ASNFIELD both return an unaliased type, as explained in sec. 4.

SYNC and RETURN describe synchronous method call and return. In SYNC, method $m$ is called on object or actor $\iota$. The method parameters $x$ and the method body $e$ are looked up using the method $m$ and the type $S$ of $\iota$ from the heap. A new frame is pushed on to the stack, consisting of $m$, the address of the receiver, the values of the arguments, and the continuation. In RETURN, the topmost frame is popped from the stack and execution continues.

ASYNC and BEHAVE describe asynchronous method calls and execution. In ASYNC, a message consisting of the behaviour identifier $b$ and the arguments is appended to the receiver’s message queue. In BEHAVE, an actor with an empty call stack and a non-empty message queue removes the oldest message from the queue, and pushes a new frame on the stack.

CTOR and ATOR describe the construction of new objects and actors. In CTOR, a new address $\omega$ is allocated on the heap and the fields are initialised to null3. A new frame is pushed on the stack in the same way as for SYNC. In ATOR, instead of pushing a new frame on the stack, the new actor’s queue is initialised with a constructor message containing the constructor identifier $k$ and the arguments. The first local execution rule for a new actor will be BEHAVE, which will execute the body of the constructor $k$.

REC is a no-op in the operational semantics, but has an impact in the type system, where T-REC affects the reference capability of the result of the expression.

GLOBAL defines global execution and says that if an actor can execute, then its stack and next expression to execute will be updated.

6. Related Work

Linear types [31] provide the basis for uniqueness type systems. The insight that a type that is usable only once allows for mutation in a pure functional language leads directly to using linearity for concurrency-safe mutation [5]. A combination of unique pointers and ownership types [14] is used in PEFJ [7] to accomplish this.

In [10], a set of capabilities and exclusive capabilities, including identity, is used to build a uniqueness and immutability type system. Several important concepts are articulated

3 This is a simplification. In Pony, we support object initialisation, and have no null values.
in this work, including the notion that exclusive capabilities
deny the existence of capabilities through other aliases, the
use of destructive reads to manage capabilities, and the ex-
istence of the null capability (similar but not identical to tag
in our system).

Fractional permissions [9] encode uniqueness and im-
mutability as well as providing implicit static alias tracking
without alias analysis.

Relaxing the notion of uniqueness to external uniqueness [12] and separate uniqueness [22] allows for richer and more complex data structures to be simply encoded while maintaining all of the useful properties of linear types.

Using ownership types to express immutability at the
object and reference level in OIGJ [32], rather than at the
class level, allows immutable references to objects of any
type.

In Kilim [29], tree-structured messages are used to com-
bine work on uniqueness with zero-copy messages between
actors. While this is a significant restriction, the combination of actor-model concurrency, uniqueness, immutability and destructive read semantics is powerful. External uniqueness has also been extended to cover actor-model concurrency [13], providing a richer type system without tree-structure requirements. In [30], access permissions are combined with data flow analysis for implicit concurrency, which is in some sense the inverse of actor-model concurrency.

In [19], reference capabilities combined with viewpoint adaptation and recovery build a powerful data race free type system with significant usability advantages for the programmer. In addition, external uniqueness is relaxed even further to isolation, where immutable portions of an isolated object can be aliased externally.

In [6], a type and effect system for deterministic semantics is provided. This is a powerful system, but does not provide the unbounded non-deterministic semantics available in the actor-model.

In Rust [24], atomic reference counts, mutexes, allow properties, and ownership types are combined to achieve data race freedom. The use of both run-time and compile-time methods, and the addition of an unsafe module that can violate the type system, is an interesting compromise approach.

Our work is built on a deny properties [17] model instead
of a permissions or fractional permissions model. We show
that the type annotations used in related work are all expres-
sions of these deny properties, and that additional annota-
tions exist (particularly trn and the use of tag for typing
actors). We extend viewpoint adaptation and add our concept
of safe-to-write, allowing direct manipulation of isolated
types without recovery. Our use of tag with the actor-model
gives us a copy-less, lock-less operational semantics.

In table 4, we summarise some features of our work
and compare with those in Gordon et al. [19]. Àémium
### Table 4. Feature comparison.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Our Work</th>
<th>Gordon</th>
<th>Æminium</th>
<th>DPJ</th>
<th>Kilim</th>
<th>Haller</th>
<th>Scala</th>
<th>Erlang</th>
<th>Rust</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero-copy</td>
<td>√</td>
<td></td>
<td></td>
<td>√</td>
<td></td>
<td>√</td>
<td></td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Data-race free</td>
<td>√</td>
<td></td>
<td></td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Statically data-race free</td>
<td>√</td>
<td></td>
<td></td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Non-tree messages</td>
<td>√</td>
<td></td>
<td></td>
<td>√</td>
<td></td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Read unique (iso)</td>
<td>√</td>
<td></td>
<td></td>
<td>√</td>
<td></td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Write unique (trn)</td>
<td></td>
<td></td>
<td></td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mutability (ref)</td>
<td>√</td>
<td></td>
<td></td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Immutability (val)</td>
<td>√</td>
<td></td>
<td></td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cyclic immutability</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Identity (tag)</td>
<td></td>
<td></td>
<td></td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Destructive read</td>
<td>√</td>
<td></td>
<td></td>
<td>√</td>
<td></td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>Recovery</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Using uniques (iso ⊗ x)</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Actors</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

7. Implementation and benchmarking

We have implemented a native code compiler using our type system and a custom actor-model runtime, including the scheduler, memory allocator, garbage collector, message queues, etc. We have also implemented a standard library and several real-world data analytics programs. Our experience so far leads us to believe our reference capabilities system is expressive and easy to use, and the language is suitable for any problem that displays non-deterministic concurrency and mutable state. Specific examples include data analytics, financial systems, and video games.

To minimise the required annotations, Pony uses default reference capabilities (tag for actors, ref for objects, val for both built-in and user-defined primitives), while allowing the default reference capability for a type to be overridden (e.g. to default String to val instead of ref). In addition, the compiler guides the programmer as to which annotations should be used, infers annotations locally, and performs automatic recovery in some circumstances. As a result, when implementing the HPCC RandomAccess benchmark we require just 8 reference capability annotations and 3 uses of recover in 249 LOC. In approximately 10k LOC in the standard library, 89.3% of types required no annotation.

Deny properties are also amenable to a highly efficient implementation. We have benchmarked our language against other actor-model languages with the CAF [11] benchmark suite [2] and against MPI with the HPCC RandomAccess benchmark [1]. Results are the average of 100 runs, normalised against Erlang performance on a single core such that performance improvement linear to core count would be shown as a straight line sloping up. We chose to normalise against Erlang because it is a mature system with consistent performance across core counts, with little jitter.

In fig. 7, we show actor creation performance when creating an interconnected tree of actors that cannot be collected until the program completes (the worst case for Pony). Here, we are garbage collecting actors themselves [15] as well as objects, but still outperforms existing systems other than CAF, which is neither garbage collected nor data-race free. In fig. 8, we show performance of a highly contended mailbox, where additional cores tend to degrade performance. In fig. 9, we show performance of a mixed case, where a heavy message load is combined with brute force factorisation of large integers.

In fig. 10, we show a benchmark that is not tailored for actors: we take the RandomAccess benchmark from high-performance computing, which tests random access memory subsystem performance, and demonstrate that our imple-
The full Pony language as implemented in the compiler includes additional features, such as generic types, traits, structural types, type expressions (unions, intersections and tuples), a non-null type system, sound constructors, pattern matching, exceptions, and garbage collection. The Pony runtime will eventually support distributed computation, without a reduction in single-node performance.

The compiler, a web-based development sandbox, and a language tutorial are available at http://ponylang.org.

8. Conclusions and further work

We have used deny properties to provide a more fundamental basis for uniqueness and immutability. We have uncovered a new form of uniqueness, write uniqueness, and have explored the use of an identity reference capability for asynchronous method calls. Our extensions to viewpoint adaptation, including safe-to-write semantics, aliasing for non-reflexive sub-typing, and unaliased types, allow more operations on unique types.

In future work, we intend to extend the formalisation in this paper to prove soundness, and to cover additional type system features such as generics, algebraic data types, and a non-null type system. We also intend to formalise our use of the type system to improve both concurrent and distributed garbage collection. In addition, we intend to put

---

9 We show only power-of-two core counts because the MPI implementation is optimised for this case.
together additional benchmarks to improve our performance evaluation.

References
- $\varphi(x) = \varphi \downarrow_2 (x) \downarrow_1$
- $\varphi[x \mapsto v] = (\varphi \downarrow_1, \varphi \downarrow_2 [x \mapsto v], \varphi \downarrow_3)$
- $\chi(t,f) = \chi(v) \downarrow_2 (f)$
- $\chi[\omega; f \mapsto v] = \chi[\omega \mapsto (\chi(\omega) \downarrow_1, \chi(\omega) \downarrow_2 [f \mapsto v])]$
- $\chi[\alpha; f \mapsto v] = \chi[\alpha \mapsto (\chi(\alpha) \downarrow_1, \chi(\alpha) \downarrow_2 [f \mapsto v], \chi(\alpha) \downarrow_3, \chi(\alpha) \downarrow_4$ $\chi(\alpha) \downarrow_5)]$
- $\chi[\alpha \mapsto (\alpha, e)] = \chi[\alpha \mapsto (\chi(\alpha) \downarrow_1, \chi(\alpha) \downarrow_2, \chi(\alpha) \downarrow_4, \chi(\alpha) \downarrow_5)]$
- $\chi[\alpha \mapsto P] = \chi[\alpha \mapsto (\chi(\alpha) \downarrow_1, \chi(\alpha) \downarrow_2, P, \chi(\alpha) \downarrow_4, \chi(\alpha) \downarrow_5)]$

Figure 11. Auxiliary definitions

Appendix

We use the naming conventions given in fig.2, and the shorthands defined in fig. 11.

Lookup functions are defined in fig. 12. Function $\mathcal{P}$ returns a type definition for a class identifier $C$ or actor identifier $A$. This contains the fields $F$, constructors $K$, functions $M$, and behaviours $B$ defined for that type. Since classes have no asynchronous behaviour, the last entry in $\mathcal{P}(C)$ is empty, i.e. $\varepsilon$. Function $\mathcal{F}$ returns the identifiers of all fields defined in a type $S$, and function $\mathcal{F}$ returns the type of field $f$ in $S$. Function $\mathcal{M}$ returns method information for some method in $S$. This is overloaded on both the method identifier and the type identifier in order to handle class constructors, actor constructors, synchronous methods (functions) and asynchronous methods (behaviours). The result is a tuple of: the receiver type, the names and types of the parameters, the body of the method, and the return type. The reference capability of the receiver and the return type can vary for synchronous methods, but not for constructors or asynchronous methods. Constructors always operate on a $\text{ref}$ receiver, since the constructor must write to the new object’s fields, and return a $\text{ref}$ result, since the new object is initially mutable but also unalised, as the constructor’s reference to the receiver ($\text{this}$) is discarded when the constructor returns. This allows a constructor that is passed only sendable references as parameters to be embedded in a recover expression, which allows constructing an object with any reference capability. Asynchronous methods always operate on a $\text{ref}$ receiver. This is because the receiver of an asynchronous method is always an actor; when the body is executed, a new stack with the receiver as the root actor is created. Since each actor executes the body of a single behaviour (or asynchronous constructor) at any given time, every behaviour body can read from and write to the receiver. Since an asynchronous method cannot, by definition, perform any operations at the call site before returning, the only possible return values are the receiver or $\text{null}$. We have chosen to return the receiver to allow chaining method calls.

Figure 12. Lookup functions

\[
\begin{align*}
\mathcal{F}(S) & = \{ f | \text{var } f : T \in F \} \\
\mathcal{F}(S,f) & = T \\
\mathcal{P}(C) & = \mathcal{F}(C) \quad (\text{newk}(x:T) \Rightarrow e) \in \mathcal{K} \\
\mathcal{M}(C,k) & = (\text{Cref}, \mathcal{F}, e, \text{Cref}) \\
\mathcal{P}(A) & = \mathcal{F}(A) \quad (\text{be}(x : T) \Rightarrow e) \in \mathcal{B} \\
\mathcal{M}(A,b) & = (\text{Avar}, \mathcal{F}, e, \text{Aref})
\end{align*}
\]

Figure 13. Well-formed programs

\[
\begin{align*}
\varphi(x) = \varphi & \downarrow_2 (x) \downarrow_1 \\
\varphi[x \mapsto v] = (\varphi \downarrow_1, \varphi \downarrow_2 [x \mapsto v], \varphi \downarrow_3) \\
\chi(t,f) = \chi(v) & \downarrow_2 (f) \\
\chi[\omega; f \mapsto v] = \chi[\omega \mapsto (\chi(\omega) \downarrow_1, \chi(\omega) \downarrow_2 [f \mapsto v])] \\
\chi[\alpha; f \mapsto v] = \chi[\alpha \mapsto (\chi(\alpha) \downarrow_1, \chi(\alpha) \downarrow_2 [f \mapsto v], \chi(\alpha) \downarrow_3, \chi(\alpha) \downarrow_4$ $\chi(\alpha) \downarrow_5)] \\
\chi[\alpha \mapsto (\alpha, e)] = \chi[\alpha \mapsto (\chi(\alpha) \downarrow_1, \chi(\alpha) \downarrow_2, \chi(\alpha) \downarrow_4, \chi(\alpha) \downarrow_5)] \\
\chi[\alpha \mapsto P] = \chi[\alpha \mapsto (\chi(\alpha) \downarrow_1, \chi(\alpha) \downarrow_2, P, \chi(\alpha) \downarrow_4, \chi(\alpha) \downarrow_5)] \\
\mathcal{F}(S) & = \{ f | \text{var } f : T \in F \} \\
\mathcal{F}(S,f) & = T \\
\mathcal{P}(C) & = \mathcal{F}(C) \quad (\text{newk}(x:T) \Rightarrow e) \in \mathcal{K} \\
\mathcal{M}(C,k) & = (\text{Cref}, \mathcal{F}, e, \text{Cref}) \\
\mathcal{P}(A) & = \mathcal{F}(A) \quad (\text{be}(x : T) \Rightarrow e) \in \mathcal{B} \\
\mathcal{M}(A,b) & = (\text{Avar}, \mathcal{F}, e, \text{Aref})
\end{align*}
\]

Figure 14. Auxiliary well-formedness definitions