Deny Capabilities for Safe, Fast Actors

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 [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 wish to 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 must be amenable to a highly efficient implementation.

Existing approaches to static data race freedom use capabilities [24] to describe what a reference is allowed to do. In previous work, capabilities have been expressed as permissions [10], fractional permissions [9], uniqueness [12], immutability [26], and isolation [19] (a refinement of separate uniqueness [22], which is a refinement of external uniqueness [12]). One issue with these systems is that what a reference is allowed to do must be used to reason about what other references to the same object must be prevented from doing.

We have taken a different approach and use capabilities to describe what other aliases are denied by the existence of a reference. We use a matrix of deny properties [17], with notions such as isolation, mutability, and immutability all being derived from these properties. What other references to the same object can do is explicit rather than implied.

Other approaches have combined actors with data-race freedom [13, 22, 27]. 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 capabilities, we were able to develop a more flexible type system that supports such patterns. 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 capability types, tag and trn (transition). A tag 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 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 references.

We also extend viewpoint adaptation [16, 19] to apply to every 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 of a reference may have a different capability from the initial reference. This addresses a key issue in 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 capabilities also provide a static region system [21], requiring no additional annotation. The \texttt{txn} 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.

**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, \texttt{txn}.
- A capability, \texttt{tag}, that can be used to type actors.
- **Viewpoint adaptation** and safe-to-write semantics for reading and writing unique types.
- **Temporary identifiers** to safely traverse unique structures.
- 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.
- A formal system.

Moreover, a native code compiler, runtime, and standard library exist, which we use 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 capabilities as deny properties in sec. 2, a formal analysis of data race free heaps in sec. 3, a formal type system in sec. 4, a syntax in sec. 5, an operational semantics in sec. 6, a soundness proof in sec. 7, related work in sec. 8, an implementation and benchmarks in sec. 9, and conclusions and further work in sec. 10.

2. **Capabilities as deny properties**

Rather than indicate which operations are allowed on a reference, our capabilities indicate what operations are denied on other references to the same object. 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 capability stands for a pair of local and global deny properties. These are shown in table 1. For example, \texttt{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 capability can deny local aliases that it allows globally. Therefore, some cells in the matrix are empty. For example, there is no 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 \texttt{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.

If an actor has a \texttt{box} reference to an object, no other reference can be used by other actors to write to that object. This means that other actors may be able to read the object and other references 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 \texttt{box} for immutability allows a program to enforce read-only behaviour, similar to const in C/C++. For example:

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

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

In \texttt{size2}, by indicating that the receiver has \texttt{val} capability, we make a stronger guarantee: we deny both local and global write aliases. As a result, it is not possible for \texttt{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 \texttt{val} reference has the same local and global deny properties, it is possible to send a \texttt{val} reference to another actor. A \texttt{val} reference is effectively a value type, similar to values in functional languages.

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

We use the keyword \texttt{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 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 a reference 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 reference.

A tag reference has no deny properties, but it can be used for asynchronous method calls, i.e., calling behaviours. A capability with no permissions has appeared in previous work [25], but without allowing asynchronous method calls.

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 use a destructive read.

An assignment expression returns the previous value of the left-hand side of an assignment rather than the value of the right-hand side, making assignment equivalent to a destructive read. Our type system introduces the concept of unaliased types, annotated with o, in order to type values for which an alias has been removed. Here, the destructive read 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 references 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.

In order to pass mutable data between actors, we use iso references. All mutable 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 references to the object sent.

Table 1. Capability matrix. Capabilities in italics are sendable.
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 read 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 capability in the future, in order to share it with another actor.

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

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

Here\(^1\) 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 box, which is a subtype of Booking val and can be shared with the Accountant actor. Without a write unique type, 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. Consistent heap visibility

The core of the soundness of our approach is consistent heap visibility, which requires that aliasing in the heap must satisfy all the deny properties specified by the capabilities attached to fields and variables. This leads to the notions of local and global compatibility. Namely, two capabilities are locally compatible \( \kappa \sim \ell \kappa' \) if neither has a local deny property that prevents the existence of the other. Similarly, they are globally compatible, \( \kappa \sim g \kappa' \), if neither has a global deny property that prevents the existence of the other. These relationships are defined in table 2, e.g. \( \text{ref} \sim \ell \text{ref} \) but \( \text{ref} \not\sim g \text{ref} \). Both relations are symmetric.

In fig. 1, we show a diagrammatic representation of a heap \( \chi_0 \) which contains actors \( \alpha_1 \) and \( \alpha_2 \), and objects \( t_{10} \ldots t_{19} \). The top rectangles indicate stack frames, for example \( \chi_0(t_{11}) = (\_ , \_ \cdot \alpha_1 \cdot \varphi_1 \cdot \varphi_2 \cdot \_ ) \) and \( \varphi_1(t_2) = t_{18} \). The objects are in rounded boxes, and the annotated arrows indicate the contents of their fields, e.g. \( \chi_0(t_{14}, f_{10}) = t_{19} \). The annotations next to the field identifiers (ref, val, etc.) give types to the variables. Note that \( \alpha_1 = t_{10} \) and \( \alpha_2 = t_{14} \).

For consistent heap visibility we require that different paths originating from the same actor and pointing to the same object have locally consistent visibility, while paths originating from different actors and pointing to the same object have globally consistent visibility. For example, in fig. 1 the path this.f1.f5.f8 starting at the first frame of actor \( \alpha_1 \) and the path this.f10 at the first frame of actor \( \alpha_2 \) are aliases, as they both reach object \( t_{19} \). The first path sees \( t_{19} \) as t.tag, while the second sees it as t.val. These are globally compatible capabilities, and therefore these paths preserve consistent heap visibility. On the other hand, if we added a ref field to \( t_{15} \), such that it pointed to \( t_{19} \), the resulting capabilities would not be globally compatible.

For the formal definition of consistent heap visibility, we need notions of:

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

Table 2. Compatible capabilities.

---

\(^1\) In this example, we are using generic types and default capabilities (ref for objects and tag for actors). While the full language supports these, we will not formalise them here.
Environments the annotations. Thus, we have an implicit global environment. In fig.
poraries) to extended types and global environments,

\[ \Delta, \chi, \iota \vdash \iota' : \kappa, p, p', t \text{ where } \text{Stable}(\Delta, \alpha, p) \text{ and } \text{Stable}(\Delta, \alpha, p') \]

1. If \( \Delta, \chi, \alpha \vdash \iota : \kappa \) and \( \Delta, \chi, \alpha' \vdash \iota' : \kappa' \) and \( \alpha \neq \alpha' \) then \( \kappa \sim_g \kappa' \)

2. If \( \Delta, \chi, \alpha \vdash \iota : \kappa, p \) and \( \Delta, \chi, \alpha \vdash \iota' : \kappa', p' \) then
   (a) \( \chi, \alpha \vdash p \sim p' \) or
   (b) \( \kappa \sim \iota \kappa' \)

3. If \( \Delta, \chi, \alpha \vdash \iota : \kappa \) and \( \Delta, \chi, \alpha \vdash \iota' : \kappa', p' \) and \( \Delta, \chi, \iota \vdash \iota' : \kappa'' \) and \( \kappa \in \{\text{iso}, \text{trn}\} \) then
   (a) \( \chi, \alpha \vdash \iota \in p' \) or
   (b) \( \kappa'' \in \{\text{val}, \text{box}\} \) and \( \kappa' \sim_g \text{val} \) or
   (c) \( \kappa'' \in \{\text{iso}, \text{trn}, \text{ref}\} \) and \( \kappa \sim \iota \kappa' \)

4. If \( \Delta(\alpha, i, t) = S \kappa \) and \( \kappa \in \{\text{iso}, \text{trn}\} \) and \( \chi(\alpha, i, t) = \chi(\alpha, p_1) = \iota \)
   (a) \( p_1 = (i, t) \) or
   (b) \( \exists \iota', \kappa', p_2, \bar{I} \) such that
     i. \( \kappa \leq \kappa' \)
     ii. \( \kappa' \in \{\text{iso}, \text{trn}\} \)
     iii. \( p_1 = p_2 \cdot \bar{I} \)
     iv. \( \Delta, \chi, \alpha \vdash \iota' : \kappa', p_2 \)
     v. \( \Delta, \chi, \iota' \vdash \iota : \kappa, \bar{I} \)

Given in fig. 4. The definition ensures that \( \kappa \triangleright \kappa' = \kappa' \) if \( \kappa \)
iscritable (deep mutability), \( \kappa \triangleright \kappa' = \text{val} \) if either \( \kappa \) or \( \kappa' \)
isval (deep immutability) and box \( \triangleright \kappa' = \text{box} \) unless \( \kappa' \in \{\text{iso}, \text{val}, \text{tag}\} \). For example, iso \( \triangleright \text{ref} = \text{ref} \).

The rules in fig. 3 say that an address sees itself as ref, an actor sees a stack identifier as the capability provided by \( \Delta \), and an address sees another address as a deep viewpoint adapted capability. Note that, for visibility, tag types are not seen. Therefore, our example gives us:

\[ \Delta_0, \chi_0, \alpha_1 \vdash \iota_10 : \text{ref}(1, \text{this}), \quad \text{but also} \quad \Delta_0, \chi_0, \alpha_1 \vdash \iota_10 : \text{box}(1, \text{this}) \cdot f1 \cdot f2. \]
\[ \Delta_0, \chi_0, \alpha_2 \vdash \iota_19 : \text{val}(1, \text{this}) \cdot f10, \quad \text{but also} \quad \Delta_0, \chi_0, \alpha_1 \vdash \iota_{19} : \text{tag}(1, \text{this}) \cdot f1 \cdot f5 \cdot f8. \]

In fig. 4, two paths are compatible if they share the last step or they are the same identifier with no fields, an address \( \iota \) in a path if some prefix of the path points to \( \iota \), and a path is stable, Stable(\( \Delta, \alpha, p \)), if its initial identifier is not a unique temporary. For example, \( \chi_0, \alpha_2 \vdash (1, \text{this}) \cdot f10 \sim (1, \text{y1}) \cdot f10. \) Also, Stable(\( \Delta_0, \alpha_1, (1, \text{this}) \cdot f1 \cdot f4 \)) and \( \neg\text{Stable}(\Delta_0, \alpha_1, (2, t2) \cdot f9) \), even though the two paths are aliases.

We define consistent heap visibility in fig. 5. We require:
1. Global compatibility. Any two distinct actors that can see the same address must see that address with globally compatible capabilities.

2. Local compatibility. An actor that sees an address in multiple ways must either see compatible paths or locally compatible capabilities.

3. Containment properties of $\text{iso}$ and $\text{trn}$. Given $\alpha$ that sees $t$ as some unique $\kappa$ and sees $t'$ as $\kappa'$ via some stable $p'$, and given that $\iota$ sees $t'$ as $\kappa''$:
   \begin{itemize}
     \item[(a)] $\iota'$ must be contained by $\iota$, or
     \item[(b)] neither $\iota$ nor $\alpha$ can write to $t'$, or
     \item[(c)] $\iota$ can write to $t'$ and $\alpha$ sees $t'$ as locally compatible with $\kappa$.
   \end{itemize}

4. Properties of unique temporary identifiers. Given $t$ that points to $\iota$, some other path $p_1$ to the same $\iota$ must be either:
   \begin{itemize}
     \item[(a)] also $t$ or
     \item[(b)] that path $p_1$ must have a prefix $p_2$ that sees some $t'$ with a unique capability $\kappa''$ less precise than $\kappa$ and $t'$ must see $t$ as $\kappa$.
   \end{itemize}

An implication of well-formed visibility is that if two variables (temporary or otherwise) are aliases and one of them has unique type (aliased or unaliased) then 1) they come from the same actor and 2) they are either the same variable or they have locally compatible capabilities, cf. lemmas 8 and 9 in the appendix. Note that $WFV.1 \rightarrow 3$ are concerned with stable paths only, while $WFV.4$ is about unstable paths. In particular, $WFV.4$ allows a unique temporary to break the requirements from $WFV.3$ and alias something writeable from a unique.

The heap from fig. 1 has consistent visibility. The paths $(1,\text{this}) \cdot f1 \cdot f5 \cdot f8$ from $\alpha_1$ and $(1,\text{this}) \cdot f10$ from $\alpha_2$ satisfy $WFV.1$, while $(1,x1) \cdot f4$ from $\alpha_1$ and $(2,\text{this})$ from $\alpha_1$ satisfy $WFV.2$ and $WFV.3$. On the other hand, the temporary (2,t2) is not stable, and therefore not restricted by $WFV.2$ or $WFV.3$, but does adhere to $WFV.4$. Finally, the assignment $\text{this.f1.f5.f6 = this.f1.f5.f7}$ would break $WFV.2$, while setting $t2$ to point to $\iota_{15}$ would break $WFV.4$.

4. Type system

The type system has the format $\Gamma \vdash e : ET$ and is defined in fig. 6. The following aspects required special attention:

1. The treatment of operations which discard aliases.
2. The distinction between operations which introduce stable aliases vs. those which create only temporary aliases.
3. Capabilities when accessing fields.
5. The treatment of actors.

**Operations which discard aliases** Assignments operations discard aliases, as they return the previous value of the left-hand side ($\text{ASNLOCAL}$ and $\text{ASNFIELD}$) after overwriting it. The fact that an alias has been discarded is important in the cases where the capability is unique ($\text{iso}$ or $\text{trn}$). We indicate this through the unaliased annotation $\circ$, which expresses that there is no stable path to the corresponding object.

For example, the assignment $\text{this.f1.f5 = null}$ in the first frame of actor $\alpha_1$ in fig. 1 would return a new temporary which would be the unique reference to $\iota_6$. The type of this expression would be $S \text{iso}$ for some $S$. Because unaliasing is of importance only when the underlying capability is $\text{iso}$, $\text{trn}$ or $\text{ref}$, 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-ASNFLD).

Object constructors also introduce unaliased values, as indicated in the rule T-CTOR. Also, $null$ has no stable alias, and thus is unaliased, cf. T-NUL.

**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 $\vdash_A$ (cf. T-ASNLOCAL). The aliasing 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-CTOR and T-ATOR), and the right-hand side of a field assignment (T-ASNFLD).

The aliased type judgment $\Gamma \vdash_A e : ET$ is defined in terms of the unaliased type judgment $\Gamma \vdash e : ET'$, where $ET$ has to be a super-type of the aliased version of $ET'$, i.e. $A(ET') \leq ET$. The operation $A(ET)$ gives the type that an alias of $ET$ would have. When aliasing an unaliased type there is no previous alias to consider, and therefore $A(S \kappa) = S \kappa$. For other types, the result must be the minimal super-type of the underlying type which is locally compatible with it, i.e. $A(S \kappa) = S \kappa'$ where $\kappa' \leq A(\kappa')$ and $A(\kappa') \sim \kappa$.

**Definition 1.** Aliasing and unaliasing.

- $A(S \kappa) = S \kappa$
- $A(S \kappa) = \begin{cases} S \text{tag} & \text{iff } \kappa = \text{iso} \\ S \text{box} & \text{iff } \kappa = \text{trn} \\ S \kappa & \text{otherwise} \end{cases}$
- $U(S \kappa) = \begin{cases} S \kappa & \text{iff } \kappa \in \{\text{iso, trn, ref}\} \\ S \kappa & \text{otherwise} \end{cases}$
\[
\frac{x \in \Gamma}{\Gamma \vdash x : \Gamma(x)}\quad \text{T-LOCAL}
\]
\[
\frac{S \in P}{\Gamma \vdash \text{null} : S_{\text{iso}}}\quad \text{T-NUL}
\]
\[
\frac{\Gamma(x) = S \kappa \quad \Gamma \vdash A e : S \kappa}{\Gamma \vdash x = e : U(S \kappa)}\quad \text{T-ASNLOCAL}
\]
\[
\frac{M(S, m) = (T, x : T, e, ET)}{\Gamma \vdash e.m(\xi) : ET}\quad \text{T-SYNC}
\]
\[
\frac{M(C, k) = (C.f, x : C, e, Cref)}{\Gamma \vdash C.k(\xi) : Cref}\quad \text{T-CTOR}
\]
\[
\frac{\Gamma \vdash e : ET' \quad A(ET') \leq T}{\Gamma \vdash A e : T}\quad \text{T-ALIAS}
\]
\[
\frac{\Gamma \vdash e : S \kappa}{\Gamma \vdash e : S \kappa}\quad \text{T-SUBSUME}
\]
\[
\frac{\begin{array}{c}
ET \leq ET'' \quad ET'' \leq ET' \\
S \kappa \leq S \kappa \quad S \kappa \leq S \kappa'
\end{array}}{\kappa \leq \kappa'}
\]
\[
\text{iso} \leq \text{trn} \leq \{\text{ref, val}\} \leq \text{box} \leq \text{tag}
\]
\[
\text{Sendable}(T) \iff T = S \kappa \land \kappa \in \{\text{iso, val, tag}\}
\]

**Figure 6.** Expression typing

**Figure 7.** Sub-types and sendable types.

Thus, through a combination of aliasing and unaliasing, we can obtain unique types when needed. For example, for \(x\) and \(y\) of type \(\text{C} \text{trn}\), the assignment \(x = y\) is illegal, because the aliased type of \(y\) is \(\text{C} \text{box}\) and \(\text{C} \text{box} \neq \text{C} \text{trn}\). However, the assignment \(x = (y = \text{null})\) is legal, because the type of \(y = \text{null}\) is \(\text{C} \text{trn}\), and the alias of \(\text{C} \text{trn}\) is \(\text{C} \text{trn}\).

**Capabilities at field read** When reading a field \(f\) from an object \(i\) we obtain a temporary. The capability of this temporary must be a combination of \(\kappa\), the capability of the path leading to \(i\), and \(\kappa'\), the capability with which \(i\) sees the field. We express this through the operator \(\triangleright\), defined in fig. 3. This operator is less precise than \(\triangleright\), i.e., \(\kappa \triangleright \kappa' \leq \kappa \triangleright \kappa'\triangleright \kappa'\). The new temporaries introduced must preserve well-formed heap visibility, in particular WFV. \(\mathcal{A}\). These rules forbid temporary aliases to \text{trn} or \text{ref} fields of an \text{iso}, and therefore we obtain \(\text{iso} \triangleright \text{trn} = \text{iso} \triangleright \text{ref} = \text{tag}\). Also, they require that any aliases to \text{ref} fields of a \text{trn} are \text{box}, including temporary references. Therefore, \(\text{trn} \triangleright \text{ref} = \text{box}\).

\[
\frac{\Gamma \vdash e : S \kappa}{\mathcal{F}(S, f) = S' \kappa'}\quad \text{T-FLD}
\]
\[
\frac{\Gamma \vdash e : S' \kappa \triangleright \kappa'}{\Gamma \vdash e.f : S' \kappa \triangleright \kappa'}\quad \text{T-FLD}
\]
\[
\frac{\Gamma \vdash e : ET \quad \Gamma \vdash e' : ET'}{\Gamma \vdash e.e' : ET'}\quad \text{T-SEQ}
\]
\[
\frac{\Gamma \vdash e : S \kappa \quad \Gamma \vdash A e' : S' \kappa'}{\mathcal{F}(S, f) = S' \kappa' \quad \kappa' \leq \kappa'\prime \quad \kappa \triangleright \kappa'\prime \vdash \kappa \leq \kappa'\prime'}\quad \text{T-ASNFLD}
\]
\[
\frac{\mathcal{M}(A, b) = (A \text{ref}, x : T, e, A \text{tag})}{\Gamma \vdash A e : A \text{tag}}\quad \text{T-ASYNC}
\]
\[
\frac{\mathcal{M}(A, k) = (A \text{tag}, x : A, e, A \text{tag})}{\Gamma \vdash A e : A \text{tag}}\quad \text{T-ATOR}
\]
\[
\Gamma \backslash \{x \mid \neg \text{Sendable}(\Gamma(x))\} \vdash e : ET\quad \text{T-REC}
\]

**Table 3.** Viewpoint adaptation.

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

<table>
<thead>
<tr>
<th>(\kappa \triangleright \kappa')</th>
<th>(\kappa')</th>
</tr>
</thead>
<tbody>
<tr>
<td>iso</td>
<td>check</td>
</tr>
<tr>
<td>trn</td>
<td>check</td>
</tr>
<tr>
<td>ref</td>
<td>check</td>
</tr>
<tr>
<td>val</td>
<td>check</td>
</tr>
<tr>
<td>box</td>
<td>check</td>
</tr>
<tr>
<td>tag</td>
<td>check</td>
</tr>
</tbody>
</table>
Thus, taking our earlier example, the type of this.f1.f5 is iso, while the type of this.f1.f5.f6 is tag. Compare this with visibility, which gives $\Delta_0, \chi^{\alpha_1} \vdash \iota_7 : \text{ref.(1, this)} \cdot f1 \cdot f5 \cdot f6$.

Storing a reference into a field of an object $t$ 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 $\kappa \circ \kappa'$, as defined in fig. 4, expresses which reference capabilities $\kappa'$ are safe to write 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.

Notice, that these rules allow us to write to fields which are not readable, i.e. of type tag. For example, the field read this.f1.f5.f6 has type tag, but the field assignment this.f1.f5.f6 = (x1 = null) is legal even though the field f6 is ref and ref is not safe to write into iso. Namely, $\chi x1 = null$ has type iso, and aliased type iso, and iso $\leq$ ref, and iso is safe to write to iso.

**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 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 ref, then recover e has type iso.

**Definition 2.** Capability recovery

$$\begin{align*}
\mathcal{R}(\kappa, \phi) &= \begin{cases}
\text{Siso} & \text{if } \kappa \in \{\text{iso, trn, ref}\} \\
\text{Sval} & \text{if } \kappa \in \{\text{val, box}\} \\
\text{Stag} & \text{otherwise}
\end{cases} \\
\mathcal{R}(\text{ET}) &= \text{the sendable capability that retains the same local read and/or write guarantee. In other words, a writeable capability can become iso and a readable capability can become val.}
\end{align*}$$

**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 not be seen as a tag. However, since an actor sees itself as a ref (by fig. 3), any other actor will see it as tag (by WFV.1). 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, by WFV.2, any other actors 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 capability of the method declaration is ref or box. Interestingly, for asynchronous (behaviour) calls, the receiving actor only needs to be seen as a tag (T-ASYNC), even though the receiver capability in the behaviour is ref. This is in contrast to method calls, where the receiver object/actor has to be seen as a capability which is a subtype of the receiver 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.

**Further observations about the type system** In contrast to many type systems, typing is not covariant with the capabilities assigned to variables or fields. That is, $\Gamma \vdash e : \text{ET}$ and $\Gamma(x) = S \kappa$ and $\kappa' \leq \kappa$ does not imply that $\Gamma[x \mapsto S \kappa'] \vdash e : \text{ET'}$ for some type $\text{ET'}$. For example, take class C with a field f of type Cref, and let $\Gamma$ such that $\Gamma(x) = \text{Cref}$ and $\Gamma' = \Gamma[x \mapsto \text{Ctref}]$. Then $x.f = x$ is type correct in $\Gamma'$ but not in $\Gamma'$.

**5. Syntax**

In fig. 8 we present the syntax. 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 synchronous methods (functions) and constructors. We use the term method and identifier n to refer to con-
6. 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. 10. We use some shorthand notation for clarity - more in app. sec. 17.

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. A.

Local execution is defined in fig. 11. $EXPRHOLE$ allows execution to propagate to the context. $FLD$, $NULL$, and $SEQ$ are as expected.

$ASNLOCAL$ and $ASNFD$ 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 will be used in the type system, where $T-ASNLOCAL$ and $T-ASNFD$ 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 $\pi$ and the method body $\sigma$ 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 null. 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 capabilities of the result of the expression.

$EXCEPT$ is unusual in that it allows dereferencing null. We use it here simply to ignore the uninteresting (for our current purposes) behaviour of null.

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

7. Soundness

A heap $\chi$ is well-formed as defined in fig. 12 if all objects in the heap are well-formed, all actors in the heap are well-formed, and visibility is well-formed. An object is well-formed if all its fields belong to the type defined in the object’s class. An actor is well-formed if its stack frames and messages are well-formed. A stack frame is well-formed if 1) its receiver and arguments are well-formed, 2) all local identifiers are well-formed, 3) if it is the only stack frame,

---

Figure 10. Runtime entities
it has no continuation and the receiver is the actor. 4) if it is not the only stack frame, its return value and temporary identifiers are well-formed wrt. the previous frame, and 5) if it is the last frame, temporary identifiers are well-formed and the expression has the expected type.

Treatment of temporaries Temporaries with unique capabilities, iso or trn, are fragile: on the one hand they may break the encapsulation of other iso or trn objects. For example, because iso ⊢ iso = iso, a field read (FLD) may return a temporary pointing within the encapsulation of iso. On the other hand, an assignment to another field or variable might break their encapsulation.

We require that in a frame, no more than one temporary has an iso or trn capability, and this temporary appears on a field assignment or a field read. We also require that any temporaries that appear within a recover expression are either inaccessible from any frame or are only accessible through sendable local variables.

Definition 3. Well-formed temporaries. \( WFT(\Delta, \chi, \alpha, i, e) \) iff:

1. No temporary appears more than once in \( e \).
2. If \( T(\Gamma) \neq \emptyset \), then \( e \equiv E[e'] \), where \( e' \) is a redex of the form \( t.f = y \) and \( T(\Gamma) = \{ t \} \), where \( \Gamma = \Delta(\alpha, i) \) and \( T(\Gamma) \equiv \{ t \mid \Gamma(t) = S \land t \in \{ iso, trn \} \} \).
3. If \( e = E[recover e'] \) and \( \Delta, \chi, \alpha \vdash i' : \_ (i, t) \) and \( \Delta, \chi, \alpha \vdash i : \text{trn}(i', i'') \cdot E \) where \( t \) is free in \( e' \) then either \( Sendable(\Delta(\alpha, i', z)) \) or \( (i, z) = (i', t') \) and \( z \) is not free in \( E[i] \).

The requirements above do not apply to unaliased unique capabilities, e.g. iso or trn. When proving type preservation, we maintain the property \( WFT(\Delta, \chi, \alpha, i, e) \) by turning the types of temporaries with unique capabilities \( \kappa \in \{ iso, trn \} \) into their aliases, \( A(\kappa) \), as soon as the temporary is no longer involved in field reads or updates in the current redex. The type of the expression is preserved despite this change, because the type rules from...
Theorem 1. A well-formed heap ensures data race freedom.

∀Δ, χ, α₁, α₂, f, g, if

1. Δ ⊢ χ₀, and
2. χ(α₁) = (_..., σ₁, _, E₁[z₁, f = z₃]), and
3. χ(α₂) = (_..., σ₂, _, E₂[z₂, g])

then χ(α₁, σ₁ · z₁) = χ(α₂, σ₂ · z₂).

Proof. Follows from the type system and the application of WFV.1 (global consistency).

Theorem 2. Well-formedness is preserved.

∀Δ, χ, if Δ ⊢ χ₀ and χ → χ' then ∃Δ'. Δ' ⊢ χ'₀.

Proof. Follows from lemmas 17-20 in the appendix.

Atomicity: Because the type of any entity does not change, any readable reference is always readable, and so guarantees no other actor can write to it. This holds not just for methods, but for behaviours. As a result, theorem 1 guarantees that behaviours are atomic, a stronger guarantee than data-race freedom. In the full language, where null is absent, this is achieved without the null pointer exceptions that destructive read otherwise introduces. We will provide a full argument for atomicity and its importance in reasoning about actor-model programming in future work.

8. Related Work

Linear types [29] 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 PRFJ [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 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 existence of the null capability (similar but not identical to tag in our system).

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

Relaxing the notion of uniqueness to external uniqueness [12] allows for richer and more complex data structures to be simply encoded while maintaining all of the useful properties of linear types. In the same work, the concept of converting an externally unique reference to an immutable reference is developed.

Using ownership types to express immutability at the object and reference level in OIGJ [30], rather than at the class level, allows immutable references to objects of any type.

In Kilim [27], tree-structured messages are used to combine 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 [28], access permissions are combined with data flow analysis for implicit concurrency, which is in some sense the inverse of actor-model concurrency.

In [19], 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.

2 Kilim messages are data-race free but the rest of Java is not.
3 The proposed system is data-race free but the rest of Scala is not.
4 Rust uses atomic reference counts and read-writer locks to prevent data races.
5 Scala has types that are immutable by design, but cannot annotate references to mutable types as immutable.
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 [23], 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 expressions of these deny properties, and that additional annotations exist (particularly $\text{trn}$ and the use of $\text{tag}$ 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 $\text{tag}$ with the actor-model gives us a copy-less, lock-less operational semantics.

In table 5, we summarise some features of our work and compare with those in Gordon et al. [19], Äeminium [28], Deterministic Parallel Java [6], Kilim [27], Haller and Odersky [22], Scala, Erlang, and Rust [23].

<table>
<thead>
<tr>
<th>Feature</th>
<th>Our Work</th>
<th>Gordon</th>
<th>Äeminium</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>✓$^2$</td>
<td>✓$^3$</td>
<td>✓$^4$</td>
<td>✓</td>
<td>✓$^5$</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>✓$^2$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓$^3$</td>
</tr>
<tr>
<td>Read unique ($\text{iso}$)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓$^2$</td>
<td>✓$^3$</td>
<td>✓$^4$</td>
<td>✓$^5$</td>
<td>✓$^6$</td>
<td>✓$^7$</td>
</tr>
<tr>
<td>Write unique ($\text{trn}$)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Mutability ($\text{ref}$)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓$^2$</td>
<td>✓$^3$</td>
<td>✓$^4$</td>
<td>✓$^5$</td>
<td>✓$^6$</td>
<td>✓$^7$</td>
</tr>
<tr>
<td>Immutability ($\text{val}$)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓$^8$</td>
</tr>
<tr>
<td>Cyclic immutability</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓$^2$</td>
<td>✓$^3$</td>
<td>✓$^4$</td>
<td>✓$^5$</td>
<td>✓$^6$</td>
<td>✓$^7$</td>
</tr>
<tr>
<td>Identity ($\text{tag}$)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓$^2$</td>
<td>✓$^3$</td>
<td>✓$^4$</td>
<td>✓$^5$</td>
<td>✓$^6$</td>
<td>✓$^7$</td>
</tr>
<tr>
<td>Destructive read</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓$^2$</td>
<td>✓$^3$</td>
<td>✓$^4$</td>
<td>✓$^5$</td>
<td>✓$^6$</td>
<td>✓$^7$</td>
</tr>
<tr>
<td>Recovery</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓$^2$</td>
<td>✓$^3$</td>
<td>✓$^4$</td>
<td>✓$^5$</td>
<td>✓$^6$</td>
<td>✓$^7$</td>
</tr>
<tr>
<td>Using uniques ($\text{iso} \triangleright x$)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓$^2$</td>
<td>✓$^3$</td>
<td>✓$^4$</td>
<td>✓$^5$</td>
<td>✓$^6$</td>
<td>✓$^7$</td>
</tr>
<tr>
<td>Actors</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓$^2$</td>
<td>✓$^3$</td>
<td>✓$^4$</td>
<td>✓$^5$</td>
<td>✓$^6$</td>
<td>✓$^7$</td>
</tr>
<tr>
<td>Formal proof</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓$^2$</td>
<td>✓$^3$</td>
<td>✓$^4$</td>
<td>✓$^5$</td>
<td>✓$^6$</td>
<td>✓$^7$</td>
</tr>
<tr>
<td>Native compilation</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓$^2$</td>
<td>✓$^3$</td>
<td>✓$^4$</td>
<td>✓$^5$</td>
<td>✓$^6$</td>
<td>✓$^7$</td>
</tr>
</tbody>
</table>

Table 5. Feature comparison.

9. 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 implemented large portions of a standard library and several real world data analytics programs. Our experience so far leads us to believe our 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.

The language uses carefully chosen default capabilities to minimise the required annotations. 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 LINPACK GUPS (in app. F) we require just 8 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 HPC Challenge LINPACK GUPS [1]. Benchmarking was done on a 12-core 2.3 GHz

---

$^6$ A version of identity, none, appears in [25].
In fig. 13, we show actor creation performance. Here, our implementation is 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. 14, we show performance of a highly contended mailbox, where additional cores tend to degrade performance⁷. In fig. 15, we show performance of a mixed case, where a heavy message load is combined with brute force factorisation of large integers.

In fig. 16, we show a benchmark that is not tailored for actors: we take the GUPS benchmark from high-performance computing, which tests random access memory subsystem performance, and demonstrate that our implementation is significantly faster than the highly optimised MPI implementation⁸.

The full 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 compiler, a web-based development sandbox, and a language tutorial are available⁹.

10. 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 capability for asynchronous method calls. Our extensions to viewpoint adaptation, including safe-to-write semantics, aliasing for non-reflexive

---

⁷ In fig. 13 and 14, Scala performance with fewer than 3 cores has been elided to compress the y axis.

⁸ We show only power-of-two core counts because the MPI implementation is optimised for this case.

⁹ These are supplied in supplementary material.
sub-typing, and unaliased types, allow more operations on unique types.

In future work, we intend to extend the formalisation in this paper to cover and prove soundness for these features. We also intend to formalise our use of the type system to improve both concurrent and distributed garbage collection.

References

Appendix

A. Naming conventions, shorthands and lookup functions

We use the naming conventions given in fig. 9, and the shorthands defined in fig. 17.

Lookup functions are defined in fig. 18. Function $P$ returns a type definition for a class identifier $C$ or actor identifier $A$. This contains the fields $F$, constructors $K$, functions $E$, and behaviours $B$ defined for that type. Since classes have no asynchronous behaviour, the last entry in $P(C)$ is empty, i.e. $ε$. Function $F$ returns the identifiers of all fields defined in a type $S$, and function $F$ returns the type of field $f$ in $S$. Function $M$ 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 information returned is a tuple of four components: the receiver type, the names and types of the parameters, the body of the method in the form of a source expression, and the return type. The 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 ref receiver, since the constructor must write to the new object’s fields, and return a ref result, since the new object is initially mutable but also unaified, since the constructor’s reference to the receiver (this) is implicitly discarded when the constructor returns. This allows a constructor that is passed only sendable references as parameters to be embedded in a receive expression, giving the capability iso, which can be aliased as iso, which is a subtype of all other capabilities. This allows constructing an object with any capability. Asynchronous methods always operate on a 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 null. We have chosen to return the receiver to allow chaining method calls.

B. Operational semantics

Definition 4. We call an expression $e$ a redex if it has one of the following forms:

$e ::= z.f | z.f = y | z.m(\gamma) | z.b(\gamma) | S.k(\pi)$

Lemma 1. Uniqueness of contexts. For any expressions $e_1$, $e_2$ and contexts $E_1[\cdot], E_2[\cdot]$, if $E_1[e_2] \equiv E_2[e_2]$ and $e_1$ and $e_2$ are redexes then $E_1[\cdot] \equiv E_2[\cdot]$ and $e_1 \equiv e_2$.

• $ϕ(x) = ϕ \downarrow_2 (x) \downarrow_1$
• $ϕ[x \mapsto v] = (ϕ \downarrow_1, ϕ \downarrow_2 [x \mapsto v], ϕ \downarrow_3)$
• $χ(t, f) = χ(t) \downarrow_2 (f)$
• $χ[\omega, f \mapsto v] = χ[\omega \mapsto (\chi(\omega) \downarrow_1, \chi(\omega) \downarrow_2 [f \mapsto v])]$
• $χ[α, f \mapsto v] = χ[α \mapsto (χ(α) \downarrow_1, χ(α) \downarrow_2 [f \mapsto v], χ(α) \downarrow_3, χ(α) \downarrow_4, χ(α) \downarrow_5)]$
• $χ[α \mapsto (σ, e)] = χ[α \mapsto (χ(α) \downarrow_1, χ(α) \downarrow_2, χ(α) \downarrow_3, σ, e)]$
• $χ[α \mapsto \overline{m}] = χ[α \mapsto (χ(α) \downarrow_1, χ(α) \downarrow_2, \overline{m}, χ(α) \downarrow_4, χ(α) \downarrow_5)]$

Figure 17. Auxiliary definitions

$$P = \text{CT} \cdot \text{AT}$$
$$\text{class} C F K E R ∈ \text{CT} \frac{P(C) = F K E R \in C ∈ P}{P(C) = F K E R \in C ∈ P}$$

$$P = \text{CT} \cdot \text{AT}$$
$$\text{actor} A F K E B ∈ \text{AT} \frac{P(A) = F K E B}{A ∈ P}$$

$$P(S) = F K E B$$
$$\mathcal{F}(S) = \{ f | \text{var} f : T ∈ F \}$$

$$P(S) = F K E B \ \text{var} f : T ∈ F \frac{\mathcal{F}(S, f) = T}{\mathcal{F}(S, f) = T}$$

$$P(C) = F K E B \ (\text{new} k(x : T) \Rightarrow e) ∈ K \frac{\mathcal{M}(C, k) = (\text{Cref}, x : T, e, \text{Cref})}{\mathcal{M}(C, k) = (\text{Cref}, x : T, e, \text{Cref})}$$

$$P(A) = F K E B \ (\text{new} k(x : T) \Rightarrow e) ∈ K \frac{\mathcal{M}(A, k) = (\text{Avar}, x : T, e, \text{A tag})}{\mathcal{M}(A, k) = (\text{Avar}, x : T, e, \text{A tag})}$$

$$P(S) = F K E B \ (\text{fun} \ k(m(x : T) : ET) \Rightarrow e) ∈ \overline{N} \frac{\mathcal{M}(S, m) = (S \nu, x : T, e, ET)}{\mathcal{M}(S, m) = (S \nu, x : T, e, ET)}$$

Figure 18. Lookup functions
∀S ∈ P ⊢ S○

WF-PROGRAM

⊢ Po

P(S) = F K M E

∀var : S ⊢ Ken K

∀M ∈ E ⊢ M ○

∀v ∈ E ⊢ v ○

WF-TYPE

|this → C var, x → T | ⊢ e : C var○

WF-CTOR

C ⊢ new k(x : T) ⇒ e○

|this → S κr, x → T | ⊢ e : ET

WF-SYNC

S ⊢ fun κr, m(x : T) : ET ⇒ e○

Sendable(Γ1)

|this → A var, x → T | ⊢ e : A tag

WF-ATOR

A ⊢ new k(x : T) ⇒ e○

Sendable(Γ1)

|this → A var, x → T | ⊢ e : A tag

WF-ASYNC

A ⊢ be b(x : T) ⇒ e○

Figure 19. Well-formed programs

• z ∈ φ iff z ∈ dom(φ ↓2)
• α ∈ χ iff α ∈ dom(χ)
• Δ ⊢ α ∈ χ iff α ∈ dom(χ)
• Δ ⊢ ε ∈ χ iff ∃ε′ such that Δ ⊢ ε′ ∈ χ and Δ, χ, ε′ ⊢ ε:

• M(φ, χ) = M(χ(φ(this) ↓1, φ ↓1)

Figure 20. Auxiliary well-formedness definitions

C. Type system and well-formed programs

The rules for a well-formed program are presented in fig. 19. The WF-PROGRAM rule indicates a program is well-formed if all types in the program are well-formed. The WF-TYPE rule indicates that a type is well-formed if the types of all of its fields are well-formed, its constructors are well-formed, and its synchronous and asynchronous methods are well-formed. The WF-CTOR, WF-SYNC, and WF-ASYNC rules indicate that a method is well-formed when the body of the method in results in a subtype of the return type of the method. The body of the method is evaluated using an environment composed of the receiver and the method parameters, each mapped to their type, as shown in fig. 6.

Lemma 2. Context lemma.

1. Γ ⊢ E[e] : ET ⇒ ∃ET′ and Γ, y → ET′ ⊢ E[y] : ET and Γ ⊢ e : ET and y /∈ dom(Γ)


∀κ, κ1, κ2 :

1. If κ1 ∼ g κ2, then κ1 ∼ l κ2.
2. If κ1 ⊑ κ2, then
   (a) κ1 ∼ l κ ⇒ κ2 ∼ g κ
   (b) κ1 ∼ g κ ⇒ κ2 ∼ g κ
3. If κ1 ∼ l κ2, and both κ1 ▷ κ and κ2 ▷ κ are defined, then κ1 ▷ κ ∼ l κ2 ▷ κ.
4. If κ1 ∼ g κ2, and both κ1 ▷ κ and κ2 ▷ κ are defined, then κ1 ▷ κ ∼ g κ2 ▷ κ.
5. κ2 ⊑ κ1 ▷ κ2 or κ1 = val or κ1 ▷ κ2 undefined
6. If A(κ1) ⊑ κ2 then
   (a) κ1 ∼ l κ ⇒ κ2 ∼ l κ
   (b) κ1 ∼ g κ ⇒ κ2 ∼ g κ
   (c) A(κ1 ▷ κ) ≤ κ2 ▷ κ
7. If A(κ1) ⊑ κ2 and A(κ2) ⊑ κ4 then
   (a) κ1 ∼ l κ2 ⇒ κ3 ∼ l κ4
   (b) κ1 ∼ g κ2 ⇒ κ3 ∼ g κ4

Proof. By case analysis on κ1 and κ2.

On the other hand, κ1 ⊑ κ2 does not imply that κ ▷ κ2 ≤ κ ▷ κ2. For example, iso ≤ trn, but box ▷ iso = tag ▷ box ▷ trn = box. Similarly, κ1 ⊑ κ2 does not imply that κ1 ▷ κ ≤ κ2 ▷ κ; take iso ▷ trn = tag ▷ trn ▷ trn = trn.

Finally the ▷ operator is not associative, i.e. (κ1 ▷ κ2) ▷ κ3 /∈ κ1 ▷ (κ2 ▷ κ3); take (iso ▷ trn) ▷ val = ∩ ▷ iso ▷ (trn ▷ val) = val.

D. Well-formed runtime configurations


∀κ, κ1,..., κn :

1. If κ1 ≤ κ2 then κ1 ▷ κ ≤ κ2 ▷ κ, or κ2 = val.
2. κ1 ▷ κ2 = val if κ1 ▷ κ2 = val.
3. κ1 ▷ κ2 ≤ κ1 ▷ κ2
4. (...(κ1 ▷ κ2) ▷ κ3...) ▷ κn ≤ (...(κ1 ▷ κ2) ▷ κ3...) ▷ κn
5. (...(κ1 ▷ κ2) ▷ κ3...) ▷ κn = val if (...(κ1 ▷ κ2) ▷ κ3...) ▷ κn = val
6. If κ1 ∼ l κ2 and κ1, κ2 /∈ tag then κ1 ▷ κ ∼ l κ2 ▷ κ or κ1 ▷ κ2 = ref
7. If κ1 ∼ g κ2 and κ1, κ2 /∈ tag then κ1 ▷ κ ∼ g κ2 ▷ κ
8. If A(κ1) ≤ κ2 and κ1 ▷ κ2 = tag then A(κ1 ▷ κ) ≤ κ2 ▷ κ

Lemma 5. Capabilities are preserved along paths.

If Δ, χ, α /⊢ l : κ, p and Δ, χ, α /⊢ l : κ′, p then κ = κ′.


E. Soundness

The property central to any soundness argument is the preservation of the well-formed visibility property, WFV(Δ, χ),
and the well-formed temporaries property $WFT(\Delta, \chi, \alpha, i, e)$ for all expressions and continuations. To study the former, we need properties about the creation of new paths, while for the latter, we need to control the types we assign to the temporaries in each step.

### E.1 New paths

**Lemma 6. Simplification.**

1. $\mathcal{A}(\kappa_1 \phi) \leq \kappa_2$
2. $\kappa_2 \leq \kappa_3$
3. $\kappa_4 < \kappa_2$ or $\kappa_4 < \kappa_3$

Then

4. $\mathcal{A}(\kappa_1 \phi) \leq \kappa_3$
5. $\kappa_4 < \mathcal{A}(\kappa_1 \phi) < \kappa_4 < \kappa_3$

**Proof.** (4) follows from (1) and (2). For (5), if $\kappa_4 < \kappa_3$, done. Otherwise, $\kappa'_2 = \mathcal{A}(\kappa_1 \phi)$. If $\kappa_4 = \text{ref}$, then for all $\kappa_1, \kappa_4 < \kappa'_2$. If $\kappa_4 = \text{trn}$, then $\kappa_2 \in \{\text{iso, trn, val, tag}\} \not\equiv \kappa_3$. If $\kappa'_2 \in \{\text{iso, trn, val, tag}\}$ then $\kappa_1 \in \{\text{iso, trn, val}\}$ and $\text{trn} \triangleleft \kappa'_2$. If $\kappa'_2 = \text{tag}$ then $\kappa_3 = \text{tag}$, which contradicts $\kappa_4 < \kappa_3$. If $\kappa_4 = \text{iso}$, the same holds, except $\kappa_4$ cannot be $\text{trn}$. □

**Definition 5.** Unaliased types can be treated as base types.

$\text{ET} \not\equiv \text{ET}$ if $\text{ET}' \equiv \text{ET}$, or $\text{ET}' = \kappa \kappa_{\phi}$ and $\text{ET} = \kappa$

**Definition 6.** An identifier $z$ is aliased in a runtime expression $e$ iff

$\exists e', f, y, \varphi, n, S$ such that

• $e \equiv E[x = z]$
• $e \equiv E[e'.f = z]$
• $e \equiv E[e'.n(y, z, \varphi)]$
• $e \equiv E[z.n(y)]$
• $e \equiv E[S.k(y, z, \varphi)]$

**Lemma 7. Inversion.**

If $\Gamma \vdash e : \text{ET}$ then

1. If $e \equiv x$ then $\Gamma(x) \equiv \text{ET}$
2. If $e \equiv e_1, f$ then $\exists S, S', \kappa, \kappa_1$ such that $\Gamma \vdash e_1 : S \kappa$ and $\mathcal{F}(S, f) = S' \kappa_1 \kappa'$ and $\text{ET} = S' \kappa \kappa'$
3. If $e \equiv null$ then $\exists S$ such that $S \equiv \text{null} \equiv \text{ET}$
4. If $e \equiv e_1; e_2$ then $\exists \text{ET}_1$ such that $\Gamma \vdash e_1 : \text{ET}_1$ and $\Gamma \vdash e_2 : \text{ET}$
5. If $e \equiv e_1; e_2$ then $\exists S, \kappa, \kappa_1, \phi$ such that $\Gamma(x) = S \kappa$ and $\Gamma \vdash e_1 : S \kappa \phi$ and $\mathcal{F}(S, f) = S \kappa_2 \kappa_3$ and $\text{ET} = S \kappa_2 \kappa_3 \kappa_1 \phi$
6. If $e \equiv e_1, f$ then $\exists S_1, S_2, \kappa_1, \kappa_2, \phi$ such that $\Gamma \vdash e_1 : S_1 \kappa_1$ and $\Gamma \vdash e_2 : S_2 \kappa_2 \phi$ and $\mathcal{F}(S_1, f) = S_2 \kappa_3 \kappa_4$ and $\mathcal{A}(\kappa_2 \phi) \leq \kappa_3$
7. If $e \equiv e_1; e_2$ then $\exists S_0, \kappa_0, \kappa_1, \phi, \bar{\pi}, \text{ET}$ such that $\Gamma \vdash e_0 : S_0 \kappa_0 \phi$ and $\mathcal{A}(\kappa_0 \phi) \leq \kappa_0$ and $\mathcal{M}(S_0, m) = (S_0 \kappa_0, \bar{\pi}, \text{ET}, \text{ET}')$ and $\Gamma \vdash e_0 : \text{ET}_1$ and $\mathcal{A}(\text{ET}_1) \leq \text{T}_1$ and $\text{ET}' \equiv \text{ET}

8. If $e \equiv e_0, b(v)$ then $\exists A, \kappa_0, \kappa_3, \phi, \mathcal{T}$ such that $\Gamma \vdash e_0 : A \kappa_0 \phi$ and $\mathcal{A}(A \kappa_0 \phi) \leq \kappa_0$ and $\mathcal{M}(A, b) = (A \text{ref}, \bar{\pi}, \text{ET}, A \tag)$ and sendable($\text{T}_1$) and $\Gamma \vdash e_1 : \text{ET}_1$ and $\mathcal{A}(\text{ET}_1) \leq \text{T}_1$ and $\text{ET} = \text{ET}

9. If $e \equiv C.k(\bar{v})$ then $\exists \text{ET}, \bar{v}$ such that $\mathcal{M}(C, k) = (C \text{ref}, \bar{\pi}, \text{ET}, A \tag)$ and $\Gamma \vdash e_1 : \text{ET}_1$ and $\mathcal{A}(\text{ET}_1) \leq \text{T}_1$ and $C \text{ref} \equiv \text{ET}

10. If $e \equiv A.k(\bar{v})$ then $\exists \text{ET}, \bar{v}$ such that $\mathcal{A}(A, k) = (A \text{ref}, \bar{\pi}, \text{ET}, A \tag)$ and sendable($\text{T}_1$) and $\Gamma \vdash e_1 : \text{ET}_1$ and $\mathcal{A}(\text{ET}_1) \leq \text{T}_1$ and $\text{ET} = \text{ET}

11. If $e \equiv e'$ then $\exists \text{ET}'$ such that $\Gamma' = \Gamma \setminus \{ x \mid \neg \text{sendable}(\Gamma(x)) \}$ and $\Gamma' \vdash e' : \text{ET}'$ and $\mathcal{R}(\text{ET}') \equiv \text{ET}'$

**Proof.** By induction on the typing of $\Gamma \vdash e : \text{ET}$. For case 6 (field assignment), apply lemma 6. □

**Lemma 8. Temporaries and variables with unique capabilities are unique.**

If

1. $WFV(\Delta, \chi)$
2. $\chi(\alpha, i, z) = \chi(\alpha', i', z') = \iota$
3. $\Delta(\alpha, i, z) = S \kappa \phi$
4. $\kappa \equiv \{\text{iso, trn}\}
5. $\Delta, \chi, \alpha' \vdash \iota : \kappa', (i', z')$

Then $\alpha = \alpha'$ and either $\kappa \equiv \alpha'$ or $\iota = (i', z')$. □

**Proof.** Assume that $\alpha \neq \alpha'$. Then, by $WFV.1$, $\kappa \equiv \alpha' \kappa'$. This implies $\kappa' = \text{tag}$, which contradicts 5. Therefore, $\alpha = \alpha'$ and $\Delta, \chi, \alpha' \vdash \iota : \kappa, (i, z)$. If $\text{Stable}(\Delta, \alpha, (i, z))$ then by $WFV.2$, either $\kappa \equiv \alpha' \kappa$ (done) or $\chi(\alpha, i, z) = (i', z')$ (done). If $\neg \text{Stable}(\Delta, \alpha, (i, z))$ then $z = \tau$ and by $WFV.4$ either $(i, z) = (i', z')$ (done) or $\exists e_i, k', \bar{v}, \bar{\pi}$ such that $\kappa \equiv k'$ and $\kappa' \equiv \{\text{iso, trn}\}$ and $(i', z') = p' \cdot \bar{v}$ and $\Delta, \chi, \alpha' \vdash \iota' : \kappa', p'$ and $\Delta, \chi, \alpha' \vdash \iota : \kappa, (0, \text{this}) \cdot \bar{v}$ and $\iota = \iota'$. This gives us $\Delta, \chi, \alpha' \vdash \iota : \kappa, (0, \text{this})$, which by the definition of visibility gives us $\kappa = \text{ref}$, which contradicts (4) (done). □

**Lemma 9. Isolation in well-formed visibility.**

If

1. $WFV(\Delta, \chi)$
2. $\Delta(\alpha, i, t) = S \kappa \phi$ and $\chi(\alpha, i, t) = \iota$ and $\kappa \equiv \{\text{iso, trn}\}$
3. $\Delta, \chi, \alpha' \equiv \iota : \kappa', p$

Then

4. If $\kappa \phi \equiv \text{iso}$ then $\alpha = \alpha'$ and $p = (i, t)$.
5. If $\kappa \phi \equiv \text{trn}$ then $\alpha = \alpha'$ and either $p = (i, t)$ or $\kappa' = \text{box}$.  


6. If $\kappa \phi = \text{iso}$ then $\alpha = \alpha'$ and either $p = (i, t)$ or $\exists \iota', \kappa', \pi', \tau'$ such that $\kappa \leq \kappa''$ and $\kappa'' \in \{\text{iso, trn}\}$ and $p = p' : \tau$ and $\Delta, \chi, \alpha \vdash \iota' : \kappa', \pi' \text{ and } \Delta, \chi, \iota' \vdash i : \text{iso, trn}$.

7. If $\kappa \phi = \text{trn}$ then $\alpha = \alpha'$ and either $p = (i, t)$ or $\kappa' = \text{box}$ or $\exists \iota', \kappa', \pi', \tau'$ such that $p = p' : \tau$ and $\Delta, \chi, \alpha \vdash \iota' : \text{trn, } \pi' \text{ and } \Delta, \chi, \iota' \vdash i : \text{trn, trn}$.

Proof. (4) and (5) follow from lemma 8. (5) and (6) follow from lemma 8 and $WFV.A$.

Lemma 10. Aliasing and replaceability.

If
1. $\Gamma \vdash e : ET$ and $z$ is aliased in $e$
2. $z$ does not appear more than once in $e$
3. $\Gamma(z)$ is not unaliased
4. $\Gamma' = \Gamma[z \mapsto A(\Gamma(z))]$

Then $\Gamma' \vdash e : ET$.

Proof. By induction over the structure of $e$. We apply lemma 7. Moreover, we use the fact that $\forall \kappa, A(\Delta(\kappa)) = \Delta(\kappa)$. The base cases are expressions that can alias $z$.

- If $e \equiv x = z$ then, by lemma 7, we obtain $\Gamma(x) = S \kappa$ and $\Gamma(z) = S \kappa' \phi$ and $\phi \neq o$ and $\Delta(\kappa') \leq \kappa$. Therefore, we have $\Delta(\Delta(\kappa')) \leq \kappa$ and so $\Gamma' \vdash x = z : ET$.

- If $e \equiv e'.f = z$ then, by lemma 7, we obtain $\Gamma' \vdash e' : S \kappa$ and $\Delta(S.f) = S' \kappa'$ and $\Gamma(z) = S' \kappa'' \phi$ and $\phi \neq o$ and $\Delta(\kappa'') \leq \kappa'$. Therefore, we have $\Delta(\Delta(\kappa'')) \leq \kappa'$ and so $\Gamma' \vdash e'.f = z : ET$.

- If $e \equiv e'.n(y, z, \pi)$ then, by lemma 7, we obtain $\Gamma \vdash e' : S \kappa \text{ and } \Delta(S.n) = (\_ \pi : S \kappa, \_ \pi : S \kappa, \_ \pi : S \kappa) \text{ and } \Gamma(z) = S \kappa' \phi$ and $\phi \neq o$ and $\Delta(\kappa') \leq \kappa$. Therefore, we have $\Delta(\Delta(\kappa')) \leq \kappa$ and so $\Gamma' \vdash e'.n(y, z, \pi) : ET$.

- If $e \equiv S.k(\_ \pi, z, \pi)$ then, by lemma 7, we obtain $\Delta(\Delta(\kappa')) \leq \kappa$ and so $\Gamma' \vdash S.k(\_ \pi, z, \pi) : ET$.

For the inductive step, if $e \equiv E[e']$ and $z$ is aliased in $e$, then, by lemma 2, we obtain that $E[T'] : y \notin \Gamma$ such that $\Gamma \vdash e' : ET'$ and $\Gamma[y \mapsto ET'] \vdash E[y] : ET$, and so $\Gamma' \vdash e' : ET'$. Therefore, by lemma 2, we obtain $\Gamma' \vdash E[e'] : ET$.


If
1. $z$ appears once in expression $e$
2. $z$ is not aliased in $e$

Then $\exists E'$ such that
3. $e \equiv E'[z.f]$, or
4. $e \equiv E'[z.f = e']$, or
5. $e \equiv E'[\text{recover } z]$, or


Lemma 12. If $\Gamma, x : T_1 \vdash e : ET_1$ and $\Delta(T_2) \leq T_1$ then $\exists \Delta e, \Gamma, x : T_2 \vdash e : ET_2$ and $ET_2 = ET_1$ or $\Delta(ET_2) \leq ET_1$.

Proof. By structural induction on the typing and lemma 3.
2. \[ \Delta(\alpha, i, z) = S \kappa \phi \text{ and } \Delta(\alpha, i, z') = S' \kappa' \phi' \]

3. \[ \chi(i, f) = \iota'' \text{ and } F(S, f) = S'' \kappa'' \]

4. \[ \chi' = \chi((i, f) \mapsto \iota', (\alpha, i, t) \mapsto \iota'') \]

5. \[ T = \bot \text{ if } z = t', S \kappa \text{ otherwise} \]

6. \[ T' = \bot \text{ if } z' = t'', S' \kappa' \text{ otherwise} \]

7. \[ \Delta' = \Delta(\alpha, i, z) \mapsto T, (\alpha, i, z') \mapsto T', (\alpha, i, t) \mapsto U(S' \kappa \kappa' \kappa'') \]

8. \[ T(\Delta(\alpha, i)) \subseteq \{ z, z' \} \]

Then

9. \[ \forall \alpha', i'', \kappa'', p' \text{ if } \Delta', \chi', \alpha' \vdash i'' : \kappa'', p' \text{ then} \]

   \[ \begin{array}{l}
   (a) \Delta, \chi, \alpha' \vdash i'' : \kappa'', p' \\
   (b) \alpha' = \alpha \text{ and } \exists \bar{\kappa}, \bar{\kappa} \text{ such that} \\
   \quad \text{i. } p = (i, z) \cdot \bar{\kappa} \text{ and } \Delta, \chi, \alpha' \vdash i'' : \kappa'' \cdot \bar{\kappa}, \text{ or} \\
   \quad \text{ii. } \kappa'''' = U(\kappa \kappa) \bar{\kappa} \text{ and } \alpha' \vdash i'' : \kappa'''' \cdot \bar{\kappa}, i, \bar{\kappa}, \bar{\kappa}' \text{ and } \exists \kappa'''' \cdot \bar{\kappa} \text{ such that} \\
   \quad \text{iii. } \exists \kappa'''' \cdot \bar{\kappa} \text{ such that} \end{array} \]

10. If \( A(\kappa'') \leq \kappa'''' \text{ and } (\kappa \kappa) \kappa'''' \) and \( \text{WFV}(\Delta, \chi) \) then \( \text{WFV}(\Delta', \chi') \) and \( T(\Delta'(\alpha, i)) = \emptyset \)

**Lemma 16.** New paths through message passing.

1. \[ \chi(\alpha, i, z) = \iota \text{ and } \Delta(\alpha, i, z) = S \kappa \phi \]

2. \[ \chi' = \chi((\alpha', \iota, j, x) \mapsto \iota) \]

3. \[ T = \bot \text{ if } z = t, S \kappa \text{ otherwise} \]

4. \[ \Delta' = \Delta(\alpha, i, z) \mapsto T, (\alpha, i, \iota, j, x) \mapsto S \kappa' \]

5. \[ T(\Delta(\alpha, i)) \subseteq \{ z \} \]

Then

6. \[ \forall \alpha'', i'', \kappa'', p \text{ if } \Delta', \chi', \alpha'' \vdash i'' : \kappa'', p \text{ then} \]

   \[ \begin{array}{l}
   (a) \Delta, \chi, \alpha'' \vdash i'' : \kappa'', p' \\
   (b) \alpha'' = \alpha' \text{ and } \exists \bar{\kappa}, \bar{\kappa} \text{ such that} \\
   \quad \text{i. } p = (i, z) \cdot \bar{\kappa} \\
   \quad \text{ii. } \kappa'''' = U(\kappa \kappa) \bar{\kappa} \text{ and } \Delta, \chi, \alpha'' \vdash i'' : \kappa'''' \cdot \bar{\kappa}, \text{ or} \\
   \quad \text{iii. } \exists \kappa'''' \cdot \bar{\kappa} \text{ such that} \end{array} \]

7. If \( A(\kappa) \leq \kappa'' \text{ and sendable}(\kappa'') \) and \( \text{WFV}(\Delta, \chi) \) then \( \text{WFV}(\Delta', \chi') \) and \( T(\Delta'(\alpha, i)) = \emptyset \)

**E.2 Preservation of well-formedness**

**Lemma 17.** Type preservation on same frame.

For all heaps \( \chi \), actors \( \alpha \), global type environments \( \varphi \), stacks \( \sigma \) and expressions \( e \), if

1. \[ \chi(\alpha) = (\iota, 1, \alpha, \varphi, E[e]) \text{ and } |\varphi| = i - 1 \]

2. \[ \chi, \alpha \vdash \varphi, e \mapsto \chi''', \alpha \vdash \varphi', e' \]

3. \[ \chi' = \chi''''(\alpha \mapsto (\alpha \cdot \varphi', E[e''])) \]

4. \[ \Delta(\alpha, i) \vdash e : ET \]

5. \[ \Delta \vdash \chi'''' \]

Then \( \exists \Delta' \text{ such that} \)

1. \[ \Delta'(\alpha, i) \vdash e' : ET \]
--chunk N log2 of the chunk size.
 Defaults to 10.
--actors N log2 of the actor count.
 Defaults to 2.
***
}
return false
end

env.out.print("*logtable: " + logtable.string() +
\"iterate: " + iterate.string() +
\"logchunk: " + logchunk.string() +
\"logactors: " + logactors.string() +
"
)

actor Main
let _env: Env
let _config: Config = Config
var _updates: U64 = 0
var _confirm: U64 = 0
let _start: U64
var _actors: Array[Updater] val
new create(env: Env) =>
  _env = env
if _config(env) then
  let actor_count = 1 << _config.logactors
  let loglocal = _config.logtable - _config.logactors
  let chunk_size = 1 << _config.logchunk
  let chunk_iterate = chunk_size * _config.iterate
  _updates = chunk_iterate * actor_count
  _confirm = actor_count
  var updaters = recover Array[Updater](actor_count)
end
for i in Range[U64](0, actor_count) do
  updaters.push(Updater(this, actor_count, i, _loglocal, chunk_size, chunk_iterate * i))
end
_actors = consume updaters
_start = Time.nanos()
try
  for a in _actors.values() do
    a.start(_actors, _config.iterate)
  end
end
else
  _start = 0
  _actors = recover Array[Updater] end
end
be done() =>
if (_confirm = _confirm - 1) == 1 then
  try
    for a in _actors.values() do
      a.done()
  end
end
be confirm() =>
  _confirm = _confirm + 1
if _confirm == _actors.size() then
  let elapsed = (Time.nanos() - _start).f64()
  let guts = _updates.f64() / elapsed
  _env.out.print("Time: " + (elapsed / 1e9).string() +
  \"GUTS: " + guts.string() +
  )
to(_output.size()).receive(consume data)
else
  _reuse.push(consume data)
end
until _output.size() == 0 end
end
if iterate > 1 then
  apply(iterate - 1)
else
  _main.done()
end
be receive(data: Array[U64] iso) =>
try
  for i in Range(0, data.size()) do
    let datum = data(i)
    var j = (datum >> _loglocal) and _mask
    _table(j) = _table(j) xor datum
  end
  data.clear()
  _reuse.push(consume data)
end
be done() =>
  _main.confirm()

primitive PolyRand
fun apply(prev: U64): U64 =>
  (prev << 1) xor if prev.i64() < 0 then _poly() else 0 end
fun seed(from: U64): U64 =>
  var n = from % _period()
  if n == 0 then
    return 1
  end
  var m2 = Array[U64].undefined(64)
  var temp = U64(1)
  try
    for i in Range(0, 64) do
      m2(i) = temp
      temp = this(temp)
      temp = this(temp)
    end
  end
  var i: U64 = 64 - n.clz()
  var r = U64(2)
  try
    while i > 0 do
      temp = 0
      for j in Range(0, 64) do
        if (r >> j) and 1 != 0 then
          temp = temp xor m2(j)
        end
      end
      r = temp
      i = i - 1
      if (n >> i) and 1 != 0 then
        r = this(r)
      end
    end
    r
  end
  fun _poly(): U64 => 7
  fun _period(): U64 => 1317624576693539401