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 `trn` 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*, `trn`.
- A capability, `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, `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 `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 `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 `box` for immutability allows a program to enforce read-only behaviour, similar to `const` in C/C++. For example:

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

Note that the receiver capability is specified after the keyword `fun`. In `size1`, by indicating that the receiver has `box` capability, we can be certain that `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 `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 other reference if that reference is held by the same actor. The `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 `size2`, by indicating that the receiver has `val` 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.

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

We use the keyword `actor` to indicate a class that can have *behaviours* (asynchronous methods), and we use the

	Deny global read/write aliases	Deny global write aliases	Allow all global aliases
Deny local read/write aliases	*Isolated (iso)*		
Deny local write aliases	Transition (trn)	*Value (val)*	
Allow all local aliases	Reference (ref)	Box (box)	*Tag (tag)*
	(Mutable)	(Immutable)	(Opaque)

Table 1. Capability matrix. Capabilities in *italics* are sendable.

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.

```
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 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.

```
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 use a *destructive read*.

```
actor Dataflow
  be step(list: List iso, flow: Dataflow tag) =>
    next.step(list) // Not allowed
    next.step(list = null)
```

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 $\circ$, in order to type values for which an alias has been removed. Here, the destructive read produces a List iso$\circ$ 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 outlive 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.

```
actor Dataflow
  be step(list1: List iso, list2: List iso,
      next: Dataflow tag) =>
    list1.next = (list2 = null)
    next.step(list1 = null)
```

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:

```
actor Dataflow
  fun ref append(list1: List iso,
      list2: List ref) =>
    list1.next = list2 // Not allowed
```

Even if list1.next had the type List ref, 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 capability in the future, in order to share it with another actor.

```
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))

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 trn`, 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, eg. `ref` $\sim_\ell$ `ref` but `ref` $\not\sim_g$ `ref`. Both relations are symmetric.

In fig. 1, we show a diagrammatic representation of a heap χ_0 which contains actors α_1 and α_2, and objects $\iota_{10}...\iota_{19}$. The top rectangles indicate stack frames, for example $\chi_0(\alpha_1) = (\_, \_, \alpha_1 \cdot \varphi_1 \cdot \varphi_2, \_)$ and $\varphi_1(\texttt{this}) = \iota_{10}$

[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.

$\kappa \sim \kappa'$	κ'					
κ	iso	trn	ref	val	box	tag
iso						ℓ, g
trn					ℓ	ℓ, g
ref			ℓ		ℓ	ℓ, g
val				ℓ, g	ℓ, g	ℓ, g
box		ℓ	ℓ	ℓ, g	ℓ, g	ℓ, g
tag	ℓ, g	ℓ, g	ℓ, g	ℓ, g	ℓ, g	ℓ, g

Table 2. Compatible capabilities.

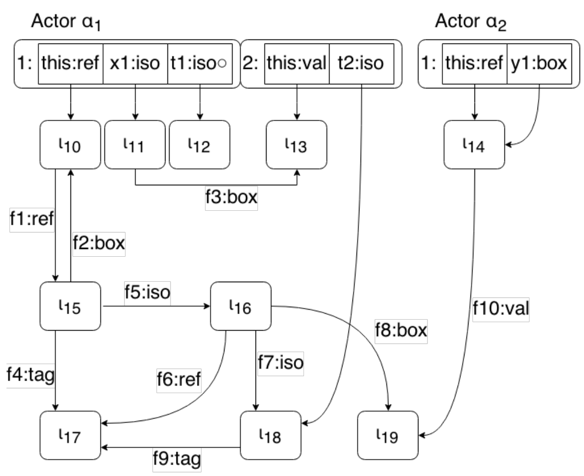

Figure 1. A representation of part of a heap.

and $\varphi_2(\texttt{t}_2) = \iota_{18}$. The objects are in rounded boxes, and the annotated arrows indicate the contents of their fields, e.g. $\chi_0(\iota_{14}, \texttt{f10}) = \iota_{19}$. The annotations next to the field identifiers (`ref`, `val`, etc.) give types to the variables. Note that $\alpha_1 = \iota_{10}$ and $\alpha_2 = \iota_{14}$.

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 α_1 and the path `this.f10` at the first frame of actor α_2 are aliases, as they both reach object ι_{19}. The first path sees ι_{19} as `tag`, while the second sees it as `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 ι_{15}, such that it pointed to ι_{19}, the resulting capabilities would not be globally compatible.

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

Γ	$\in$	*Env*	$=$	*LocalID* $\to$ *ExtType*
Δ	$\in$	*GlobalEnv*	$=$	$(\mathit{ActorAddr} \times \mathit{Integer}) \to \mathit{Env}$
p	$\in$	*Path*	$=$	$(\mathit{Integer} \times \mathit{LocalID}) \cdot \overline{\mathit{FieldID}}$

Figure 2. Global environments and paths.

- $\Delta, \chi, \iota \vdash \iota : \texttt{ref}, (0, \texttt{this})$
- $\Delta, \chi, \alpha \vdash \iota : \kappa, (i, \texttt{z})$ iff $\chi(\alpha, (i \cdot \texttt{z})) = \iota$ and $\Delta(\alpha, i, \texttt{z}) = \texttt{S}\,\kappa\,\phi$ and $\kappa \neq \texttt{tag}$
- $\Delta, \chi, \iota \vdash \iota' : \kappa \blacktriangleright \kappa', p \cdot \texttt{f}$ iff $\Delta, \chi, \iota \vdash \iota'' : \kappa, p$ and $\chi(\iota'', \texttt{f}) = \iota'$ and $\mathcal{F}(\chi(\iota'') \downarrow_1, \texttt{f}) = \texttt{S}\,\kappa'$ and $\kappa \blacktriangleright \kappa' \neq \texttt{tag}$
- $\Delta, \chi, \iota \vdash \iota' : \kappa$ iff $\exists p$ such that $\Delta, \chi, \iota \vdash \iota' : \kappa, p$

Figure 3. Visibility.

- $\kappa \blacktriangleright \kappa' = \begin{cases} \kappa' & \text{if } \kappa \in \{\texttt{iso}, \texttt{trn}, \texttt{ref}\} \\ \texttt{val} & \text{if } \kappa = \texttt{val} \wedge \kappa' = \texttt{val} \\ \texttt{box} & \text{if } \kappa = \texttt{box} \wedge \kappa' \notin \{\texttt{iso}, \texttt{val}, \texttt{tag}\} \\ \texttt{tag} & \text{otherwise} \end{cases}$
- $\chi, \alpha \vdash p_1 \cdot \texttt{f} \sim p_2 \cdot \texttt{f}$ iff $\chi(\alpha, p_1) = \chi(\alpha, p_2)$
- $\chi, \alpha \vdash (i, \texttt{z}) \sim (i, \texttt{z})$
- $\chi, \alpha \vdash \iota \in p$ iff $\exists p', \bar{\texttt{f}}$ such that $p = p'.\bar{\texttt{f}}$ and $\chi(\alpha, p') = \iota$
- $\chi(\alpha, (i, \texttt{z}) \cdot \overline{\texttt{f}}) = \chi(\varphi_i(\texttt{z}), \overline{\texttt{f}})$ where $\chi(\alpha) \downarrow_4 = \alpha \cdot \overline{\varphi}$
- $\chi(\alpha, (-i, \texttt{x}_\texttt{j}) \cdot \overline{\texttt{f}}) = \chi(v_j, \overline{\texttt{f}})$ where $\chi(\alpha) \downarrow_3 = \overline{\mu}$ and $\mu_i = (\_, \overline{v})$
- $\mathit{Stable}(\Delta, \alpha, (i, \texttt{z}) \cdot \overline{\texttt{f}})$ iff $\Delta(\alpha, i, \texttt{z}) \notin \{\texttt{iso}, \texttt{trn}\}$ or $\texttt{z} \neq \texttt{t}$

Figure 4. Topological properties of paths.

1. Paths p and global environments Δ, which give types to the local variables and temporaries in each frame or message, as defined in fig. 2.
2. Path visibility $\Delta, \chi, \iota \vdash \iota' : \kappa, p$, which says that the object or actor ι sees the object or actor ι' as capability κ through path p, as defined in fig. 3.
3. Topological properties of paths, as defined in fig. 4.

Environments Γ map variables (i.e. local variables or temporaries) to extended types and global environments, Δ map actor addresses and integers to environments. In fig. 1, we indicate the types assigned to local variables through the annotations. Thus, we have an implicit global environment Δ_0, such that $\Delta_0, \chi_0, \alpha_1 \vdash \iota_{10} : \texttt{ref}, (1, \texttt{this})$, and $\Delta_0, \chi_0, \alpha_2 \vdash \iota_{19} : \texttt{val}, (1, \texttt{this}) \cdot \texttt{f10}$.

To define path visibility, we need the notion of deep viewpoint adaptation $\kappa \blacktriangleright \kappa'$, which combines two capabilities as

$\mathit{WFV}(\Delta, \chi)$ *iff*
$\forall \alpha, \alpha', \iota, \iota' \in \chi. \forall \kappa, \kappa', p, p', \texttt{t}$ where $\mathit{Stable}(\Delta, \alpha, p)$ and $\mathit{Stable}(\Delta, \alpha, p')$

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) $\kappa \sim_\ell \kappa'$
3. If $\Delta, \chi, \alpha \vdash \iota : \kappa$ and $\Delta, \chi, \alpha \vdash \iota' : \kappa', p'$ and $\Delta, \chi, \iota \vdash \iota' : \kappa''$ and $\kappa \in \{\texttt{iso}, \texttt{trn}\}$ then
   (a) $\chi, \alpha \vdash \iota \in p'$ or
   (b) $\kappa'' \in \{\texttt{val}, \texttt{box}\}$ and $\kappa' \sim_g \texttt{val}$ or
   (c) $\kappa'' \in \{\texttt{iso}, \texttt{trn}, \texttt{ref}\}$ and $\kappa \sim_\ell \kappa'$
4. If $\Delta(\alpha, i, \texttt{t}) = \texttt{S}\,\kappa$ and $\kappa \in \{\texttt{iso}, \texttt{trn}\}$ and $\chi(\alpha, i, \texttt{t}) = \chi(\alpha, p_1) = \iota$ then
   (a) $p_1 = (i, \texttt{t})$ or
   (b) $\exists \iota', \kappa', p_2, \overline{\texttt{f}}$ such that
      i. $\kappa \leq \kappa'$
      ii. $\kappa' \in \{\texttt{iso}, \texttt{trn}\}$
      iii. $p_1 = p_2 \cdot \overline{\texttt{f}}$
      iv. $\Delta, \chi, \alpha \vdash \iota' : \kappa', p_2$
      v. $\Delta, \chi, \iota' \vdash \iota : \kappa, \overline{\texttt{f}}$

Figure 5. Well-formed visibility.

given in fig. 4. The definition ensures that $\kappa \blacktriangleright \kappa' = \kappa'$ if κ is writeable (deep mutability), $\kappa \blacktriangleright \kappa' = \texttt{val}$ if either κ or κ' is val (deep immutability) and $\texttt{box} \blacktriangleright \kappa' = \texttt{box}$ unless $\kappa' \in \{\texttt{iso}, \texttt{val}, \texttt{tag}\}$. For example, $\texttt{iso} \blacktriangleright \texttt{ref} = \texttt{ref}$.

The rules in fig. 3 say that an address sees itself as ref, an actor sees a stack identifier as the capability provided by Δ, 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:

- $\Delta_0, \chi_0, \alpha_1 \vdash \iota_{10} : \texttt{ref}, (1, \texttt{this})$, but also $\Delta_0, \chi_0, \alpha_1 \vdash \iota_{10} : \texttt{box}, (1, \texttt{this}) \cdot \texttt{f1} \cdot \texttt{f2}$.
- $\Delta_0, \chi_0, \alpha_2 \vdash \iota_{19} : \texttt{val}, (1, \texttt{this}) \cdot \texttt{f10}$, but also $\Delta_0, \chi_0, \alpha_1 \vdash \iota_{19} : \texttt{tag}, (1, \texttt{this}) \cdot \texttt{f1} \cdot \texttt{f5} \cdot \texttt{f8}$.

In fig. 4, two paths are compatible if they share the last step or they are the same identifier with no fields, an address ι is in a path if some prefix of the path points to ι, and a path is stable, $\mathit{Stable}(\Delta, \alpha, p)$, if its initial identifier is not a unique temporary. For example, $\chi_0, \alpha_2 \vdash (1, \texttt{this}) \cdot \texttt{f10} \sim (1, \texttt{y1}) \cdot \texttt{f10}$. Also, $\mathit{Stable}(\Delta_0, \alpha_1, (1, \texttt{this}) \cdot \texttt{f1} \cdot \texttt{f4})$ and $\neg \mathit{Stable}(\Delta_0, \alpha_1, (2, \texttt{t2}) \cdot \texttt{f9})$, 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 `iso` and `trn`. Given α that sees ι as some unique κ and sees ι' as κ' via some stable p', and given that ι sees ι' as κ'':
   (a) ι' must be contained by ι, or
   (b) neither ι nor α can write to ι', or
   (c) ι can write to ι' and α sees ι' as locally compatible with κ.
4. Properties of unique temporary identifiers. Given `t` that points to ι , some other path p_1 to the same ι must be either:
   (a) also `t` or
   (b) that path p_1 must have a prefix p_2 that sees some ι' with a unique capability κ' less precise than κ and ι' must see ι as κ.

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 $WFV.1-3$ 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 $(1,\mathtt{this})\cdot\mathtt{f1}\cdot\mathtt{f5}\cdot\mathtt{f8}$ from α_1 and $(1,\mathtt{this})\cdot\mathtt{f10}$ from α_2 satisfy $WFV.1$, while $(1,\mathtt{x1})\cdot\mathtt{f4}$ from α_1 and $(2,\mathtt{this})$ from α_1 satisfy $WFV.2$ and $WFV.3$. On the other hand, the temporary $(2,\mathtt{t2})$ is not stable, and therefore not restricted by $WFV.2$ or $WFV.3$, but does adhere to $WFV.4$. Finally, the assignment $\mathtt{this.f1.f5.f6} = \mathtt{this.f1.f5.f7}$ would break $WFV.2$, while setting `t2` to point to ι_{15} would break $WFV.4$.

4. Type system

The type system has the format $\Gamma \vdash \mathtt{e} : \mathtt{ET}$ 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.
4. 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 capability is unique (`iso` or `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 $\mathtt{this.f1.f5} = \mathtt{null}$ in the first frame of actor α_1 in fig. 1 would return a new temporary which would be the unique reference to ι_{16}. The type of this expression would be $\mathtt{S}\ \mathtt{iso}\circ$ for some S. Because unaliasing is of importance only when the underlying capability is `iso`, `trn` or `ref`, we have defined the unaliasing operation $\mathcal{U}$, 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-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 $\Gamma \vdash \mathtt{e} : \mathtt{ET}$ and the *aliased* type judgment $\Gamma \vdash_{\mathcal{A}} \mathtt{e} : \mathtt{ET}$. For example, when assigning an expression `e` to a variable `x`, the right-hand side is typed in the judgment $\vdash_{\mathcal{A}}$ (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 $\Gamma \vdash_{\mathcal{A}} \mathtt{e} : \mathtt{ET}$ is defined in terms of the unaliased type judgment $\Gamma \vdash \mathtt{e} : \mathtt{ET}'$, where ET has to be a super-type of the aliased version of $\mathtt{ET}'$, i.e. $\mathcal{A}(\mathtt{ET}') \leq \mathtt{ET}$. The operation $\mathcal{A}(\mathtt{ET})$ gives the type that an alias of ET would have. When aliasing an unaliased type there is no previous alias to consider, and therefore $\mathcal{A}(\mathtt{S}\,\kappa\circ) = \mathtt{S}\,\kappa$. For other types, the result must be the minimal super-type of the underlying type which is locally compatible with it, i.e. $\mathcal{A}(\mathtt{S}\,\kappa) = \mathtt{S}\,\kappa'$ where $\kappa' \leq \mathcal{A}(\kappa')$ and $\mathcal{A}(\kappa') \sim_{\ell} \kappa'$.

Definition 1. Aliasing and unaliasing.

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

$$\frac{\mathtt{x} \in \Gamma}{\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x})}\ \textsc{T-Local} \qquad \frac{\Gamma \vdash \mathtt{e} : \mathtt{S}\,\kappa \quad \mathcal{F}(\mathtt{S},\mathtt{f}) = \mathtt{S}'\,\kappa'}{\Gamma \vdash \mathtt{e.f} : \mathtt{S}'\,\kappa \triangleright \kappa'}\ \textsc{T-Fld}$$

$$\frac{\mathtt{S} \in \mathtt{P}}{\Gamma \vdash \mathtt{null} : \mathtt{S}\,\mathtt{iso}\circ}\ \textsc{T-Null} \qquad \frac{\Gamma \vdash \mathtt{e} : \mathtt{ET} \quad \Gamma \vdash \mathtt{e}' : \mathtt{ET}'}{\Gamma \vdash \mathtt{e}; \mathtt{e}' : \mathtt{ET}'}\ \textsc{T-Seq}$$

$$\frac{\Gamma(\mathtt{x}) = \mathtt{S}\,\kappa \quad \Gamma \vdash_{\mathcal{A}} \mathtt{e} : \mathtt{S}\,\kappa}{\Gamma \vdash \mathtt{x} = \mathtt{e} : \mathcal{U}(\mathtt{S}\,\kappa)}\ \textsc{T-AsnLocal} \qquad \frac{\begin{array}{c}\Gamma \vdash \mathtt{e} : \mathtt{S}\,\kappa \quad \Gamma \vdash_{\mathcal{A}} \mathtt{e}' : \mathtt{S}'\,\kappa' \\ \mathcal{F}(\mathtt{S},\mathtt{f}) = \mathtt{S}'\,\kappa'' \quad \kappa' \leq \kappa'' \quad \vdash \kappa \triangleleft \kappa' \vee \vdash \kappa \triangleleft \kappa''\end{array}}{\Gamma \vdash \mathtt{e.f} = \mathtt{e}' : \mathcal{U}(\mathtt{S}'\,\kappa \triangleright \kappa'')}\ \textsc{T-AsnFld}$$

$$\frac{\begin{array}{c}\mathcal{M}(\mathtt{S},\mathtt{m}) = (\mathtt{T}, \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{e}, \mathtt{ET}) \\ \Gamma \vdash_{\mathcal{A}} \mathtt{e} : \mathtt{T} \quad \Gamma \vdash_{\mathcal{A}} \mathtt{e_i} : \mathtt{T_i}\end{array}}{\Gamma \vdash \mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{ET}}\ \textsc{T-Sync} \qquad \frac{\begin{array}{c}\mathcal{M}(\mathtt{A},\mathtt{b}) = (\mathtt{A}\,\mathtt{ref}, \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{e}, \mathtt{A}\,\mathtt{tag}) \\ \Gamma \vdash_{\mathcal{A}} \mathtt{e} : \mathtt{A}\,\mathtt{tag} \quad \Gamma \vdash_{\mathcal{A}} \mathtt{e_i} : \mathtt{T_i}\end{array}}{\Gamma \vdash \mathtt{e.b}(\overline{\mathtt{e}}) : \mathtt{A}\,\mathtt{tag}}\ \textsc{T-Async}$$

$$\frac{\begin{array}{c}\mathcal{M}(\mathtt{C},\mathtt{k}) = (\mathtt{C}\,\mathtt{ref}, \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{e}, \mathtt{C}\,\mathtt{ref}\circ) \\ \Gamma \vdash_{\mathcal{A}} \mathtt{e_i} : \mathtt{T_i}\end{array}}{\Gamma \vdash \mathtt{C.k}(\overline{\mathtt{e}}) : \mathtt{C}\,\mathtt{ref}\circ}\ \textsc{T-Ctor} \qquad \frac{\begin{array}{c}\mathcal{M}(\mathtt{A},\mathtt{k}) = (\mathtt{A}\,\mathtt{ref}, \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{e}, \mathtt{A}\,\mathtt{tag}) \\ \Gamma \vdash_{\mathcal{A}} \mathtt{e_i} : \mathtt{T_i}\end{array}}{\Gamma \vdash \mathtt{A.k}(\overline{\mathtt{e}}) : \mathtt{A}\,\mathtt{tag}}\ \textsc{T-Ator}$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{ET}' \quad \mathcal{A}(\mathtt{ET}') \leq \mathtt{T}}{\Gamma \vdash_{\mathcal{A}} \mathtt{e} : \mathtt{T}}\ \textsc{T-Alias} \qquad \frac{\Gamma \backslash \{\mathtt{x} \mid \neg \mathit{Sendable}(\Gamma(\mathtt{x}))\} \vdash \mathtt{e} : \mathtt{ET}}{\Gamma \vdash \mathtt{recover}\,\mathtt{e} : \mathcal{R}(\mathtt{ET})}\ \textsc{T-Rec}$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{S}\,\kappa\circ}{\Gamma \vdash \mathtt{e} : \mathtt{S}\,\kappa}\ \textsc{T-Subsume}$$

Figure 6. Expression typing

$$\frac{\mathtt{ET} \leq \mathtt{ET}'' \quad \mathtt{ET}'' \leq \mathtt{ET}'}{\mathtt{ET} \leq \mathtt{ET}'} \qquad \frac{}{\mathtt{S}\,\kappa\circ \leq \mathtt{S}\,\kappa} \qquad \frac{\kappa \leq \kappa'}{\mathtt{S}\,\kappa \leq \mathtt{S}\,\kappa'}$$

$$\mathtt{iso} \leq \mathtt{trn} \leq \{\mathtt{ref}, \mathtt{val}\} \leq \mathtt{box} \leq \mathtt{tag}$$

$$\mathit{Sendable}(\mathtt{T}) \textit{ iff } \mathtt{T} = \mathtt{S}\,\kappa \wedge \kappa \in \{\mathtt{iso}, \mathtt{val}, \mathtt{tag}\}$$

Figure 7. Sub-types and sendable types.

$\kappa \triangleright \kappa'$			κ'			
κ	iso	trn	ref	val	box	tag
iso	iso	tag	tag	val	tag	tag
trn	iso	trn	box	val	box	tag
ref	iso	trn	ref	val	box	tag
val	val	val	val	val	val	tag
box	tag	box	box	val	box	tag
tag	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

Table 3. Viewpoint adaptation.

$\kappa \triangleleft \kappa'$			κ'			
κ	iso	trn	ref	val	box	tag
iso	$\surd$			$\surd$		$\surd$
trn	$\surd$	$\surd$		$\surd$		$\surd$
ref	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
val						
box						
tag						

Table 4. Safe to write.

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

Capabilities at field read When reading a field $\mathtt{f}$ from an object ι we obtain a temporary. The capability of this temporary must be a combination of κ, the capability of the path leading to ι, and κ', the capability with which ι sees the field. We express this through the operator $\triangleright$, defined in fig. 3. This operator is less precise than $\blacktriangleright$, i.e. $\kappa \blacktriangleright \kappa' \leq \kappa \triangleright \kappa'$. The new temporaries introduced must preserve well-formed heap visibility, in particular $WFV.4$. These rules forbid temporary aliases to $\mathtt{trn}$ or $\mathtt{ref}$ fields of an $\mathtt{iso}$, and therefore we obtain $\mathtt{iso} \triangleright \mathtt{trn} = \mathtt{iso} \triangleright \mathtt{ref} = \mathtt{tag}$. Also, they require that any aliases to $\mathtt{ref}$ fields of a $\mathtt{trn}$ are $\mathtt{box}$, including temporary references. Therefore, $\mathtt{trn} \triangleright \mathtt{ref} = \mathtt{box}$.

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 $\Delta_0, \chi_0\alpha_1 \vdash \iota_{17} : \texttt{ref}, (1, \texttt{this}) \cdot \texttt{f1} \cdot \texttt{f5} \cdot \texttt{f6}$.

Storing a reference into a field of an object ι 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 fig. 4, expresses which reference capabilities κ' are safe to write into origin κ. 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 $\texttt{this.f1.f5.f6} = (\texttt{x1} = \texttt{null})$ is legal even though the field f6 is ref and ref is not safe to write into iso. Namely, $\texttt{x1} = \texttt{null}$ has type iso∘, and aliased type iso, and $\texttt{iso} \leq \texttt{ref}$, 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 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 ref∘, then recover e has type iso∘.

Definition 2. Capability recovery

$$\mathcal{R}(\texttt{S}\,\kappa\,\phi) = \begin{cases} \texttt{S}\,\texttt{iso}\,\phi & \textit{iff } \kappa \in \{\texttt{iso}, \texttt{trn}, \texttt{ref}\} \\ \texttt{S}\,\texttt{val} & \textit{iff } \kappa \in \{\texttt{val}, \texttt{box}\} \\ \texttt{S}\,\texttt{tag} & \textit{otherwise} \end{cases}$$

$\mathcal{R}(\texttt{ET})$ is 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.

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 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 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.

P	∈	*Program*	::=	$\overline{\texttt{CT}}\,\overline{\texttt{AT}}$
CT	∈	*ClassDef*	::=	$\texttt{class}\,\texttt{C}\,\overline{\texttt{F}}\,\overline{\texttt{K}}\,\overline{\texttt{M}}$
AT	∈	*ActorDef*	::=	$\texttt{actor}\,\texttt{A}\,\overline{\texttt{F}}\,\overline{\texttt{K}}\,\overline{\texttt{M}}\,\overline{\texttt{B}}$
S	∈	*TypeID*	::=	A | C
T	∈	*Type*	::=	$\texttt{S}\,\kappa$
ET	∈	*ExtType*	::=	T | S (iso | trn | ref)∘
F	∈	*Field*	::=	var f : T
K	∈	*Ctor*	::=	$\texttt{new}\,\texttt{k}(\overline{\texttt{x}} : \overline{\texttt{T}}) \Rightarrow \texttt{e}$
M	∈	*Func*	::=	$\texttt{fun}\,\kappa\,\texttt{m}(\overline{\texttt{x}} : \overline{\texttt{T}}) : \texttt{ET} \Rightarrow \texttt{e}$
B	∈	*Behv*	::=	$\texttt{be}\,\texttt{b}(\overline{\texttt{x}} : \overline{\texttt{T}}) \Rightarrow \texttt{e}$
n	∈	*MethodID*	::=	k | m | b
κ	∈	*Cap*	::=	iso | trn | ref | val | box | tag
e	∈	*Expr*	::=	this | x | x = e | null | e; e
			|	e.f | e.f = e | recover e
			|	$\texttt{e.m}(\overline{\texttt{e}})$ | $\texttt{e.b}(\overline{\texttt{e}})$ | $\texttt{S.k}(\overline{\texttt{e}})$
E[·]	∈	*ExprHole*	::=	x = E[·] | E[·]; e | (E[·]) | E[·].f
			|	e.f = E[·] | E[·].f = z | $\texttt{E}[\cdot]\texttt{.n}(\overline{\texttt{z}})$
			|	$\texttt{e.n}(\overline{\texttt{z}}, \texttt{E}[\cdot], \overline{\texttt{e}})$ | recover E[·]

Figure 8. Syntax

C	∈	*ClassID*	k	∈	*CtorID*
A	∈	*ActorID*	m	∈	*FuncID*
f	∈	*FieldID*	b	∈	*BehvID*
this, x	∈	*SourceID*	n	∈	*CtorID* ∪ *BehvID*
t	∈	*TempID*	y, z	∈	*LocalID*

Figure 9. Identifiers

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, $\Gamma \vdash \texttt{e} : \texttt{ET}$ and $\Gamma(\texttt{x}) = \texttt{S}\,\kappa$ and $\kappa' \leq \kappa$ does not imply that $\Gamma[\texttt{x} \mapsto \texttt{S}\,\kappa'] \vdash \texttt{e} : \texttt{ET}'$ for some type $\texttt{ET}'$. For example, take class C with a field f of type C ref, and Γ such that $\Gamma(\texttt{x}) = \texttt{C}\,\texttt{ref}$ and $\Gamma' = \Gamma[\texttt{x} \mapsto \texttt{C}\,\texttt{trn}]$. Then $\texttt{x.f} = \texttt{x}$ is type correct in Γ but not in Γ'.

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-

$$
\begin{array}{llll}
\chi & \in & Heap & = Addr \rightarrow (Actor \vee Object) \\
\sigma & \in & Stack & = ActorAddr \cdot \overline{Frame} \\
\varphi & \in & Frame & = MethodID \times (LocalID \rightarrow Value) \\
 & & & \quad \times ExprHole \\
 & & LocalID & = SourceID \cup TempID \\
v & \in & Value & = Addr \cup \{null\} \\
\iota & \in & Addr & = ActorAddr \cup ObjectAddr \\
\alpha & \in & ActorAddr & \\
\omega & \in & ObjectAddr & \\
 & & Actor & = ActorID \times (FieldID \rightarrow Value) \\
 & & & \quad \times \overline{Message} \times Stack \times Expr \\
 & & Object & = ClassID \times (FieldID \rightarrow Value) \\
\mu & \in & Message & = MethodID \times \overline{Value}
\end{array}
$$

Figure 10. Runtime entities

structors, 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 *capability annotations* κ on types and functions, where:

$\kappa \in \{\texttt{iso}, \texttt{trn}, \texttt{ref}, \texttt{val}, \texttt{box}, \texttt{tag}\}$

These capabilities are the foundation of our type system.

Types consist of a class or actor identifier `S` followed by a capability κ. In addition, extended types `ET` can be *unaliased*, $\circ$. 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 $\overline{\mathtt{F}}$, with the convention that the n^{th} element is referred to as $\mathtt{F_n}$. Similarly, $\overline{\mathtt{x}} : \overline{\mathtt{T}}$ indicates a pairwise sequence of identifiers and types. To reduce notation, we assume a *fixed* program `P`.

6. Operational semantics

The operational semantics has the shape $\chi \rightarrow \chi'$, where χ, χ' are heaps mapping object addresses ω to their class identifier and their fields, and actor addresses α 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. fig. 17.

We use `x` to indicate a source identifier, `t` to indicate a temporary identifier, and `y` and `z` to indicate identifiers which may be either.

A call stack consists of an actor address α followed by a sequence of frames φ. 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 $\chi, \sigma, \mathtt{e} \rightsquigarrow \chi', \sigma', \mathtt{e}'$ expresses local execution within a *single* actor. $\mathcal{M}$ and $\mathcal{F}$ 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 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 will be 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 ι. The method parameters $\overline{\mathtt{x}}$ and the method body `e` are looked up using the method `m` and the type `S` of ι 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 ω 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 χ 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,

$$\frac{\chi, \sigma\cdot\varphi, \mathtt{e} \rightsquigarrow \chi', \sigma\cdot\varphi', \mathtt{e}'}{\chi, \sigma\cdot\varphi, \mathtt{E}[\mathtt{e}] \rightsquigarrow \chi', \sigma\cdot\varphi', \mathtt{E}[\mathtt{e}']}\ \textsc{ExprHole}$$

$$\frac{\mathtt{t} \notin \varphi \quad \iota = \varphi(\mathtt{z}) \quad \varphi' = \varphi[\mathtt{t} \mapsto \chi(\iota, \mathtt{f})]}{\chi, \sigma\cdot\varphi, \mathtt{z.f} \rightsquigarrow \chi, \sigma\cdot\varphi', \mathtt{t}}\ \textsc{Fld}$$

$$\frac{\mathtt{t} \notin \varphi \quad \varphi' = \varphi[\mathtt{t} \mapsto \mathit{null}]}{\chi, \sigma\cdot\varphi, \mathtt{null} \rightsquigarrow \chi, \sigma\cdot\varphi', \mathtt{t}}\ \textsc{Null}$$

$$\frac{}{\chi, \sigma, \mathtt{z}; \mathtt{e} \rightsquigarrow \chi, \sigma, \mathtt{e}}\ \textsc{Seq}$$

$$\frac{\mathtt{t} \notin \varphi \quad \varphi' = \varphi[\mathtt{x} \mapsto \varphi(\mathtt{z}), \mathtt{t} \mapsto \varphi(\mathtt{x})]}{\chi, \sigma\cdot\varphi, \mathtt{x} = \mathtt{z} \rightsquigarrow \chi, \sigma\cdot\varphi', \mathtt{t}}\ \textsc{AsnLocal}$$

$$\frac{\begin{array}{c}\mathtt{t} \notin \varphi \quad \iota = \varphi(\mathtt{z}) \quad \varphi' = \varphi[\mathtt{t} \mapsto \chi(\iota, \mathtt{f})] \\ \chi' = \chi[\varphi(\mathtt{z}), \mathtt{f} \mapsto \varphi(\mathtt{y})]\end{array}}{\chi, \sigma\cdot\varphi, \mathtt{z.f} = \mathtt{y} \rightsquigarrow \chi', \sigma\cdot\varphi', \mathtt{t}}\ \textsc{AsnFld}$$

$$\frac{\begin{array}{c}\iota = \varphi(\mathtt{z}) \quad \mathcal{M}(\chi(\iota)\downarrow_1, \mathtt{m}) = (\_, \overline{\mathtt{x}} : \_, \mathtt{e}, \_) \\ \varphi' = (\mathtt{m}, [\mathtt{this} \mapsto \iota, \overline{\mathtt{x}} \mapsto \varphi(\overline{\mathtt{y}})], \mathtt{E}[\cdot])\end{array}}{\chi, \sigma\cdot\varphi, \mathtt{E}[\mathtt{z.m}(\overline{\mathtt{y}})] \rightsquigarrow \chi, \sigma\cdot\varphi\cdot\varphi', \mathtt{e}}\ \textsc{Sync}$$

$$\frac{\begin{array}{c}\mathtt{t} \notin \varphi \quad \iota = \varphi'(\mathtt{z}) \\ \varphi'\downarrow_3 = \mathtt{E}[\cdot] \quad \varphi'' = \varphi[\mathtt{t} \mapsto \iota]\end{array}}{\chi, \sigma\cdot\varphi\cdot\varphi', \mathtt{z} \rightsquigarrow \chi, \sigma\cdot\varphi'', \mathtt{E}[\mathtt{t}]}\ \textsc{Return}$$

$$\frac{\alpha = \varphi(\mathtt{z}) \quad \chi(\alpha)\downarrow_3 = \overline{\mu}}{\chi, \sigma\cdot\varphi, \mathtt{z.b}(\overline{\mathtt{y}}) \rightsquigarrow \chi[\alpha \mapsto \overline{\mu}\cdot(\mathtt{b}, \varphi(\overline{\mathtt{y}}))], \sigma\cdot\varphi, \mathtt{z}}\ \textsc{Async}$$

$$\frac{\begin{array}{c}\mathtt{A} = \chi(\alpha)\downarrow_1 \quad (\mathtt{n}, \overline{v})\cdot\overline{\mu} = \chi(\alpha)\downarrow_3 \\ \mathcal{M}(\mathtt{A}, \mathtt{n}) = (\_, \overline{\mathtt{x}} : \_, \mathtt{e}, \_) \\ \varphi = (\mathtt{n}, [\mathtt{this} \mapsto \alpha, \overline{\mathtt{x}} \mapsto \overline{v}], \cdot)\end{array}}{\chi, \alpha, \varepsilon \rightsquigarrow \chi[\alpha \mapsto \overline{\mu}], \alpha\cdot\varphi, \mathtt{e}}\ \textsc{Behave}$$

$$\frac{\begin{array}{c}\omega \notin \mathit{dom}(\chi) \quad \overline{\mathtt{f}} = \mathcal{F}s(\mathtt{C}) \\ \mathcal{M}(\mathtt{C}, \mathtt{k}) = (\_, \overline{\mathtt{x}} : \_, \mathtt{e}, \_) \\ \chi' = \chi[\omega \mapsto (\mathtt{C}, \overline{\mathtt{f}} \mapsto \mathit{null})] \\ \varphi' = (\mathtt{k}, [\mathtt{this} \mapsto \omega, \overline{\mathtt{x}} \mapsto \varphi(\overline{\mathtt{y}})], \mathtt{E}[\cdot])\end{array}}{\chi, \sigma\cdot\varphi, \mathtt{E}[\mathtt{C.k}(\overline{\mathtt{y}})] \rightsquigarrow \chi', \sigma\cdot\varphi\cdot\varphi', \mathtt{e}}\ \textsc{Ctor}$$

$$\frac{\begin{array}{c}\alpha \notin \mathit{dom}(\chi) \quad \overline{\mathtt{f}} = \mathcal{F}s(\mathtt{A}) \\ \mathtt{t} \notin \varphi \quad \varphi' = \varphi[\mathtt{t} \mapsto \alpha] \\ \chi' = \chi[\alpha \mapsto (\mathtt{A}, \overline{\mathtt{f}} \mapsto \mathit{null}, (\mathtt{k}, \varphi(\overline{\mathtt{y}}), \alpha, \varepsilon)]\end{array}}{\chi, \sigma\cdot\varphi, \mathtt{A.k}(\overline{\mathtt{y}}) \rightsquigarrow \chi', \sigma\cdot\varphi', \mathtt{t}}\ \textsc{Ator}$$

$$\frac{\chi, \sigma, \mathtt{e} \rightsquigarrow \chi', \sigma', \mathtt{e}'}{\chi, \sigma, \mathtt{recover}\ \mathtt{e} \rightsquigarrow \chi', \sigma', \mathtt{recover}\ \mathtt{e}'}\ \textsc{Rec1}$$

$$\frac{\mathtt{t} \notin \varphi \quad \varphi' = \varphi[\mathtt{t} \mapsto \varphi(\mathtt{z})]}{\chi, \sigma, \mathtt{recover}\ \mathtt{z} \rightsquigarrow \chi, \sigma, \mathtt{t}}\ \textsc{Rec2}$$

$$\frac{\mathtt{t} \notin \varphi \quad \varphi(\mathtt{z}) = \mathit{null} \quad \varphi' = \varphi[\mathtt{t} \mapsto \mathit{null}]}{\begin{array}{c}\chi, \sigma\cdot\varphi, \mathtt{z.f} \rightsquigarrow \chi, \sigma\cdot\varphi', \mathtt{t} \\ \chi, \sigma\cdot\varphi, \mathtt{z.f} = \mathtt{y} \rightsquigarrow \chi, \sigma\cdot\varphi', \mathtt{t} \\ \chi, \sigma\cdot\varphi, \mathtt{z.n}(\overline{\mathtt{y}}) \rightsquigarrow \chi, \sigma\cdot\varphi', \mathtt{t}\end{array}}\ \textsc{Except}$$

$$\frac{\chi, \chi(\alpha)\downarrow_4, \chi(\alpha)\downarrow_5 \rightsquigarrow \chi', \sigma, \mathtt{e}}{\chi \rightarrow \chi'[\alpha \mapsto (\sigma, \mathtt{e})]}\ \textsc{Global}$$

Figure 11. Execution.

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 $\mathtt{iso} \triangleright \mathtt{iso} = \mathtt{iso}$, 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. $WFT(\Delta, \chi, \alpha, i, \mathtt{e})$ iff:

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

The requirements above do not apply to *unaliased unique* capabilities, e.g. $\mathtt{iso}\circ$, or $\mathtt{trn}\circ$. When proving type preservation, we maintain the property
$WFT(\Delta, \chi, \alpha, i, \mathtt{e})$ by turning the types of temporaries with unique capabilities $\kappa \in \{\mathtt{iso}, \mathtt{trn}\}$ into their aliases, $\mathcal{A}(\kappa)$, 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

- $\Delta \vdash \chi\diamond$ iff $\forall\iota, \alpha \in dom(\chi)$, $\chi \vdash \iota\diamond$ and $\Delta, \chi \vdash \alpha\diamond$ and $WFV(\Delta, \chi)$
- $\chi \vdash \iota\diamond$ iff $\forall \mathtt{f}, \mathcal{F}(\chi(\iota)\downarrow_1, \mathtt{f}) = \mathtt{S}\,\kappa$ implies $\chi(\iota, \mathtt{f})\downarrow_1 = \mathtt{S}$
- $\Delta, \chi \vdash \alpha\diamond$ iff $\chi(\alpha) = (\_, \_, \overline{\mu}, \alpha \cdot \overline{\varphi}, \mathtt{e})$ and $\forall i, \Delta, \chi, \alpha \vdash \varphi_i, i\diamond$ and $\forall j, \Delta, \chi \vdash \mu_j, j\diamond$
- $\Delta, \chi, \alpha \vdash \varphi, i\diamond$ iff given $\varphi = (\mathtt{m}, \_, \mathtt{E}[\cdot])$ and $\mathcal{M}(\varphi, \chi) = (\mathtt{T}, \overline{\mathtt{x} : \mathtt{T}}, \_, \mathtt{ET})$ and $\Delta(\alpha, i) = \Gamma$ then
  1. $\Gamma(\mathtt{this}) = \mathtt{T}$ and $\forall j \in 1..|\overline{\mathtt{T}}|.\Gamma(\mathtt{x_j}) = \mathtt{T_j}$
  2. $\forall \mathtt{z} \in \varphi, \Gamma(\mathtt{z}) = \mathtt{S}\,\kappa\,\phi$ and $\chi(\varphi(\mathtt{z}))\downarrow_1 = \mathtt{S}$
  3. If $i = 1$ then $\mathtt{E}[\cdot] = \cdot$ and $\varphi(\mathtt{this}) = \alpha$
  4. If $i > 1$, given $\chi(\alpha)\downarrow_4 = \alpha \cdot \overline{\varphi}$ and $\Gamma' = \Delta(\alpha, i-1)$ and $\mathtt{t} \notin \Gamma'$ and $\Gamma'' = \Gamma'[\mathtt{t} \mapsto \mathtt{ET}]$ then
     (a) $\Gamma'' \vdash \mathtt{E}[\mathtt{t}] : \mathcal{M}(\varphi_{i-1}, \chi)\downarrow_4$
     (b) $WFT(\Delta[(\alpha, i) \mapsto \Gamma''], \chi, \alpha, i, \mathtt{E}[\mathtt{t}])$
  5. If $i = |\chi(\alpha)\downarrow_4|$ then $WFT(\Delta, \chi, \alpha, i, \mathtt{e})$ and $\Gamma \vdash \mathtt{e} : \mathtt{ET}$
- $\Delta, \chi, \alpha \vdash \mu, i\diamond$ iff given $\mu = (\mathtt{b}, \overline{v})$ and $v_j = \iota$ and $\mathcal{M}(\chi(\alpha)\downarrow_1, \mathtt{b}) = (\_, \overline{\mathtt{x} : \mathtt{S}\,\kappa}, \_, \_)$ and $\Delta(\alpha, -i) = \Gamma$ then
  1. $\chi(\iota)\downarrow_1 = \mathtt{S_j}$
  2. $\Gamma(\mathtt{x_j}) = \mathtt{S}\,\kappa$

Figure 12. Well-formed heaps.

fig. 6 require the alias of a type $(\ldots \vdash_{\mathcal{A}} \ldots)$ in all such situations. This is explained further in lemma 10 in the appendix.

Theorem 1. *A well-formed heap ensures data race freedom.* $\forall\Delta, \chi, \alpha_1, \alpha_2, \mathtt{f}, \mathtt{g}$, *if*

1. $\Delta \vdash \chi\diamond$, *and*
2. $\chi(\alpha_1) = (\_, \_, \sigma_1, \_, \mathtt{E_1}[\mathtt{z_1}.\mathtt{f} = \mathtt{z_3}])$, *and*
3. $\chi(\alpha_2) = (\_, \_, \sigma_2, \_, \mathtt{E_2}[\mathtt{z_2}.\mathtt{g}])$

then $\chi(\alpha_1, |\sigma_1| \cdot \mathtt{z_1}) \neq \chi(\alpha_2, |\sigma_2| \cdot \mathtt{z_2})$.

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

Theorem 2. *Well-formedness is preserved.* $\forall\Delta, \chi$, *if* $\Delta \vdash \chi\diamond$ *and* $\chi \to \chi'$ *then* $\exists\Delta'.\Delta' \vdash \chi'\diamond$.

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.

	Our Work	Gordon	Æminium	DPJ	Kilim	Haller	Scala	Erlang	Rust
Zero-copy	√	√	√	√	√	√	√		√
Data-race free	√	√	√	√	√[2]	√[3]		√	√
Statically data-race free	√	√	√	√	√	√			[4]
Non-tree messages	√	√				√	√	√	√
Read unique (`iso`)	√	√	√		√	√			
Write unique (`trn`)	√								
Mutability (`ref`)	√	√	√	√	√	√	√		√
Immutability (`val`)	√	√	√		√		[5]	√	√
Cyclic immutability	√	√							
Identity (`tag`)	√		[6]						
Destructive read	√	√			√	√			√
Recovery	√	√							
Using uniques ($\texttt{iso} \triangleright \texttt{x}$)	√								
Actors	√				√	√	√	√	
Formal proof	√	√	√	√	√	√			
Native compilation	√								√

Table 5. Feature comparison.

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

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].

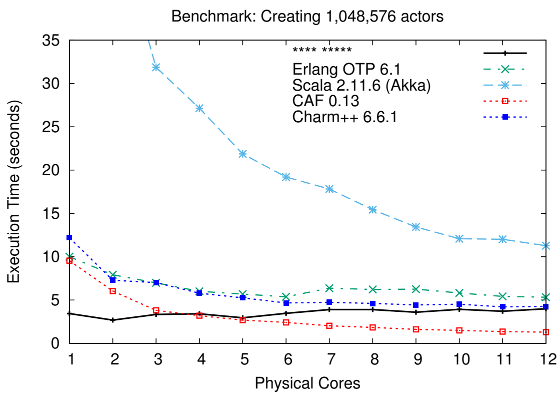

Figure 13. Actor creation, where **** is our work.

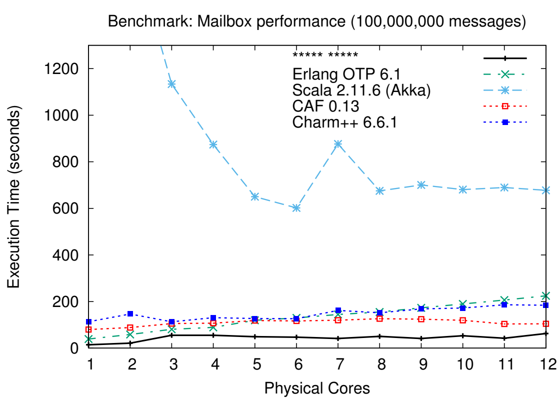

Figure 14. Mailbox performance, where **** is our work.

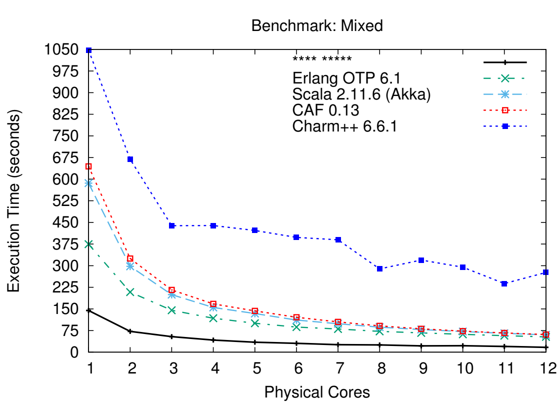

Figure 15. Mixed case performance, where **** is our work.

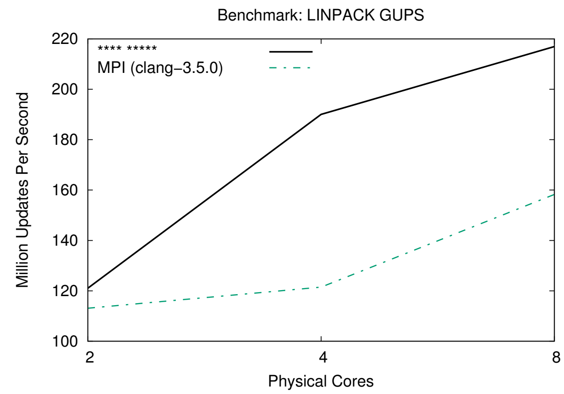

Figure 16. LINPACK GUPS, where **** is our work.

Opteron 6338P with 64 GB of memory across 2 NUMA nodes. The results shown are the average of 100 runs.

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 performance[7]. 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 implementation[8].

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 available[9].

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

[1] http://icl.cs.utk.edu/hpcc/.

[2] https://github.com/actor-framework/benchmarks/.

[3] G. Agha and C. Hewitt. Concurrent programming using actors. In *Object-oriented concurrent programming*, pages 37–53. MIT Press, 1987.

[4] J. Armstrong, R. Virding, C. Wikström, and M. Williams. Concurrent programming in erlang. 1993.

[5] H. G. Baker. "use-once" variables and linear objects: storage management, reflection and multi-threading. *ACM Sigplan Notices*, 30(1):45–52, 1995.

[6] R. L. Bocchino Jr, V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. *ACM Sigplan Notices*, 44(10):97–116, 2009.

[7] C. Boyapati and M. Rinard. A parameterized type system for race-free java programs. In *ACM SIGPLAN Notices*, volume 36, pages 56–69. ACM, 2001.

[8] J. Boyland. Alias burying: Unique variables without destructive reads. *Software: Practice and Experience*, 31(6):533–553, 2001.

[9] J. Boyland. Checking interference with fractional permissions. In *Static Analysis*, pages 55–72. Springer, 2003.

[10] J. Boyland, J. Noble, and W. Retert. Capabilities for sharing. In *ECOOP 2001-Object-Oriented Programming*, pages 2–27. Springer, 2001.

[11] D. Charousset, T. C. Schmidt, R. Hiesgen, and M. Wählisch. Native actors: a scalable software platform for distributed, heterogeneous environments. In *Proceedings of the 2013 workshop on Programming based on actors, agents, and decentralized control*, pages 87–96. ACM, 2013.

[12] D. Clarke and T. Wrigstad. External uniqueness is unique enough. *ECOOP 2003–Object-Oriented Programming*, pages 59–67, 2003.

[13] D. Clarke, T. Wrigstad, J. Östlund, and E. Johnsen. Minimal ownership for active objects. *Programming Languages and Systems*, pages 139–154, 2008.

[14] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *ACM SIGPLAN Notices*, volume 33, pages 48–64. ACM, 1998.

[15] S. Clebsch and S. Drossopoulou. Fully concurrent garbage collection of actors on many-core machines. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages and applications*, pages 553–570. ACM, 2013.

[16] D. Cunningham, W. Dietl, S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers. Universe types for topology and encapsulation. In *Formal Methods for Components and Objects*, pages 72–112. Springer Berlin Heidelberg, 2008.

[17] M. Dodds, X. Feng, M. Parkinson, and V. Vafeiadis. Deny-guarantee reasoning. In *Programming Languages and Systems*, pages 363–377. Springer, 2009.

[18] C. Flanagan and M. Abadi. Types for safe locking. In *Programming Languages and Systems*, pages 91–108. Springer Berlin Heidelberg, 1999.

[19] C. S. Gordon, M. J. Parkinson, J. Parsons, A. Bromfield, and J. Duffy. Uniqueness and reference immutability for safe parallelism. In *ACM SIGPLAN Notices*, volume 47, pages 21–40. ACM, 2012.

[20] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[21] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *ACM SIGPLAN Notices*, volume 37, pages 282–293. ACM, 2002.

[22] P. Haller and M. Odersky. Capabilities for uniqueness and borrowing. In *ECOOP 2010–Object-Oriented Programming*, pages 354–378. Springer, 2010.

[23] N. D. Matsakis and F. S. Klock, II. The rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, pages 103–104, New York, NY, USA, 2014. ACM.

[24] M. S. Miller, K.-P. Yee, J. Shapiro, et al. Capability myths demolished. Technical report, Technical Report SRL2003-02, Johns Hopkins University Systems Research Laboratory, 2003. http://www. erights. org/elib/capability/duals, 2003.

[25] K. Naden, R. Bocchino, J. Aldrich, and K. Bierhoff. A type system for borrowing permissions. *SIGPLAN Not.*, 47(1):557–570, Jan. 2012.

[26] J. Östlund, T. Wrigstad, D. Clarke, and B. Åkerblom. Ownership, uniqueness, and immutability. *Objects, Components, Models and Patterns*, pages 178–197, 2008.

[27] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for java. In *ECOOP 2008–Object-Oriented Programming*, pages 104–128. Springer, 2008.

[28] S. Stork, K. Naden, J. Sunshine, M. Mohr, A. Fonseca, P. Marques, and J. Aldrich. Æminium: A permission-based concurrent-by-default programming language approach. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(1):2, 2014.

[29] P. Wadler. Linear types can change the world. In *IFIP TC*, volume 2, pages 347–359. Citeseer, 1990.

[30] Y. Zibin, A. Potanin, P. Li, M. Ali, and M. D. Ernst. Ownership and immutability in generic java. In *ACM Sigplan Notices*, volume 45, pages 598–617. ACM, 2010.

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 $\mathcal{P}$ returns a type definition for a class identifier $\mathtt{C}$ or actor identifier $\mathtt{A}$. This contains the fields $\overline{\mathtt{F}}$, constructors $\overline{\mathtt{K}}$, functions $\overline{\mathtt{M}}$, and behaviours $\overline{\mathtt{B}}$ defined for that type. Since classes have no asynchronous behaviour, the last entry in $\mathcal{P}(\mathtt{C})$ is empty, i.e. ε. Function $\mathcal{F}s$ returns the identifiers of all fields defined in a type $\mathtt{S}$, and function $\mathcal{F}$ returns the type of field $\mathtt{f}$ in $\mathtt{S}$. Function $\mathcal{M}$ returns method information for some method in $\mathtt{S}$. 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 unaliased, 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 `recover` 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 $\mathtt{e}$ a *redex* if it has one of the following forms:

$$\mathtt{e} \ ::= \ \mathtt{z.f} \mid \mathtt{z.f} = \mathtt{y} \mid \mathtt{z.m}(\overline{\mathtt{y}}) \mid \mathtt{z.b}(\overline{\mathtt{y}}) \mid \mathtt{S.k}(\overline{\mathtt{z}})$$

Lemma 1. *Uniqueness of contexts. For any expressions* $\mathtt{e}_1$, $\mathtt{e}_2$ *and contexts* $\mathtt{E}_1[\cdot]$, $\mathtt{E}_2[\cdot]$, *if* $\mathtt{E}_1[\mathtt{e}_2] \equiv \mathtt{E}_2[\mathtt{e}_2]$ *and* $\mathtt{e}_1$ *and* $\mathtt{e}_2$ *are redexes then* $\mathtt{E}_1[\cdot] \equiv \mathtt{E}_2[\cdot]$ *and* $\mathtt{e}_1 \equiv \mathtt{e}_2$.

- $\varphi(\mathtt{x}) = \varphi\downarrow_2(\mathtt{x})\downarrow_1$
- $\varphi[\mathtt{x} \mapsto v] = (\varphi\downarrow_1, \varphi\downarrow_2[\mathtt{x} \mapsto v], \varphi\downarrow_3)$
- $\chi(\iota, \mathtt{f}) = \chi(\iota)\downarrow_2(\mathtt{f})$
- $\chi[\omega, \mathtt{f} \mapsto v] = \chi[\omega \mapsto (\chi(\omega)\downarrow_1, \chi(\omega)\downarrow_2[\mathtt{f} \mapsto v]]$
- $\chi[\alpha, \mathtt{f} \mapsto v] = \chi[\alpha \mapsto (\chi(\alpha)\downarrow_1, \chi(\alpha)\downarrow_2[\mathtt{f} \mapsto v], \chi(\alpha)\downarrow_3, \chi(\alpha)\downarrow_4, \chi(\alpha)\downarrow_5)]$
- $\chi[\alpha \mapsto (\sigma, \mathtt{e})] = \chi[\alpha \mapsto (\chi(\alpha)\downarrow_1, \chi(\alpha)\downarrow_2, \chi(\alpha)\downarrow_3, \sigma, \mathtt{e}]$
- $\chi[\alpha \mapsto \overline{\mu}] = \chi[\alpha \mapsto (\chi(\alpha)\downarrow_1, \chi(\alpha)\downarrow_2, \overline{\mu}, \chi(\alpha)\downarrow_4, \chi(\alpha)\downarrow_5]$

Figure 17. Auxiliary definitions

$$\frac{\mathtt{P} = \overline{\mathtt{CT}}\,\overline{\mathtt{AT}} \quad \mathtt{class\,C}\,\overline{\mathtt{F}}\,\overline{\mathtt{K}}\,\overline{\mathtt{M}} \in \overline{\mathtt{CT}}}{\mathcal{P}(\mathtt{C}) = \overline{\mathtt{F}}\,\overline{\mathtt{K}}\,\overline{\mathtt{M}}\,\varepsilon \quad \mathtt{C} \in \mathtt{P}}$$

$$\frac{\mathtt{P} = \overline{\mathtt{CT}}\,\overline{\mathtt{AT}} \quad \mathtt{actor\,A}\,\overline{\mathtt{F}}\,\overline{\mathtt{K}}\,\overline{\mathtt{M}}\,\overline{\mathtt{B}} \in \overline{\mathtt{AT}}}{\mathcal{P}(\mathtt{A}) = \overline{\mathtt{F}}\,\overline{\mathtt{K}}\,\overline{\mathtt{M}}\,\overline{\mathtt{B}} \quad \mathtt{A} \in \mathtt{P}}$$

$$\frac{\mathcal{P}(\mathtt{S}) = \overline{\mathtt{F}}\,\overline{\mathtt{K}}\,\overline{\mathtt{M}}\,\overline{\mathtt{B}}}{\mathcal{F}s(\mathtt{S}) = \{\mathtt{f} \mid \mathtt{var\,f : T} \in \overline{\mathtt{F}}\}}$$

$$\frac{\mathcal{P}(\mathtt{S}) = \overline{\mathtt{F}}\,\overline{\mathtt{K}}\,\overline{\mathtt{M}}\,\overline{\mathtt{B}} \quad \mathtt{var\,f : T} \in \overline{\mathtt{F}}}{\mathcal{F}(\mathtt{S}, \mathtt{f}) = \mathtt{T}}$$

$$\frac{\mathcal{P}(\mathtt{C}) = \overline{\mathtt{F}}\,\overline{\mathtt{K}}\,\overline{\mathtt{M}} \quad (\mathtt{new\,k}(\overline{\mathtt{x}} : \overline{\mathtt{T}}) \Rightarrow \mathtt{e}) \in \overline{\mathtt{K}}}{\mathcal{M}(\mathtt{C}, \mathtt{k}) = (\mathtt{C\,ref}, \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{e}, \mathtt{C\,ref}\circ)}$$

$$\frac{\mathcal{P}(\mathtt{A}) = \overline{\mathtt{F}}\,\overline{\mathtt{K}}\,\overline{\mathtt{M}}\,\overline{\mathtt{B}} \quad (\mathtt{new\,k}(\overline{\mathtt{x}} : \overline{\mathtt{T}}) \Rightarrow \mathtt{e}) \in \overline{\mathtt{K}}}{\mathcal{M}(\mathtt{A}, \mathtt{k}) = (\mathtt{A\,var}, \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{e}, \mathtt{A\,tag})}$$

$$\frac{\mathcal{P}(\mathtt{S}) = \overline{\mathtt{F}}\,\overline{\mathtt{K}}\,\overline{\mathtt{M}}\,\overline{\mathtt{B}} \quad (\mathtt{fun}\,\kappa\,\mathtt{m}(\overline{\mathtt{x}} : \overline{\mathtt{T}}) : \mathtt{ET} \Rightarrow \mathtt{e}) \in \overline{\mathtt{M}}}{\mathcal{M}(\mathtt{S}, \mathtt{m}) = (\mathtt{S}\,\kappa, \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{e}, \mathtt{ET})}$$

$$\frac{\mathcal{P}(\mathtt{A}) = \overline{\mathtt{F}}\,\overline{\mathtt{K}}\,\overline{\mathtt{M}}\,\overline{\mathtt{B}} \quad (\mathtt{be\,b}(\overline{\mathtt{x}} : \overline{\mathtt{T}}) \Rightarrow \mathtt{e}) \in \overline{\mathtt{B}}}{\mathcal{M}(\mathtt{A}, \mathtt{b}) = (\mathtt{A\,ref}, \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{e}, \mathtt{A\,tag})}$$

Figure 18. Lookup functions

$$\frac{\forall \mathtt{S} \in \mathtt{P}.\vdash \mathtt{S}\diamond}{\vdash \mathtt{P}\diamond} \text{ WF-PROGRAM}$$

$$\frac{\begin{array}{c}\mathcal{P}(\mathtt{S}) = \overline{\mathtt{F}}\,\overline{\mathtt{K}}\,\overline{\mathtt{M}}\,\overline{\mathtt{B}} \\ \forall \mathtt{var\,f} : \mathtt{S}\,\kappa \in \overline{\mathtt{F}}.\vdash \mathtt{S}\diamond \quad \forall \mathtt{K} \in \overline{\mathtt{K}}.\mathtt{S} \vdash \mathtt{K}\diamond \\ \forall \mathtt{M} \in \overline{\mathtt{M}}.\mathtt{S} \vdash \mathtt{M}\diamond \quad \forall \mathtt{B} \in \overline{\mathtt{B}}.\mathtt{S} \vdash \mathtt{B}\diamond\end{array}}{\vdash \mathtt{S}\diamond} \text{ WF-TYPE}$$

$$\frac{[\mathtt{this} \mapsto \mathtt{C\,var}, \overline{\mathtt{x}} \mapsto \overline{\mathtt{T}}] \vdash \mathtt{e} : \mathtt{C\,var}\circ}{\mathtt{C} \vdash \mathtt{new\,k}(\overline{\mathtt{x}} : \overline{\mathtt{T}}) \Rightarrow \mathtt{e}\diamond} \text{ WF-CTOR}$$

$$\frac{[\mathtt{this} \mapsto \mathtt{S}\kappa_\mathtt{r}, \overline{\mathtt{x}} \mapsto \overline{\mathtt{T}}] \vdash \mathtt{e} : \mathtt{ET}}{\mathtt{S} \vdash \mathtt{fun}\,\kappa_\mathtt{r}\,\mathtt{m}(\overline{\mathtt{x}} : \overline{\mathtt{T}}) : \mathtt{ET} \Rightarrow \mathtt{e}\diamond} \text{ WF-SYNC}$$

$$\frac{\begin{array}{c}Sendable(\mathtt{T_i}) \\ [\mathtt{this} \mapsto \mathtt{A\,var}, \overline{\mathtt{x}} \mapsto \overline{\mathtt{T}}] \vdash \mathtt{e} : \mathtt{A\,tag}\end{array}}{\mathtt{A} \vdash \mathtt{new\,k}(\overline{\mathtt{x}} : \overline{\mathtt{T}}) \Rightarrow \mathtt{e}\diamond} \text{ WF-ATOR}$$

$$\frac{\begin{array}{c}Sendable(\mathtt{T_i}) \\ [\mathtt{this} \mapsto \mathtt{A\,var}, \overline{\mathtt{x}} \mapsto \overline{\mathtt{T}}] \vdash \mathtt{e} : \mathtt{A\,tag}\end{array}}{\mathtt{A} \vdash \mathtt{be\,b}(\overline{\mathtt{x}} : \overline{\mathtt{T}}) \Rightarrow \mathtt{e}\diamond} \text{ WF-ASYNC}$$

Figure 19. Well-formed programs

- $\mathtt{z} \in \varphi$ iff $\mathtt{z} \in dom(\varphi\downarrow_2)$
- $\alpha \in \chi$ iff $\alpha \in dom(\chi)$
- $\Delta \vdash \alpha \in \chi$ iff $\alpha \in dom(\chi)$
- $\Delta \vdash \iota \in \chi$ iff $\exists \iota'$ such that $\Delta \vdash \iota' \in \chi$ and $\Delta, \chi, \iota' \vdash \iota : \_$
- $\mathcal{M}(\varphi, \chi) = \mathcal{M}(\chi(\varphi(\mathtt{this})\downarrow_1, \varphi\downarrow_1)$

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. $\Gamma \vdash \mathtt{E[e]} : \mathtt{ET} \Rightarrow \exists \mathtt{ET}'$ *and* $\Gamma, \mathtt{y} \mapsto \mathtt{ET}' \vdash \mathtt{E[y]} : \mathtt{ET}$ *and* $\Gamma \vdash \mathtt{e} : \mathtt{ET}'$ *and* $\mathtt{y} \notin dom(\Gamma)$
2. $\Gamma, \mathtt{y} \mapsto \mathtt{ET}' \vdash \mathtt{E[y]} : \mathtt{ET}$ *and* $\Gamma \vdash \mathtt{e} : \mathtt{ET}'$ *and* $\mathtt{y}$ *free in* $\mathtt{E}[\cdot] \Rightarrow \Gamma \vdash \mathtt{E[e]} : \mathtt{ET}$

Lemma 3. *Properties of capability operators.*
$\forall \kappa, \kappa_1, \kappa_2$:

1. *If* $\kappa_1 \sim_g \kappa_2$*, then* $\kappa_1 \sim_l \kappa_2$.
2. *If* $\kappa_1 \leq \kappa_2$*, then*
   (a) $\kappa_1 \sim_l \kappa \Rightarrow \kappa_2 \sim_l \kappa$
   (b) $\kappa_1 \sim_g \kappa \Rightarrow \kappa_2 \sim_g \kappa$
3. *If* $\kappa_1 \sim_l \kappa_2$*, and both* $\kappa_1 \triangleright \kappa$ *and* $\kappa_2 \triangleright \kappa$ *are defined, then* $\kappa_1 \triangleright \kappa \sim_l \kappa_2 \triangleright \kappa$.
4. *If* $\kappa_1 \sim_g \kappa_2$*, and both* $\kappa_1 \triangleright \kappa$ *and* $\kappa_2 \triangleright \kappa$ *are defined, then* $\kappa_1 \triangleright \kappa \sim_g \kappa_2 \triangleright \kappa$
5. $\kappa_2 \leq \kappa_1 \triangleright \kappa_2$ *or* $\kappa_1 = \mathtt{val}$ *or* $\kappa_1 \triangleright \kappa_2$ *undefined*
6. *If* $\mathcal{A}(\kappa_1) \leq \kappa_2$ *then*
   (a) $\kappa_1 \sim_l \kappa \Rightarrow \kappa_2 \sim_l \kappa$
   (b) $\kappa_1 \sim_g \kappa \Rightarrow \kappa_1 \sim_g \kappa$
   (c) $\mathcal{A}(\kappa_1 \triangleright \kappa) \leq \kappa_2 \triangleright \kappa$
7. *If* $\mathcal{A}(\kappa_1) \leq \kappa_2$ *and* $\mathcal{A}(\kappa_2) \leq \kappa_4$ *then*
   (a) $\kappa_1 \sim_l \kappa_2 \Rightarrow \kappa_3 \sim_l \kappa_4$
   (b) $\kappa_1 \sim_g \kappa_2 \Rightarrow \kappa_3 \sim_g \kappa_4$

Proof. By case analysis on κ_1 and κ_2. □

On the other hand, $\kappa_1 \leq \kappa_2$does not imply that $\kappa \triangleright \kappa_2 \leq \kappa \triangleright \kappa_2$. For example, $\mathtt{iso} \leq \mathtt{trn}$, but $\mathtt{box} \triangleright \mathtt{iso} = \mathtt{tag} \not\leq \mathtt{box} \triangleright \mathtt{trn} = \mathtt{box}$. Similarly, $\kappa_1 \leq \kappa_2$does not imply that $\kappa_1 \triangleright \kappa \leq \kappa_2 \triangleright \kappa$; take $\mathtt{iso} \triangleright \mathtt{trn} = \mathtt{tag} \not\leq \mathtt{trn} \triangleright \mathtt{trn} = \mathtt{trn}$. Finally the $\triangleright$-operator is not associative, i.e. $(\kappa_1 \triangleright \kappa_2) \triangleright \kappa_3 \neq \kappa_1 \triangleright (\kappa_2 \triangleright \kappa_3)$; take $(\mathtt{iso} \triangleright \mathtt{trn}) \triangleright \mathtt{val} = \bot \neq \mathtt{iso} \triangleright (\mathtt{trn} \triangleright \mathtt{val}) = \mathtt{val}$.

D. Well-formed runtime configurations

Lemma 4. *Properties of deep viewpoint adaptation.*
$\forall \kappa, \kappa_1 ..., \kappa_n$:

1. *If* $\kappa_1 \leq \kappa_2$*then* $\kappa_1 \blacktriangleright \kappa \leq \kappa_2 \triangleright \kappa$*, or* $\kappa_2 = \mathtt{val}$.
2. $\kappa_1 \blacktriangleright \kappa_2 = \mathtt{val}$ *iff* $\kappa_1 \triangleright \kappa_2 = \mathtt{val}$.
3. $\kappa_1 \blacktriangleright \kappa_2 \leq \kappa_1 \triangleright \kappa_2$
4. $(...(\kappa_1 \blacktriangleright \kappa_2) \blacktriangleright \kappa_3 ...) \blacktriangleright \kappa_n \leq (...(\kappa_1 \triangleright \kappa_2) \triangleright \kappa_3 ...) \triangleright \kappa_n$
5. $(...(\kappa_1 \blacktriangleright \kappa_2) \blacktriangleright \kappa_3 ...) \blacktriangleright \kappa_n = \mathtt{val}$ *iff* $(...(\kappa_1 \triangleright \kappa_2) \triangleright \kappa_3 ...) \triangleright \kappa_n = \mathtt{val}$
6. *If* $\kappa_1 \sim_l \kappa_2$*and* $\kappa_1, \kappa_2 \neq \mathtt{tag}$*, then* $\kappa_1 \blacktriangleright \kappa \sim_l \kappa_2 \blacktriangleright \kappa$ *or* $\kappa_1 = \kappa_2 = \mathtt{ref}$
7. *If* $\kappa_1 \sim_g \kappa_2$*and* $\kappa_1, \kappa_2 \neq \mathtt{tag}$ *then* $\kappa_1 \blacktriangleright \kappa \sim_g \kappa_2 \blacktriangleright \kappa$
8. *If* $\mathcal{A}(\kappa_1) \leq \kappa_2$ *and* $\kappa_1 \neq \kappa_2 \neq \mathtt{tag}$ *then* $\mathcal{A}(\kappa_1 \blacktriangleright \kappa) \leq \kappa_2 \blacktriangleright \kappa$

Lemma 5. *Capabilities are preserved along paths.*
If $\Delta, \chi, \alpha \vdash \iota : \kappa, p$ *and* $\Delta, \chi, \alpha \vdash \iota : \kappa', p$ *then* $\kappa = \kappa'$.

Proof. By induction over the structure of p. □

E. Soundness

The property central to any soundness argument is the preservation of the well-formed visibility property, $WFV(\Delta, \chi)$, and the well-formed temporaries property $WFT(\Delta, \chi, \alpha, i, \mathtt{e})$ 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.*

If

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

Then

4. $\mathcal{A}(\kappa_1\,\phi) \leq \kappa_3$
5. $\kappa_4 \lhd \mathcal{A}(\kappa_1\,\phi)$ *or* $\kappa_4 \lhd \kappa_3$

Proof. (4) follows from (1) and (2). For (5), if $\kappa_4 \lhd \kappa_3$, done. Otherwise, $\kappa_2' = \mathcal{A}(\kappa_1\,\phi)$. If $\kappa_4 = \mathtt{ref}$, then for all $\kappa_1\,\phi$, $\kappa_4 \lhd \kappa_2'$. If $\kappa_4 = \mathtt{trn}$, then $\kappa_2 \in \{\mathtt{iso}, \mathtt{trn}, \mathtt{val}, \mathtt{tag}\} \not\ni \kappa_3$. If $\kappa_2' \in \{\mathtt{iso}, \mathtt{trn}, \mathtt{val}\}$ then $\kappa_1\,\phi \in \{\mathtt{iso}\circ, \mathtt{trn}\circ, \mathtt{val}\}$ and $\mathtt{trn} \lhd \kappa_2'$. If $\kappa_2' = \mathtt{tag}$ then $\kappa_3 = \mathtt{tag}$, which contradicts $\kappa_4 \not\lhd \kappa_3$. If $\kappa_4 = \mathtt{iso}$, the same holds, except κ_2 cannot be $\mathtt{trn}$. □

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

$\mathtt{ET}' \sqsubseteq \mathtt{ET}$ iff $\mathtt{ET}' = \mathtt{ET}$, or $\mathtt{ET}' = \mathtt{S}\,\kappa\circ$ and $\mathtt{ET} = \mathtt{S}\,\kappa$

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

$\exists \mathtt{E}[\cdot], \mathtt{e}', \mathtt{f}, \overline{\mathtt{y}}, \overline{\mathtt{e}}, \mathtt{n}, \mathtt{S}$ such that

- $\mathtt{e} \equiv \mathtt{E}[\mathtt{x} = \mathtt{z}]$ or
- $\mathtt{e} \equiv \mathtt{E}[\mathtt{e}'.\mathtt{f} = \mathtt{z}]$ or
- $\mathtt{e} \equiv \mathtt{E}[\mathtt{e}'.\mathtt{n}(\overline{\mathtt{y}}, \mathtt{z}, \overline{\mathtt{e}})]$ or
- $\mathtt{e} \equiv \mathtt{E}[\mathtt{z}.\mathtt{n}(\overline{\mathtt{y}})]$ or
- $\mathtt{e} \equiv \mathtt{E}[\mathtt{S}.\mathtt{k}(\overline{\mathtt{y}}, \mathtt{z}, \overline{\mathtt{e}})]$

Lemma 7. *Inversion.*

If $\Gamma \vdash \mathtt{e} : \mathtt{ET}$ *then*

1. *If* $\mathtt{e} \equiv \mathtt{x}$ *then* $\Gamma(\mathtt{x}) \sqsubseteq \mathtt{ET}$
2. *If* $\mathtt{e} \equiv \mathtt{e}_1.\mathtt{f}$ *then* $\exists \mathtt{S}, \mathtt{S}', \kappa, \kappa'$ *such that* $\Gamma \vdash \mathtt{e}_1 : \mathtt{S}\,\kappa$ *and* $\mathcal{F}(\mathtt{S}, \mathtt{f}) = \mathtt{S}'\,\kappa'$ *and* $\mathtt{ET} = \mathtt{S}'\,\kappa \rhd \kappa'$.
3. *If* $\mathtt{e} \equiv \mathtt{null}$ *then* $\exists \mathtt{S}$ *such that* $\mathtt{S}\,\mathtt{iso}\circ \sqsubseteq \mathtt{ET}$
4. *If* $\mathtt{e} \equiv \mathtt{e}_1; \mathtt{e}_2$ *then* $\exists \mathtt{ET}_1$ *such that* $\Gamma \vdash \mathtt{e}_1 : \mathtt{ET}_1$ *and* $\Gamma \vdash \mathtt{e}_2 : \mathtt{ET}$
5. *If* $\mathtt{e} \equiv \mathtt{x} = \mathtt{e}_1$ *then* $\exists \mathtt{S}, \kappa, \kappa', \phi$ *such that* $\Gamma(\mathtt{x}) = \mathtt{S}\,\kappa$ *and* $\Gamma \vdash \mathtt{e}_1 : \mathtt{S}\,\kappa'\,\phi$ *and* $\mathcal{A}(\kappa'\,\phi) \leq \kappa$ *and* $\mathcal{U}(\mathtt{S}\,\kappa) \sqsubseteq \mathtt{ET}$
6. *If* $\mathtt{e} \equiv \mathtt{e}_1.\mathtt{f} = \mathtt{e}_2$ *then* $\exists \mathtt{S}_1, \mathtt{S}_2, \kappa_1, \kappa_2, \phi$ *such that* $\Gamma \vdash \mathtt{e}_1 : \mathtt{S}_1\,\kappa_1$ *and* $\Gamma \vdash \mathtt{e}_2 : \mathtt{S}_2\,\kappa_2\,\phi$ *and* $\mathcal{F}(\mathtt{S}_1, \mathtt{f}) = \mathtt{S}_2\,\kappa_3$ *and* $\mathcal{A}(\kappa_2\,\phi) \leq \kappa_3$, *either* $\kappa_1 \lhd \kappa_3$ *or* $\kappa_1 \lhd \mathcal{A}(\kappa_2\,\phi)$, *and* $\mathcal{U}(\mathtt{S}_2\,\kappa_1 \rhd \kappa_3) \sqsubseteq \mathtt{ET}$
7. *If* $\mathtt{e} \equiv \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}})$ *then* $\exists \mathtt{S}_0, \kappa_0, \kappa_0', \phi, \overline{\mathtt{T}}, \overline{\mathtt{ET}}, \mathtt{ET}'$ *such that* $\Gamma \vdash \mathtt{e}_0 : \mathtt{S}_0\,\kappa_0\,\phi$ *and* $\mathcal{A}(\kappa_0\,\phi) \leq \kappa_0'$ *and* $\mathcal{M}(\mathtt{S}_0, \mathtt{m}) = (\mathtt{S}_0\,\kappa_0', \overline{\mathtt{x} : \mathtt{T}}, \_, \mathtt{ET}')$ *and* $\Gamma \vdash \mathtt{e}_\mathtt{i} : \mathtt{ET}_\mathtt{i}$ *and* $\mathcal{A}(\mathtt{ET}_\mathtt{i}) \leq \mathtt{T}_\mathtt{i}$ *and* $\mathtt{ET}' \sqsubseteq \mathtt{ET}$
8. *If* $\mathtt{e} \equiv \mathtt{e}_0.\mathtt{b}(\overline{\mathtt{e}})$ *then* $\exists \mathtt{A}, \kappa_0, \kappa_0', \phi, \overline{\mathtt{T}}$ *such that* $\Gamma \vdash \mathtt{e}_0 : \mathtt{A}\,\kappa_0\,\phi$ *and* $\mathcal{A}(\kappa_0\,\phi) \leq \kappa_0'$ *and* $\mathcal{M}(\mathtt{A}, \mathtt{b}) = (\mathtt{A}\,\mathtt{ref}, \overline{\mathtt{x} : \mathtt{T}}, \_, \mathtt{A}\,\mathtt{tag})$ *and* $\mathit{sendable}(\mathtt{T}_\mathtt{i})$ *and* $\Gamma \vdash \mathtt{e}_\mathtt{i} : \mathtt{ET}_\mathtt{i}$*and* $\mathcal{A}(\mathtt{ET}_\mathtt{i}) \leq \mathtt{T}_\mathtt{i}$ *and* $\mathtt{A}\,\mathtt{tag} = \mathtt{ET}$
9. *If* $\mathtt{e} \equiv \mathtt{C}.\mathtt{k}(\overline{\mathtt{e}})$*then* $\exists \overline{\mathtt{ET}}, \overline{\mathtt{T}}$ *such that* $\mathcal{M}(\mathtt{C}, \mathtt{k}) = (\mathtt{C}\,\mathtt{ref}, \overline{\mathtt{x} : \mathtt{T}}, \_, \mathtt{C}\,\mathtt{ref}\circ)$ *and* $\Gamma \vdash \mathtt{e}_\mathtt{i} : \mathtt{ET}_\mathtt{i}$*and* $\mathcal{A}(\mathtt{ET}_\mathtt{i}) \leq \mathtt{T}_\mathtt{i}$ *and* $\mathtt{C}\,\mathtt{ref}\circ \sqsubseteq \mathtt{ET}$
10. *If* $\mathtt{e} \equiv \mathtt{A}.\mathtt{k}(\overline{\mathtt{e}})$*then* $\exists \overline{\mathtt{ET}}, \overline{\mathtt{T}}$ *such that* $\mathcal{M}(\mathtt{A}, \mathtt{k}) = (\mathtt{A}\,\mathtt{ref}, \overline{\mathtt{x} : \mathtt{T}}, \_, \mathtt{A}\,\mathtt{tag})$ *and* $\mathit{sendable}(\mathtt{T}_\mathtt{i})$ *and* $\Gamma \vdash \mathtt{e}_\mathtt{i} : \mathtt{ET}_\mathtt{i}$*and* $\mathcal{A}(\mathtt{ET}_\mathtt{i}) \leq \mathtt{T}_\mathtt{i}$ *and* $\mathtt{A}\,\mathtt{tag} = \mathtt{ET}$
11. *If* $\mathtt{e} \equiv \mathtt{recover}\,\mathtt{e}'$ *then* $\exists \mathtt{ET}'$ *such that* $\Gamma' = \Gamma \backslash \{\mathtt{x} \mid \neg \mathit{sendable}(\Gamma(\mathtt{x}))\}$ *and* $\Gamma' \vdash \mathtt{e}' : \mathtt{ET}'$ *and* $\mathcal{R}(\mathtt{ET}') \sqsubseteq \mathtt{ET}$

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

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

If

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

Then $\alpha = \alpha'$ *and either* $\kappa \sim_\ell \kappa'$ *or* $(i, \mathtt{z}) = (i', \mathtt{z}')$.

Proof. Assume that $\alpha \neq \alpha'$. Then, by $WFV.1$, $\kappa \sim_g \kappa'$. This implies $\kappa' = \mathtt{tag}$, which contradicts 5. Therefore, $\alpha = \alpha'$ and $\Delta, \chi, \alpha' \vdash \iota : \kappa, (i, \mathtt{z})$. If $\mathit{Stable}(\Delta, \alpha, (i, \mathtt{z}))$ then by $WFV.2$, either $\kappa \sim_\ell \kappa'$ (done) or $\chi, \alpha \vdash (i, \mathtt{z}) \sim (i', \mathtt{z}')$, which requires $(i, \mathtt{z}) = (i', \mathtt{z}')$ (done). If $\neg \mathit{Stable}(\Delta, \alpha, (i, \mathtt{z}))$ then $\mathtt{z} = \mathtt{t}$ and by $WFV.4$ either $(i, \mathtt{z}) = (i', \mathtt{z}')$ (done) or $\exists \iota', \kappa'', p', \overline{\mathtt{f}}$ such that $\kappa \leq \kappa''$ and $\kappa'' \in \{\mathtt{iso}, \mathtt{trn}\}$ and $(i', \mathtt{z}') = p' \cdot \overline{\mathtt{f}}$ and $\Delta, \chi, \alpha \vdash \iota' : \kappa'', p'$ and $\Delta, \chi, \iota' \vdash \iota : \kappa, (0, \mathtt{this}) \cdot \overline{\mathtt{f}}$, so $\overline{\mathtt{f}} = \epsilon$ and $p' = (i', \mathtt{z}')$ and $\iota = \iota'$. This gives us $\Delta, \chi, \iota \vdash \iota : \kappa, (0, \mathtt{this})$, which by the definition of visibility gives us $\kappa = \mathtt{ref}$, which contradicts (4) (done). □

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

If

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

Then

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

6. *If* $\kappa\,\phi = \mathtt{iso}$ *then* $\alpha = \alpha'$ *and either* $p = (i, \mathtt{t})$ *or* $\exists\iota', \kappa'', p', \overline{\mathtt{f}}$ *such that* $\kappa \leq \kappa''$ *and* $\kappa'' \in \{\mathtt{iso}, \mathtt{trn}\}$ *and* $p = p' \cdot \overline{\mathtt{f}}$ *and* $\Delta, \chi, \alpha \vdash \iota' : \kappa'', p'$ *and* $\Delta, \chi, \iota' \vdash \iota : \mathtt{iso}, \overline{\mathtt{f}}$.
7. *If* $\kappa\,\phi = \mathtt{trn}$ *then* $\alpha = \alpha'$ *and either* $p = (i, \mathtt{t})$ *or* $\kappa' = \mathtt{box}$ *or* $\exists\iota', p', \overline{\mathtt{f}}$ *such that* $p = p' \cdot \overline{\mathtt{f}}$ *and* $\Delta, \chi, \alpha \vdash \iota' : \mathtt{trn}, p'$ *and* $\Delta, \chi, \iota' \vdash \iota : \mathtt{trn}, \overline{\mathtt{f}}$.

Proof. (4) and (5) follow from lemma 8. (5) and (6) follow from lemma 8 and *WFV.4*. □

Lemma 10. *Aliasing and replaceability.*
If

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

Then $\Gamma' \vdash \mathtt{e} : \mathtt{ET}$

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

- If $\mathtt{e} \equiv \mathtt{x} = \mathtt{z}$ then, by lemma 7, we obtain $\Gamma(\mathtt{x}) = \mathtt{S}\,\kappa$ and $\Gamma(\mathtt{z}) = \mathtt{S}\,\kappa'\,\phi$ and $\phi \neq \circ$ and $\mathcal{A}(\kappa') \leq \kappa$. Therefore, we have $\mathcal{A}(\mathcal{A}(\kappa')) \leq \kappa$ and so $\Gamma' \vdash \mathtt{x} = \mathtt{z} : \mathtt{ET}$.
- If $\mathtt{e} \equiv \mathtt{e}'.\mathtt{f} = \mathtt{z}$ then, by lemma 7, we obtain $\Gamma \vdash \mathtt{e}' : \mathtt{S}\,\kappa$ and $\mathcal{F}(\mathtt{S}, \mathtt{f}) = \mathtt{S}'\,\kappa'$ and $\Gamma(\mathtt{z}) = \mathtt{S}'\,\kappa''\,\phi$ and $\phi \neq \circ$ and $\mathcal{A}(\kappa'') \leq \kappa'$. Therefore, we have $\mathcal{A}(\mathcal{A}(\kappa'')) \leq \kappa'$ and so $\Gamma' \vdash \mathtt{e}'.\mathtt{f} = \mathtt{z} : \mathtt{ET}$.
- If $\mathtt{e} \equiv \mathtt{e}'.\mathtt{n}(\overline{\mathtt{y}}, \mathtt{z}, \overline{\mathtt{e}})$ then, by lemma 7, we obtain $\Gamma \vdash \mathtt{e}' : \mathtt{S}\,\kappa$ and $\mathcal{M}(\mathtt{S}, \mathtt{n}) = (\_, \overline{\mathtt{x} : \mathtt{S}\,\kappa}, \_, \_)$ and $\Gamma(\mathtt{z}) = \mathtt{S}_\mathtt{i}\,\kappa'_\mathtt{i}\,\phi$ and and $\phi \neq \circ$ and $\mathcal{A}(\kappa'_i) \leq \kappa_i$. Therefore, we have $\mathcal{A}(\mathcal{A}(\kappa'_i)) \leq \kappa_i$ and so $\Gamma' \vdash \mathtt{e}'.\mathtt{n}(\overline{\mathtt{y}}, \mathtt{z}, \overline{\mathtt{e}}) : \mathtt{ET}$.
- If $\mathtt{e} \equiv \mathtt{z}.\mathtt{n}(\overline{\mathtt{y}})$ then, by lemma 7, we obtain $\Gamma(\mathtt{z}) = \mathtt{S}\,\kappa\,\phi$ and $\phi \neq \circ$ and $\mathcal{M}(\mathtt{S}, \mathtt{n}) = (\mathtt{S}\,\kappa')$ and $\mathcal{A}(\kappa) \leq (\kappa')$. Therefore, we have $\mathcal{A}(\mathcal{A}(\kappa)) \leq \kappa'$ and so $\Gamma' \vdash \mathtt{z}.\mathtt{n}(\overline{\mathtt{y}}) : \mathtt{ET}$.
- If $\mathtt{e} \equiv \mathtt{S}.\mathtt{k}(\overline{\mathtt{y}}, \mathtt{z}, \overline{\mathtt{e}})$then, by lemma 7, we obtain $\mathcal{M}(\mathtt{S}, \mathtt{k}) = (\_, \overline{\mathtt{x} : \mathtt{S}\,\kappa}, \_, \_)$ and $\Gamma(\mathtt{z}) = \mathtt{S}_\mathtt{i}\,\kappa'_\mathtt{i}\,\phi$ and and $\phi \neq \circ$ and $\mathcal{A}(\kappa'_i) \leq \kappa_i$. Therefore, we have $\mathcal{A}(\mathcal{A}(\kappa'_i)) \leq \kappa_i$ and so $\Gamma' \vdash \mathtt{S}.\mathtt{k}(\overline{\mathtt{y}}, \mathtt{z}, \overline{\mathtt{e}}) : \mathtt{ET}$.

For the inductive step, if $\mathtt{e} \equiv \mathtt{E}[\mathtt{e}']$and $\mathtt{z}$ is aliased in $\mathtt{e}$, then, by lemma 2, we obtain that $\exists\mathtt{ET}', \mathtt{y} \notin \Gamma$ such that $\Gamma \vdash \mathtt{e}' : \mathtt{ET}'$ and $\Gamma[\mathtt{y} \mapsto \mathtt{ET}'] \vdash \mathtt{E}[\mathtt{y}] : \mathtt{ET}$, and so $\Gamma' \vdash \mathtt{e}' : \mathtt{ET}'$. Therefore, by lemma 2, we obtain $\Gamma' \vdash \mathtt{E}[\mathtt{e}'] : \mathtt{ET}$. □

Lemma 11. *Origins of temporary identifiers.*
If

1. $\mathtt{z}$ *appears once in expression* $\mathtt{e}$
2. $\mathtt{z}$ *is not aliased in* $\mathtt{e}$

Then $\exists\mathtt{E}'$ *such that*

3. $\mathtt{e} \equiv \mathtt{E}'[\mathtt{z}.\mathtt{f}]$*, or*
4. $\mathtt{e} \equiv \mathtt{E}'[\mathtt{z}.\mathtt{f} = \mathtt{e}']$*, or*
5. $\mathtt{e} \equiv \mathtt{E}'[\mathtt{recover}\,\mathtt{z}]$*, or*

Proof. By application of definition 6. □

Lemma 12. *If* $\Gamma, \mathtt{x} : \mathtt{T}_1 \vdash \mathtt{e} : \mathtt{ET}_1$ *and* $\mathcal{A}(\mathtt{T}_2) \leq \mathtt{T}_1$ *then* $\exists\mathtt{ET}_2.\Gamma, \mathtt{x} : \mathtt{T}_2 \vdash \mathtt{e} : \mathtt{ET}_2$ *and* $\mathtt{ET}_1 = \mathtt{ET}_2$ *or* $\mathcal{A}(\mathtt{ET}_2) \leq \mathtt{ET}_1$

Proof. By structural induction on the typing and lemma 3. □

Lemma 13. *New paths through field read.*
If

1. $\chi(\alpha, i, \mathtt{z}) = \iota$
2. $\Delta(\alpha, i, \mathtt{z}) = \mathtt{S}\,\kappa\,\phi$
3. $\chi(\iota, \mathtt{f}) = \iota'$ *and* $\mathcal{F}(\mathtt{S}, \mathtt{f}) = \mathtt{S}'\,\kappa'$
4. $\mathtt{T} = \bot$ *if* $\mathtt{z} = \mathtt{t}'$, $\mathtt{S}\,\kappa$ *otherwise*
5. $\Delta' = \Delta[(\alpha, i, \mathtt{z}) \mapsto \mathtt{T}, (\alpha, i, \mathtt{t}) \mapsto \mathtt{S}'\,\kappa \triangleright \kappa')]$
6. $\mathcal{T}(\Delta(\alpha, i)) \subseteq \{\mathtt{z}\}$

Then

7. $\forall\alpha', \iota'', \kappa'', p'$ *if* $\Delta', \chi, \alpha' \vdash \iota'', \kappa'', p'$ *then*
   (a) $\Delta, \chi, \alpha' \vdash \iota'' : \kappa'', p'$ *or*
   (b) $\alpha' = \alpha$ *and* $\exists\overline{\mathtt{f}}, \overline{\kappa}$ *such that*
      i. $p' = (i, \mathtt{t}) \cdot \overline{\mathtt{f}}$
      ii. $\kappa'' = \kappa \triangleright \kappa'\overline{\blacktriangleright \kappa}$
      iii. $\Delta, \chi, \alpha \vdash \iota'' : \kappa \blacktriangleright \kappa'\overline{\blacktriangleright \kappa}, (i, \mathtt{z}) \cdot \mathtt{f} \cdot \overline{\mathtt{f}}$
8. *If* $WFV(\Delta, \chi)$ *then* $WFV(\Delta', \chi)$ *and* $\mathcal{T}(\Delta'(\alpha, i)) \subseteq \{\mathtt{t}\}$

Lemma 14. *New paths through local assignment.*
If

1. $\chi(\alpha, i, \mathtt{z}) = \iota$ *and* $\chi(\alpha, i, \mathtt{x}) = \iota'$ *and* $\mathtt{t} \notin \chi(\alpha, i)$
2. $\Delta(\alpha, i, \mathtt{z}) = \mathtt{S}\,\kappa\,\phi$ *and* $\Delta(\alpha, i, \mathtt{x}) = \mathtt{S}\,\kappa'$
3. $\chi' = \chi[(\alpha, i, \mathtt{x}) \mapsto \iota, (\alpha, i, \mathtt{t}) \mapsto \iota']$
4. $\mathtt{T} = \bot$ *if* $\mathtt{z} = \mathtt{t}'$, $\mathtt{S}\,\kappa$ *otherwise*
5. $\Delta' = \Delta[(\alpha, i, \mathtt{z}) \mapsto \mathtt{T}, (\alpha, i, \mathtt{t}) \mapsto \mathcal{U}(\mathtt{S}\,\kappa')]$
6. $\mathcal{T}(\Delta(\alpha, i)) \subseteq \{\mathtt{z}\}$

Then

7. $\forall\alpha', \iota'', \kappa'', p$ *if* $\Delta', \chi', \alpha' \vdash \iota'' : \kappa'', p$ *then*
   (a) $\Delta, \chi, \alpha' \vdash \iota'' : \kappa'', p$ *or*
   (b) $\alpha = \alpha'$ *and* $\exists\overline{\mathtt{f}}, \overline{\kappa}$ *such that*
      i. $p = (i, \mathtt{x}) \cdot \overline{\mathtt{f}}$ *and* $\kappa'' = \kappa'\overline{\blacktriangleright \kappa}$ *and* $\Delta, \chi, \alpha \vdash \iota'' : \kappa\overline{\blacktriangleright \kappa}, (i, \mathtt{z}) \cdot \overline{\mathtt{f}}$*, or*
      ii. $p = (i \cdot \mathtt{t}) \cdot \overline{\mathtt{f}}$ *and* $\kappa'' = \kappa'\overline{\blacktriangleright \kappa}$ *and* $\Delta, \chi, \alpha \vdash \iota'' : \kappa'\overline{\blacktriangleright \kappa}, (i, \mathtt{x}) \cdot \overline{\mathtt{f}}$
8. *If* $\mathcal{A}(\kappa\,\phi) \leq \kappa'$ *and* $WFV(\Delta, \chi)$ *then* $WFV(\Delta', \chi')$ *and* $\mathcal{T}(\Delta'(\alpha, i)) \subseteq \{\mathtt{t}\}$

Lemma 15. *New paths through field assignment.*
If

1. $\chi(\alpha, i, \mathtt{z}) = \iota$ *and* $\chi(\alpha, i, \mathtt{z}') = \iota'$

2. $\Delta(\alpha, i, \mathtt{z}) = \mathtt{S}\,\kappa\,\phi$ *and* $\Delta(\alpha, i, \mathtt{z}') = \mathtt{S}'\,\kappa'\,\phi'$
3. $\chi(\iota, \mathtt{f}) = \iota''$ *and* $\mathcal{F}(\mathtt{S}, \mathtt{f}) = \mathtt{S}'\,\kappa''$
4. $\chi' = \chi[(\iota, \mathtt{f}) \mapsto \iota', (\alpha, i, \mathtt{t}) \mapsto \iota'']$
5. $\mathtt{T} = \bot$ *if* $\mathtt{z} = \mathtt{t}'$, $\mathtt{S}\,\kappa$ *otherwise*
6. $\mathtt{T}' = \bot$ *if* $\mathtt{z}' = \mathtt{t}''$, $\mathtt{S}'\,\kappa'$ *otherwise*
7. $\Delta' = \Delta[(\alpha, i, \mathtt{z}) \mapsto \mathtt{T}, (\alpha, i, \mathtt{z}') \mapsto \mathtt{T}', (\alpha, i, \mathtt{t}) \mapsto \mathcal{U}(\mathtt{S}'\,\kappa \triangleright \kappa'')]$
8. $\mathcal{T}(\Delta(\alpha, i)) \subseteq \{\mathtt{z}, \mathtt{z}'\}$

Then

9. $\forall \alpha', \iota''', \kappa''', p'$ *if* $\Delta', \chi', \alpha' \vdash \iota''' : \kappa''', p'$ *then*
   (a) $\Delta, \chi, \alpha' \vdash \iota''' : \kappa''', p'$ *or*
   (b) $\alpha' = \alpha$ *and* $\exists \overline{\mathtt{f}}, \overline{\kappa}$ *such that*
      i. $\kappa''' = \kappa \blacktriangleright \kappa''\overline{\blacktriangleright \kappa}$ *and* $p' = (i, \mathtt{z}) \cdot \mathtt{f} \cdot \overline{\mathtt{f}}$ *and* $\Delta, \chi, \alpha \vdash \iota''' : \kappa'\overline{\blacktriangleright \kappa}, (i, \mathtt{z}') \cdot \overline{\mathtt{f}}$, *or*
      ii. $\kappa''' = \mathcal{U}(\kappa \triangleright \kappa'')\overline{\blacktriangleright \kappa}$ *and* $p' = (i, \mathtt{t}) \cdot \overline{\mathtt{f}}$ *and* $\Delta, \chi, \alpha \vdash \iota''' : \kappa \blacktriangleright \kappa''\overline{\blacktriangleright \kappa}, (i, \mathtt{z}) \cdot \mathtt{f} \cdot \overline{\mathtt{f}}$, *or*
      iii. $\exists \kappa'''', p \neq (i, \mathtt{z})$ *such that* $\kappa''' = \kappa'''' \blacktriangleright \kappa''\overline{\blacktriangleright \kappa}$ *and* $p' = p \cdot \mathtt{f} \cdot \overline{\mathtt{f}}$ *and* $\Delta, \chi, \alpha \vdash \iota : \kappa'''', p$
10. *If* $\mathcal{A}(\kappa'\,\phi') \leq \kappa''$ *and* ($\kappa \triangleleft \kappa'$ *or* $\kappa \triangleleft \kappa''$) *and* $WFV(\Delta, \chi)$ *then* $WFV(\Delta', \chi')$ *and* $\mathcal{T}(\Delta'(\alpha, i)) = \emptyset$

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

1. $\chi(\alpha, i, \mathtt{z}) = \iota$ *and* $\Delta(\alpha, i, \mathtt{z}) = \mathtt{S}\,\kappa\,\phi$
2. $\chi' = \chi[(\alpha', -j, \mathtt{x}) \mapsto \iota]$
3. $\mathtt{T} = \bot$ *if* $\mathtt{z} = \mathtt{t}$, $\mathtt{S}\,\kappa$ *otherwise*
4. $\Delta' = \Delta[(\alpha, i, \mathtt{z}) \mapsto \mathtt{T}, (\alpha', -j, \mathtt{x}) \mapsto \mathtt{S}\,\kappa']$
5. $\mathcal{T}(\Delta(\alpha, i)) \subseteq \{\mathtt{z}\}$

Then

6. $\forall \alpha'', \iota'', \kappa'', p$ *if* $\Delta', \chi', \alpha'' \vdash \iota'' : \kappa'', p$ *then*
   (a) $\Delta, \chi, \alpha'' \vdash \iota'' : \kappa'', p$ *or*
   (b) $\alpha'' = \alpha'$ *and* $\exists \overline{\mathtt{f}}, \overline{\kappa}$ *such that*
      i. $p = (-j, \mathtt{x}) \cdot \overline{\mathtt{f}}$
      ii. $\kappa'' = \kappa'\overline{\blacktriangleright \kappa}$
      iii. $\Delta, \chi, \alpha \vdash \iota'' : \kappa\overline{\blacktriangleright \kappa}, (i, \mathtt{z}) \cdot \overline{\mathtt{f}}$
7. *If* $\mathcal{A}(\kappa\,\phi) \leq \kappa'$ *and* $sendable(\kappa')$ *and* $WFV(\Delta, \chi)$ *then* $WFV(\Delta', \chi')$ *and* $\mathcal{T}(\Delta'(\alpha, i)) = \emptyset$

E.2 Preservation of well-formedness

Lemma 17. *Type preservation on same frame.*
For all heaps χ, *actors* α, *global type environments* Δ, *frames* φ, *stacks* σ *and expressions* $\mathtt{e}$, *if*

1. $\chi(\alpha) = (\_, \_, \_, \alpha \cdot \overline{\varphi} \cdot \varphi, \mathtt{E}[\mathtt{e}])$ *and* $|\overline{\varphi}| = i - 1$
2. $\chi, \alpha \cdot \overline{\varphi} \cdot \varphi, \mathtt{e} \rightsquigarrow \chi'', \alpha \cdot \overline{\varphi} \cdot \varphi', \mathtt{e}'$
3. $\chi' = \chi''[\alpha \mapsto (\alpha \cdot \overline{\varphi} \cdot \varphi, \mathtt{E}[\mathtt{e}'])]$
4. $\Delta(\alpha, i) \vdash \mathtt{e} : \mathtt{ET}$
5. $\Delta \vdash \chi\diamond$

Then $\exists \Delta'$ *such that*

1. $\Delta'(\alpha, i) \vdash \mathtt{e}' : \mathtt{ET}$
2. $\Delta' \vdash \chi'\diamond$

Lemma 18. *Type preservation for method call.*
For all heaps χ *and actors* α, *if*

1. $\chi(\alpha) = (\_, \_, \_, \sigma \cdot \varphi, \mathtt{E}[\mathtt{e}])$
2. $\chi, \sigma \cdot \varphi, \mathtt{e} \rightsquigarrow \chi'', \sigma \cdot \varphi \cdot \varphi', \mathtt{e}'$
3. $\chi' = \chi''[\alpha \mapsto (\sigma \cdot \varphi \cdot \varphi', \mathtt{E}[\mathtt{e}'])]$
4. $\Delta \vdash \chi\diamond$

Then $\exists \Delta'$ *such that* $\Delta' \vdash \chi'\diamond$

Lemma 19. *Type preservation upon method return*
For all heaps χ *and actors* α, *if*

1. $\chi(\alpha) = (\_, \_, \_, \sigma \cdot \varphi \cdot \varphi', \mathtt{z})$
2. $\mathtt{t} \notin \varphi$ *and* $\varphi'' = \varphi[\mathtt{t} \mapsto \varphi'(\mathtt{z})]$
3. $\varphi' = (\_, \_, \mathtt{E}[\cdot])$
4. $\chi' = \chi[\alpha \mapsto (\sigma \cdot \varphi'', \mathtt{E}[\mathtt{t}])]$
5. $\Delta \vdash \chi\diamond$

Then $\exists \Delta'$ *such that* $\Delta' \vdash \chi'\diamond$

Lemma 20. *Type preservation upon message handling.*
For all heaps χ *and actors* α, *if*

1. $\chi(\alpha) = (\mathtt{A}, fs, (\mathtt{n} \cdot \bar{v}) \cdot \overline{\mu}, \alpha, \epsilon)$
2. $\mathcal{M}(\mathtt{A}, \mathtt{n}) = (\_, \overline{\mathtt{x} : \mathtt{T}}, \mathtt{e}, \_)$
3. $\varphi = (\mathtt{n}, [\mathtt{this} \mapsto \alpha, \bar{\mathtt{x}} \mapsto \bar{v}], \cdot)$
4. $\chi' = \chi[\alpha \mapsto (\mathtt{A}, fs, \overline{\mu}, (\alpha \cdot \varphi), \mathtt{e})]$
5. $\Delta \vdash \chi\diamond$

Then $\exists \Delta'$ *such that* $\Delta' \vdash \chi'\diamond$

F. GUPS benchmark source code

```
use "options"
use "time"
use "collections"

class Config
  var logtable: U64 = 20
  var iterate: U64 = 10000
  var logchunk: U64 = 10
  var logactors: U64 = 2

  fun ref apply(env: Env): Bool =>
    var options = Options(env)

    options
      .add("logtable", "l", None, I64Argument)
      .add("iterate", "i", None, I64Argument)
      .add("chunk", "c", None, I64Argument)
      .add("actors", "a", None, I64Argument)

    for option in options do
      match option
      | ("table", var arg: I64) => logtable = arg.u64()
      | ("iterate", var arg: I64) => iterate = arg.u64()
      | ("chunk", var arg: I64) => logchunk = arg.u64()
      | ("actors", var arg: I64) => logactors = arg.u64()
      | ParseError =>
        env.out.print(
          """
          gups_opt [OPTIONS]
            --table     N   log2 of the total table size.
                  Defaults to 20.
            --iterate   N   number of iterations.
                 Defaults to 10000.
```

```
          --chunk     N   log2 of the chunk size. Defaults to 10.
          --actors    N   log2 of the actor count. Defaults to 2.
        """
        )
      return false
    end
  end

  env.out.print(
    "logtable: " + logtable.string() +
    "\niterate: " + iterate.string() +
    "\nlogchunk: " + logchunk.string() +
    "\nlogactors: " + logactors.string()
    )
  true

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
    end

  be confirm() =>
    _confirm = _confirm + 1

    if _confirm == _actors.size() then
      let elapsed = (Time.nanos() - _start).f64()
      let gups = _updates.f64() / elapsed

      _env.out.print(
        "Time: " + (elapsed / 1e9).string() +
        "\nGUPS: " + gups.string()
        )
    end

actor Updater
  let _main: Main
  let _index: U64
  let _updaters: U64
  let _chunk: U64
  let _mask: U64
  let _loglocal: U64

  let _output: Array[Array[U64] iso]
  let _reuse: List[Array[U64] iso] = List[Array[U64] iso]
  var _others: (Array[Updater] val | None) = None
  var _table: Array[U64]
  var _rand: U64

  new create(main:Main, updaters: U64, index: U64, loglocal: U64, chunk: U64,
    seed: U64)
  =>
    _main = main
    _index = index
    _updaters = updaters
    _chunk = chunk
    _mask = updaters - 1
    _loglocal = loglocal

    _rand = PolyRand.seed(seed)
    _output = _output.create(updaters)

    let size = 1 << loglocal
    _table = Array[U64].undefined(size)

    var offset = index * size

    try
      for i in Range[U64](0, size) do
        _table(i) = i + offset
      end
    end

  be start(others: Array[Updater] val, iterate: U64) =>
    _others = others
    iteration(iterate)

  be apply(iterate: U64) =>
    iteration(iterate)

  fun ref iteration(iterate: U64) =>
    let chk = _chunk

    for i in Range(0, _updaters) do
      _output.push(
        try
          _reuse.pop()
        else
          recover Array[U64](chk) end
        end
        )
    end

    for i in Range(0, _chunk) do
      var datum = _rand = PolyRand(_rand)
      var updater = (datum >> _loglocal) and _mask

      try
        if updater == _index then
          _table(i) = _table(i) xor datum
        else
          _output(updater).push(datum)
        end
      end
    end

    try
      let to = _others as Array[Updater] val

      repeat
        let data = _output.pop()

        if data.size() > 0 then
```

```
          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
    end
    r

  fun _poly(): U64 => 7

  fun _period(): U64 => 1317624576693539401
```