Deny Capabilities for Safe, Fast Actors

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

Abstract

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

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Concurrent, distributed, and parallel languages

Keywords actors; message passing; concurrency; type systems; capabilities

1. Introduction

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

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

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

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

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

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

Other approaches have combined actors with data-race freedom, such as minimal ownership for active objects [13], capabilities for uniqueness and borrowing in Scala [22], and Kilim [29]. However, various useful patterns have not been supported, e.g. traversing and modifying an isolated data structure, or updating an object and then sending it in a message while keeping read access to it. By taking a more fundamental view of reference capabilities, we were able to develop a more flexible type system that supports such 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 reference capability types, `tag` and `trn` (*transition*). A `tag` reference capability allows identity comparison and *asynchronous* method call, but does not allow reading from or writing to the reference. We type actors as `tag`, which allows them to be integrated into the object type system and passed in messages. A `trn` reference capability is a new form of uniqueness, *write uniqueness*, that describes objects that can only be written to through a single reference, but can be read from through many aliases.

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

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

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

Contributions In this work, we present:

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

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

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

2. Reference Capabilities as Deny Properties

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

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

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

In addition, when the local deny properties and the global deny properties of a reference are the same, the reference can be safely sent as an argument to an asynchronous method call to another actor, i.e. it is *sendable*. In other words, when the local alias deny properties are the same as the global

[1] We use the term *reference* to mean the path currently being considered, and *alias* to mean any other path to the same object.

	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. Reference capability matrix. Those in *italics* are sendable.

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 alias can be used by other actors to write to that object. This means that other actors may be able to read the object, and aliases in the same actor may be able to write to it (although not both: if the actor can write to the object, other actors cannot read from it). Using `box` for immutability allows a program to enforce read-only behaviour, similar to `const` in C/C++. For example:

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

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

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

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

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

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

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

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

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

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

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

Here, by passing an `iso` reference, a `Dataflow` actor can mutate the `list` before sending it to the `flow` actor.

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

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

Our type system introduces the concept of *unaliased types,* annotated with $\circ$, in order to type values for which an alias has been removed. Here, the consume produces a `List iso`$\circ$ 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 aliases 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,
      flow: Dataflow tag) =>
    list1.next = consume list2
    flow.step(consume list1)
```

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 reference capability in the future, in order to share it with another actor.

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

Figure 1. Syntax

$\mathtt{C}$	$\in$	*ClassID*	$\mathtt{k}$	$\in$	*CtorID*
$\mathtt{A}$	$\in$	*ActorID*	$\mathtt{m}$	$\in$	*FuncID*
$\mathtt{f}$	$\in$	*FieldID*	$\mathtt{b}$	$\in$	*BehvID*
$\mathtt{this}, \mathtt{x}$	$\in$	*SourceID*	$\mathtt{n}$	$\in$	*CtorID* $\cup$ *BehvID*
$\mathtt{t}$	$\in$	*TempID*	$\mathtt{y}, \mathtt{z}$	$\in$	*LocalID*

Figure 2. Identifiers

```
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[2] 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`$\circ$, which is aliased as a `Booking trn`, which is a subtype of `Booking val` and can be shared with the `Accountant` actor. Without a *write unique* type, namely `trn`, this would require copying the `Booking`.

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

3. Syntax

In fig. 1 we present the syntax, which is a subset of Pony. We support actors in the mould of active objects, introduced with the keyword `actor`. These can have both synchron-

[2] In this example, we are using generic types and default reference capabilities (`ref` for objects and `tag` for actors). While the full Pony language supports these, we will not formalise them here.

ous 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 constructors, functions, and behaviours. The syntax of expressions is standard with the exception of the `recover` keyword - more in sec. 4.

The novel element of the syntax is the inclusion of *reference capability annotations* κ on types and functions, where:
$\kappa \in \{\mathtt{iso}, \mathtt{trn}, \mathtt{ref}, \mathtt{val}, \mathtt{box}, \mathtt{tag}\}$

These reference capabilities are the foundation of our type system.

Types consist of a class or actor identifier `S` followed by a reference capability κ. In addition, extended types `ET` can be *unaliased*, $\circ$. 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`.

4. Type System

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

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

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

Because unaliasing is of importance only when the underlying reference capability is `iso`, `trn` or `ref`, we have defined the unaliasing operation $\mathcal{U}$, 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. While the full Pony language has no *null*, we use it here to simplify the modelling of `consume x`, which is treated as $(\mathtt{x} = \mathtt{null})$.

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 $\mathcal{A}(\kappa')$ does not locally deny κ'.

Definition 1. Aliasing and unaliasing.

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

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

Reference capabilities at field read When reading a field `f` from an object ι we obtain a temporary. The reference capability of this temporary must be a combination of κ, the reference capability of the path leading to ι, and κ', the reference capability with which ι sees the field. We express this through the operator $\triangleright$, defined in table 2. When reading a field through an origin, the result must not violate the deny properties of either the origin or the field. For example, reading a `ref` field from an `iso` reference returns `tag` - thus we do not violate the deny properties of the origin or the field itself.

Storing a reference into a field of an object ι is legal if the type of the reference is both a subtype of the type of

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

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

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

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

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

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

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

Figure 3. Expression typing

$\kappa \triangleright \kappa'$			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 2. 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 3. Safe to write.

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

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

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

Figure 4. Sub-types and sendable types.

the field and also *safe to write* into the origin. The relation $\kappa \triangleleft \kappa'$, as defined in table 3, 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. Therefore, all entries for `val`, `box` and `tag` are empty. Moreover, only `iso`, `val` or `tag` references may be stored into an `iso` origin; all other writes would violate the region introduced by the `iso` origin.

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

Definition 2. Reference capability recovery

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

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

Figure 5. Runtime entities

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

The treatment of actors Actors introduce the question of who may read or update the actor's fields, the possibility of synchronous calls on actors, and the type required for asynchronous calls.

Field read and write requires that the actor should see itself as a `ref`. As a result, any other actor will see it as `tag`. Therefore no other actor except the current one will be allowed to observe an actor's fields - a nice consequence of the type system.

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

5. Operational Semantics

The operational semantics has the shape $\chi \rightarrow \chi'$, where 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. 5. We use some shorthand notation for clarity - more in app. fig. 11.

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 $\chi, \sigma, \texttt{e} \rightsquigarrow \chi', \sigma', \texttt{e}'$ expresses local execution within a *single* actor. $\mathcal{M}$ and $\mathcal{F}$ return method and field declarations. They are defined in the appendix.

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

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

SYNC and RETURN describe synchronous method call and return. In SYNC, method `m` is called on object or actor ι. The method parameters $\overline{\texttt{x}}$ 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[3]. A new frame is pushed on the stack in the same way as for SYNC. In ATOR, instead of pushing a new frame on the stack, the new actor's queue is initialised with a constructor message containing the constructor identifier `k` and the arguments. The first local execution rule for a new actor will be BEHAVE, which will execute the body of the constructor `k`.

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

[3] This is a simplification. In Pony, we support object initialisation, and have no null values.

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

$$\frac{\texttt{t} \notin \varphi \quad \varphi' = \varphi[\texttt{t} \mapsto \mathit{null}]}{\chi, \sigma \cdot \varphi, \texttt{null} \rightsquigarrow \chi, \sigma \cdot \varphi', \texttt{t}} \ \textsc{Null} \qquad \frac{}{\chi, \sigma, \texttt{z}; \texttt{e} \rightsquigarrow \chi, \sigma, \texttt{e}} \ \textsc{Seq}$$

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

$$\frac{\begin{array}{c}\iota = \varphi(\texttt{z}) \quad \mathcal{M}(\chi(\iota) \downarrow_1, \texttt{m}) = (\_, \overline{\texttt{x}} : \_, \texttt{e}, \_) \\ \varphi' = (\texttt{m}, [\texttt{this} \mapsto \iota, \overline{\texttt{x}} \mapsto \varphi(\overline{\texttt{y}})], \texttt{E}[\cdot])\end{array}}{\chi, \sigma \cdot \varphi, \texttt{E}[\texttt{z.m}(\overline{\texttt{y}})] \rightsquigarrow \chi, \sigma \cdot \varphi \cdot \varphi', \texttt{e}} \ \textsc{Sync} \qquad \frac{\begin{array}{c}\texttt{t} \notin \varphi \quad \iota = \varphi'(\texttt{z}) \\ \varphi' \downarrow_3 = \texttt{E}[\cdot] \quad \varphi'' = \varphi[\texttt{t} \mapsto \iota]\end{array}}{\chi, \sigma \cdot \varphi \cdot \varphi', \texttt{z} \rightsquigarrow \chi, \sigma \cdot \varphi'', \texttt{E}[\texttt{t}]} \ \textsc{Return}$$

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

$$\frac{\begin{array}{c}\omega \notin \mathit{dom}(\chi) \quad \overline{\texttt{f}} = \mathcal{F}s(\texttt{C}) \\ \mathcal{M}(\texttt{C}, \texttt{k}) = (\_, \overline{\texttt{x}} : \_, \texttt{e}, \_) \\ \chi' = \chi[\omega \mapsto (\texttt{C}, \overline{\texttt{f}} \mapsto \mathit{null})] \\ \varphi' = (\texttt{k}, [\texttt{this} \mapsto \omega, \overline{\texttt{x}} \mapsto \varphi(\overline{\texttt{y}})], \texttt{E}[\cdot])\end{array}}{\chi, \sigma \cdot \varphi, \texttt{E}[\texttt{C.k}(\overline{\texttt{y}})] \rightsquigarrow \chi', \sigma \cdot \varphi \cdot \varphi', \texttt{e}} \ \textsc{Ctor} \qquad \frac{\begin{array}{c}\alpha \notin \mathit{dom}(\chi) \quad \overline{\texttt{f}} = \mathcal{F}s(\texttt{A}) \\ \texttt{t} \notin \varphi \quad \varphi' = \varphi[\texttt{t} \mapsto \alpha] \\ \chi' = \chi[\alpha \mapsto (\texttt{A}, \overline{\texttt{f}} \mapsto \mathit{null}, (\texttt{k}, \varphi(\overline{\texttt{y}}), \alpha, \varepsilon)]\end{array}}{\chi, \sigma \cdot \varphi, \texttt{A.k}(\overline{\texttt{y}}) \rightsquigarrow \chi', \sigma \cdot \varphi', \texttt{t}} \ \textsc{Ator}$$

$$\frac{\chi, \sigma, \texttt{e} \rightsquigarrow \chi', \sigma', \texttt{e}'}{\chi, \sigma, \texttt{recover}\,\texttt{e} \rightsquigarrow \chi', \sigma', \texttt{recover}\,\texttt{e}'} \ \textsc{Rec1} \qquad \frac{\texttt{t} \notin \varphi \quad \varphi' = \varphi[\texttt{t} \mapsto \varphi(\texttt{z})]}{\chi, \sigma, \texttt{recover}\,\texttt{z} \rightsquigarrow \chi, \sigma, \texttt{t}} \ \textsc{Rec2}$$

$$\frac{}{\chi, \sigma, \texttt{consume}\,\texttt{x} \rightsquigarrow \chi, \sigma, \texttt{x} = \texttt{null}} \ \textsc{Consume} \qquad \frac{\chi, \chi(\alpha) \downarrow_4, \chi(\alpha) \downarrow_5 \rightsquigarrow \chi', \sigma, \texttt{e}}{\chi \to \chi'[\alpha \mapsto (\sigma, \texttt{e})]} \ \textsc{Global}$$

Figure 6. Execution.

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

6. Related Work

Linear types [31] provide the basis for uniqueness type systems. The insight that a type that is usable only once allows for mutation in a pure functional language leads directly to using linearity for concurrency-safe mutation [5]. A combination of unique pointers and ownership types [14] is used in 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] and *separate uniqueness [22]* allows for richer and more complex data structures to be simply encoded while maintaining all of the useful properties of linear types.

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

In Kilim [29], tree-structured messages are used to 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 [30], access permissions are combined with data flow analysis for implicit concurrency, which is in some sense the inverse of actor-model concurrency.

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

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

In Rust [24], atomic reference counts, mutexes, allow properties, and ownership types are combined to achieve data race freedom. The use of both run-time and compile-

	Our Work	Gordon	Æminium	DPJ	Kilim	Haller	Scala	Erlang	Rust
Zero-copy	√	√	√	√	√	√	√		√
Data-race free	√	√	√	√	√[4]	√[5]		√	√
Statically data-race free	√	√	√	√	√	√			[6]
Non-tree messages	√	√				√	√	√	√
Read unique (`iso`)	√	√	√		√	√			
Write unique (`trn`)	√								
Mutability (`ref`)	√	√	√	√	√	√	√		√
Immutability (`val`)	√	√	√		√		[7]	√	√
Cyclic immutability	√	√							
Identity (`tag`)	√		[8]						
Destructive read	√	√			√	√			√
Recovery	√	√							
Using uniques ($\texttt{iso} \triangleright \texttt{x}$)	√								
Actors	√				√	√	√	√	

Table 4. Feature comparison.

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 4, we summarise some features of our work and compare with those in Gordon et al. [19], Æminium [30], Deterministic Parallel Java [6], Kilim [29], Haller and Odersky [22], Scala, Erlang, and Rust [24].

7. Implementation and Benchmarking

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

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

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

In fig. 7, we show actor creation performance when creating an interconnected tree of actors that cannot be collec-

[4] Kilim messages are data-race free but the rest of Java is not.

[5] The proposed system is data-race free but the rest of Scala is not.

[6] Rust uses atomic reference counts and reader-writer locks to prevent data races.

[7] Scala has types that are immutable by design, but cannot annotate references to mutable types as immutable.

[8] A version of identity, `none`, appears in [27].

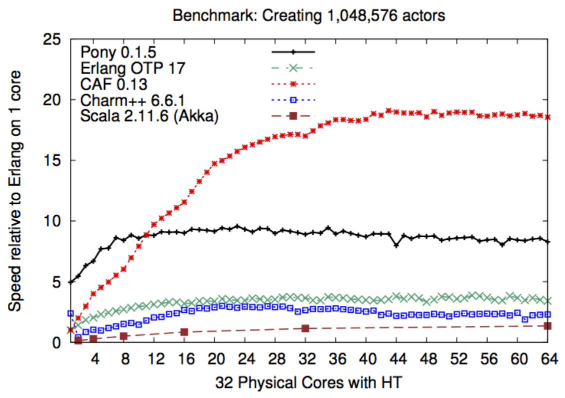

Figure 7. Actor creation

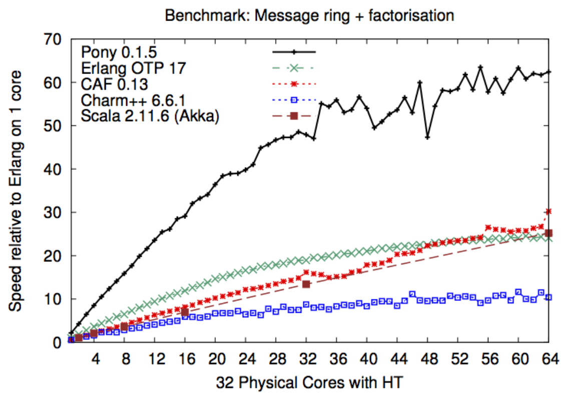

Figure 9. Mixed case performance

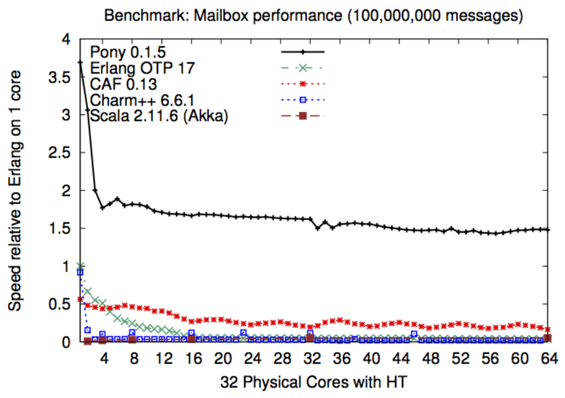

Figure 8. Mailbox performance

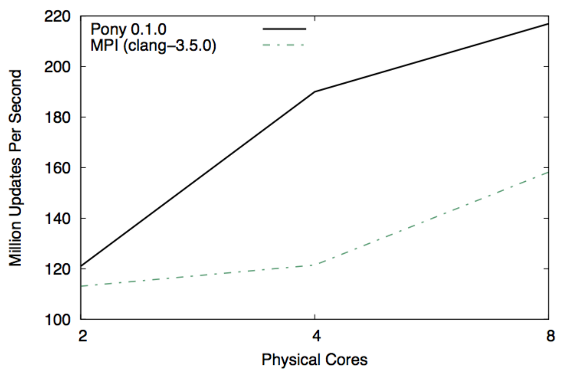

Figure 10. HPCC RandomAccess

ted until the program completes (the worst case for Pony). Here, we are garbage collecting actors themselves [15] as well as objects, but still outperforms existing systems other than CAF, which is neither garbage collected nor data-race free. In fig. 8, we show performance of a highly contended mailbox, where additional cores tend to degrade performance. In fig. 9, we show performance of a mixed case, where a heavy message load is combined with brute force factorisation of large integers.

In fig. 10, we show a benchmark that is not tailored for actors: we take the RandomAccess benchmark from high-performance computing, which tests random access memory subsystem performance, and demonstrate that our implementation is significantly faster than the highly optimised MPI implementation[9].

While all benchmarking is to some degree snake oil, we have chosen these benchmarks because a) they were designed by others, b) they are hopefully representative of some common actor-model programming idioms, and c) they have optimised implementations in existing languages and frameworks provided by programmers expert with those tools.

The full Pony language as implemented in the compiler includes additional features, such as generic types, traits, structural types, type expressions (unions, intersections and tuples), a non-null type system, sound constructors, pattern matching, exceptions, and garbage collection. The Pony runtime will eventually support distributed computation, without a reduction in single-node performance.

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

8. Conclusions and Further Work

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

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

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

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] B. W. Lampson. Protection. *ACM SIGOPS Operating Systems Review*, 8(1):18–24, 1974.

[24] 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.

[25] M. S. Miller and J. S. Shapiro. *Robust composition: towards a unified approach to access control and concurrency control*. PhD thesis, Johns Hopkins University, 2006.

[26] 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.

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

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

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

[30] 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.

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

[32] 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.

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

Figure 11. Auxiliary definitions

Appendix

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

Lookup functions are defined in fig. 12. Function $\mathcal{P}$ returns a type definition for a class identifier `C` or actor identifier `A`. This contains the fields $\overline{\mathtt{F}}$, 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 `S`, and function $\mathcal{F}$ returns the type of field `f` in `S`. Function $\mathcal{M}$ returns method information for some method in `S`. This is overloaded on both the method identifier and the type identifier in order to handle class constructors, actor constructors, synchronous methods (functions) and asynchronous methods (behaviours). The result is a tuple of: the receiver type, the names and types of the parameters, the body of the method, and the return type. The reference capability of the receiver and the return type can vary for synchronous methods, but not for constructors or asynchronous methods. Constructors always operate on a `ref` receiver, since the constructor must write to the new object's fields, and return a `ref`$\circ$ result, since the new object is initially mutable but also unaliased, as the constructor's reference to the receiver (`this`) is discarded when the constructor returns. This allows a constructor that is passed only sendable references as parameters to be embedded in a `recover` expression, which allows constructing an object with any reference capability. Asynchronous methods always operate on a `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.

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

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

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

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

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

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

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

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

Figure 12. Lookup functions

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

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

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

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

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

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

Figure 13. 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 14. Auxiliary well-formedness definitions