

# Reference Capabilities for Trait Based Reuse and Concurrency Control \* †

Elias Castegren<sup>1</sup> and Tobias Wrigstad<sup>1</sup>

<sup>1</sup> Uppsala University, Sweden, `first.last@it.uu.se`

---

## Abstract

The proliferation of shared mutable state in object-oriented programming complicates software development as two seemingly unrelated operations may interact via an alias and produce unexpected results. In concurrent programming this manifests itself as data-races. Concurrent object-oriented programming further suffers from the fact that code that warrants synchronisation cannot easily be distinguished from code that does not. The burden is placed solely on the programmer to reason about alias freedom, sharing across threads and side-effects to deduce where and when to apply concurrency control, without inadvertently blocking parallelism.

This paper presents a reference capability approach to concurrent and parallel object-oriented programming where all uses of aliases are guaranteed to be data-race free. The static type of an alias describes its possible sharing without using explicit ownership or effect annotations. Type information can express non-interfering deterministic parallelism without dynamic concurrency control, thread-locality, lock-based schemes, and guarded-by relations giving multi-object atomicity to nested data structures. Unification of capabilities and traits allows trait-based reuse across multiple concurrency scenarios with minimal code duplication. The resulting system brings together features from a wide range of prior work in a unified way.

## 1 Introduction

Shared mutable state is ubiquitous in object-oriented programming. Sharing can be more efficient than copying, especially when large data structures are involved, but with great power comes great responsibility: unless sharing is carefully maintained, changes through a reference might propagate unexpectedly, objects may be observed in an inconsistent state, and conflicting constraints on shared data may inadvertently invalidate invariants, etc. [29].

Multicore programming stresses proper control of sharing to avoid interference or data-races<sup>1</sup> and to synchronise operations on objects so that their changes appear atomic to the system. Concurrency control is a delicate balance: locking too little opens up for the aforementioned problems. Locking too much loses parallelism and decreases performance.

For example, parallelism often involves using multiple threads to run many tasks simultaneously without any concurrency control. This requires establishing non-interference by considering all the objects accessed by the tasks at any level of indirection.

Mainstream programming languages place the burden of maintaining non-interference, acquiring and releasing locks, reasoning about sharing, etc. completely on the (expert) programmer. This is unreasonable, especially considering the increasing amount of parallelism and concurrency in applications in the age of multicore and manycore machines [6].

---

\* This is an extended version of an article published at ECOOP'16 [16]

† This work was partially funded by the Swedish Research Council project Structured Aliasing, the EU project FP7-612985 Upscale (<http://www.upscale-project.eu>), and the Uppsala Programming Multicore Architectures Research Centre (UPMARC)

<sup>1</sup> Two concurrent operations accessing the same location (read-write or write-write) without any synchronisation is a data-race. Non-interference allows only read-read races and no locks.



In this paper, we explore a reference capability approach to sharing objects across threads. A capability [32, 34] is a token that grants access to a particular resource, in our case objects. Capabilities present an alternative approach to tracking and propagating computational effects to check interference: capabilities assume exclusive access to their governed resources, or only permit reading. Thus, holding a capability implies the ability to use it fully without fear of data-races. This shifts reasoning from use-site of a reference to its creation-site.

We propose a language design that integrates capabilities with traits [40], *i.e.*, reusable units from which classes are constructed. This allows static checking at a higher level of abstraction than *e.g.*, annotations on individual methods. A *mode* annotation on the trait controls how exclusivity is guaranteed, *e.g.*, by completely static means such as controlling how an object may be referenced, or dynamically, by automatically wrapping operations in locks. A trait can be combined with different modes to form different capabilities according to the desired semantics: thread-local objects, immutable objects, unsharable linear objects, sharable objects with built-in concurrency control, or sharable objects for which locks must be acquired explicitly. This extends the reusability of traits across concurrency scenarios.

The sharing or non-sharing of a value is visible statically through its type. Types are formed by composing capabilities. Composition operators control how the capabilities of a type may share data, which ultimately controls whether an object can be aliased in ways that allow manipulation in parallel. Hiding a type’s capabilities allows changing its aliasing restrictions. For example, hiding all mutating capabilities creates a temporarily immutable object which is consequently safe to share across threads (*cf.*, [9]).

Ultimately, with a small set of primitives—differently moded capabilities and composition operators—working in concert, the resulting system brings together many features from prior work: linear types [42, 24] and unique references [28, 35, 8, 18], regions [26], ownership types [17], universe types [23] and (fractional) permissions [9, 43]. As far as the authors are aware, there is no other *single* system that can express all of these concepts in a unified way.

This paper makes several contributions to the area of type-driven concurrency control:

- We present a framework for defining capabilities which work in concert to express a wide variety of concepts from prior work on alias control. The novel integration of capabilities with traits extends trait-based reuse across different concurrency scenarios without code duplication. Traits are guaranteed to be data-race free or free from any interference, which simplifies their implementation and localises reasoning. A single keyword controls this aspect. We support both internal and external locking schemes for data (§3–4).
- We formalise our system in the context of the language  $\mathcal{K}$  (pronounced kappa), state the key invariants of our system (safe aliasing, data-race freedom, strong encapsulation, thread-affinity and partial determinism) and prove them sound (§6–7).

The full proofs, dynamic semantics and a few longer code examples can be found in the appendix.

## 2 Problem Overview

Object-oriented programs construct graphs of objects whose entangled structure can make seemingly simple operations hard to reason about. For example, the behaviour of the following program (adapted from [29]) manipulating two counters `c1` and `c2` depends on whether `c1` and `c2` may alias, which may only be true for some runs of the program.

```
assert c1.value() == 42; c1.inc(); c2.inc(); assert c1.value() == 43;
```

If `c1` and `c2` *always alias*, we may reason about the sequential case, but if `c2.inc()` is performed

by *another thread*, the behaviour is affected by the scheduling of `c2.inc()`, and whether `inc()` itself is thread-safe. While aliasing is possible without sharing across threads, sharing across threads is not possible without aliasing. With this in mind, we move on to three case studies to discuss some of the challenges facing concurrent object-oriented programming.

## 2.1 Case Study: Simple Counters

To achieve thread-safety for a counter implemented in Java we can make the `inc()` method synchronised to ensure only one thread at a time can execute it. While this might seem straightforward, there are at least three problems with this approach:

1. Additional lock and unlock instructions for each increment will be inserted regardless of whether they are necessary or not – synchronising an unaliased object is a waste.
2. Making the object thread-safe does not help protect an instance from being shared, which might have correctness implications (*e.g.*, non-determinism due to concurrent accesses).
3. Unless the `value()` method is also synchronised, concurrent calls to `inc()` and `value()` may lead to a data-race, which might lead to a perception of lost increments.

In 1. and 2., the underlying problem is distinguishing objects shared across threads from thread-local objects as only the former needs synchronisation. Using two different classes for shared and unshared counters are possible, but leads to code duplication. Furthermore, if a counter is shared indirectly, *i.e.*, there is only one counter but its containing object is shared, the necessary concurrency control might be in the container. Establishing and maintaining such a “guarded-by property” warrants tool support.

In 3., the underlying problem is the absence of machinery for statically checking that all accesses to data are sufficiently protected. This might not be easy, for example, excluding data-races in methods inherited from a super class that encapsulates its locking behaviour.

## 2.2 Case Study: Data Parallelism and Task Parallelism

The counter exemplifies concurrent programming which deals with asynchronous behaviour and orchestration of operations on shared objects. In contrast, parallelism is about optimisation with the goal of improving some aspect of performance.

Consider performing the operations  $f_1$  and  $f_2$  on all elements in a collection  $E$ . A data parallel approach might apply  $f_1(f_2(e))$  in parallel to all  $e \in E$ . In contrast, a task parallel approach might execute  $f_1(e_1); \dots; f_1(e_n)$  and  $f_2(e_1); \dots; f_2(e_n)$  as two parallel tasks.

Both forms of parallelism requires proper alias management to determine whether  $f_1(e_i)$  and  $f_2(e_j)$  may safely execute in parallel. When  $i = j$ , we must determine what parts of an object’s interface might be used concurrently. When  $i \neq j$ , we must reason about the possible overlapping states of (the different) elements  $e_i$  and  $e_j$ . Furthermore, unless  $f_1(e)$  (or  $f_1(f_2(e))$ ) is safe to execute in parallel on the same object, we must exclude the possibility that  $E$  contains duplicate references to the same object.

If  $f_1$  and  $f_2$  only perform reads, any combination is trivially safe. However, correctly categorising methods as accessors or mutators manually can be tricky, especially if mutation happens deep down inside a nested object structure, and a method which may logically only read might perform mutating operations under the hood for optimisation, telemetry, etc. Extending the categorisation of methods to include mutation of disjoint parts further complicates this task. Further, as software evolves, a method’s categorisation might need to be revisited, even as a result of a non-local change (*e.g.*, in a superclass).

### 2.3 Case Study: Vector vs. ArrayList in Java

As a final case study, consider the `ArrayList` and `Vector` classes from the Java API. While both implement a list with comparable interfaces, vectors are thread-safe whereas array lists are not. There are several consequences of this design:

1. Vector objects lock individual operations. This requires multiple acquires and releases for compound operations (*e.g.*, when using an external iterator to access multiple elements).
2. The reliance on Java objects' built-in synchronisation excludes concurrent reads.
3. Just like the counter above, even thread-local vectors pay the price of synchronisation.

As a result, `ArrayList` is commonly favoured over `Vector` despite the fact that this requires locks to be acquired correctly for each use, rather than once if built into the data structure.

A lock that allows multiple concurrent reads (a readers–writer lock) would allow both vectors and array lists to be used efficiently and safely in parallel. This distinction adds an extra dimension of locking and requires categorising methods as accessors/mutators.

**Summary** The examples above illustrate a number of challenges facing programmers doing concurrent and parallel programming in object-oriented languages. In summary:

- Code that needs synchronisation for data-race freedom is indistinguishable from code that does not. The same holds for code correctly achieving non-interference.
- Conservatively adding locks to all data structure definitions or all uses of a data structure hurts performance.
- Using locks to exclude conflicting concurrent accesses is non-trivial and requires reasoning about aliasing and program-wide sharing of data structures. The same reasoning is required for partitioning a data structure across multiple threads for parallel operations on disjoint parts, or specifying read-only operations.
- The need for concurrency control varies across different usage scenarios. Building concurrency control into data structures generates overhead or leads to code duplication (one thread-safe version and one which is not). Leaving concurrency control in the hands of clients instead opens up for under-synchronisation and concurrency bugs.
- The need for alias control varies across different usage scenarios. At times, thread-locality or even stronger aliasing restrictions are desirable, for example to avoid locks or non-determinism, or to unlock compiler optimisations or simplify verification. At other times, sharing is required. The sharing requirements of a single object could even vary over time.

We now describe our reference capability system which addresses all of these problems.

## 3 Capabilities for Concurrency Control

Our starting point for this work is to unify references and capabilities. A capability is a handle to a resource—a part of or an entire object or aggregate (an object containing other objects). A capability exposes a set of operations, which can be used to access its resource without possibility of data-races. Granting and revoking capabilities corresponds to creating and destroying aliases. Capabilities' *modes* controls how they may be shared across threads:

**Exclusive** capabilities denote resources that are exclusive to one thread so that accesses are trivially free from any interference from other threads. There are two exclusive modes: **linear**, used for resources to which there is only a single handle in the program, and **thread**, which allows sharing, but only within one single thread. **linear** capabilities must be fully transferred from one thread in order to be used by another thread.

**Safe** capabilities denote resources that can be arbitrarily shared (*e.g.*, across multiple threads). There are two safe modes: **locked**, causing operations to be implicitly guarded by locks, and **read** which do not allow causing or directly observing mutation. Safe capabilities guarantee data-race freedom.

**Subordinate** capabilities (the mode **subordinate**) denote resources that are encapsulated inside some object and therefore inherit its protection against data-races or interference. Subordinate capabilities are similar to `rep` or `owner` in ownership types [17].

**Unsafe** capabilities (the mode **unsafe**) denote arbitrarily shared resources which are unsafe to use concurrently without some means of concurrency control. Accesses to unsafe capabilities must be wrapped in explicit locking instructions.

Linear capabilities impose transfer semantics on assignment. We adopt destructive reads [28] here for simplicity. This means that reading a variable holding a linear capability has the side-effect of updating it with **null**. Methods in **locked** capabilities automatically get acquire and release instructions, providing *per-method* atomicity. For **unsafe** capabilities locking must be done manually, providing *scoped* atomicity (the duration of the lock). Although straightforward, for simplicity we do not allow manual locking of **locked** in this presentation.

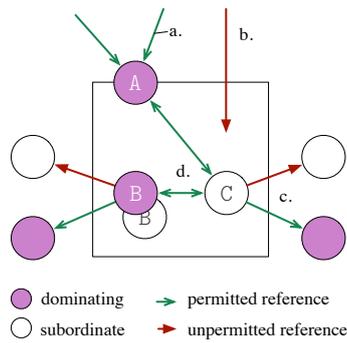
Types are compositions of one or more capabilities (*cf.*, §3.3) and expose the union of their operations. The modes of the capabilities in a type control how resources of that type can be aliased. The compositional aspect of our capabilities is an important difference from normal type qualifiers (*cf.*, *e.g.*, [25]), as accessing different parts of an object through different capabilities in the same type gives different properties.

Exclusive and read capabilities guarantee *non-interference* and enable deterministic parallelism. Safe capabilities guarantee the absence of *data-races*, *i.e.*, concurrent write–write or read–write operations to the same memory location, but do not exclude *race-conditions*, *e.g.*, two threads competing for the same lock. This means that programs will be thread-safe, only one thread can hold the lock, but not necessarily deterministic—the order in which competing threads acquire a lock is controlled by factors external to the program. This also means that capabilities using locks do not exclude the possibility of deadlocks.

### 3.1 Capability = Trait + Mode

We present our capabilities system through  $\mathcal{K}$ , a Java-like language that uses traits [40] in place of inheritance for object-oriented reuse. A  $\mathcal{K}$  capability corresponds to a trait with some required fields, provided methods, and a mode. For the reader not familiar with traits, a trait can be thought of as an abstract class whose *fields* are abstract and must be provided by a concrete subclass—see Figure 4 for a code example of traits and classes. An important property of  $\mathcal{K}$  is that an *implementer of a trait can assume freedom from data-races or interference*, which enables sequential reasoning for all data that the trait *owns*, (its subordinate capabilities), plus reachable exclusive capabilities. A trait’s mode controls how data-race freedom or non-interference is guaranteed. For example, prohibiting aliases to cross thread boundaries or inserting locks at compile-time in its methods.

The mode of a trait is either *manifest* or must be given wherever the trait is included by a class. A manifest mode is part of the declaration of the trait, meaning the trait defines a single capability. As an example of this, consider the capability `read Comparable` which provides `compare` methods to a class which do not mutate the underlying object. Traits without manifest modes can be used to construct different capabilities, *e.g.*, a trait `Cell` might be used to form both a **locked** `Cell` and a **linear** `Cell` when included in different classes, with different constraints on aliasing of its instances.



- Disallowed if A is **linear**. If A is **thread**, aliases must come from same thread.
- References from outside an aggregate to its inside are not permitted.
- References from inside an aggregate to its outside are permitted if the target is a dominating capability.
- References inside an aggregate are allowed.

The box encloses the subordinate capabilities of A. Note that B is a composition of a subordinate *and* a dominating capability (cf. §3.3), denoted by the two circles. All dominating capabilities have their own boxes (as shown for A), e.g., B has a box nested inside of A's box, inaccessible to A (cf., b.).

■ **Figure 1** Encapsulation: dominating and subordinate capabilities.

As a consequence of this design  $\mathcal{K}$  allows the same set of traits to be used to construct classes tailored to different concurrency scenarios, thus contributing to trait-based reuse.

### 3.2 Dominating and Subordinate Capabilities

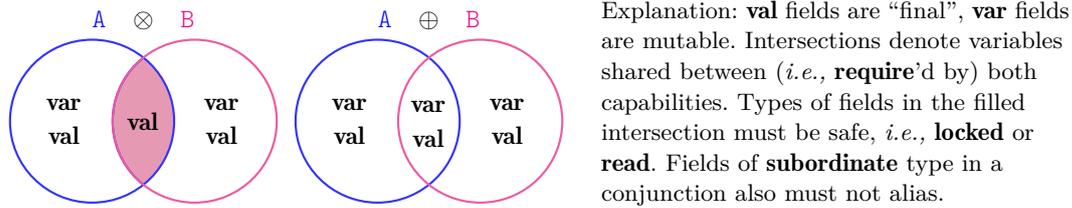
Building a data structure from linear capabilities gives *strong encapsulation*: subobjects of the data structure are not aliased from outside. However, linearity imposes a tree-shaped structure on data. Subordinate capabilities instead provide strong encapsulation by forbidding aliases from outside an aggregate to objects within the aggregate. Inside an aggregate, subordinate capabilities may be aliased freely, enabling any graph structure to be expressed.

The capabilities **linear**, **thread**, **locked** and **unsafe** are *dominating capabilities* that enclose subordinate capabilities in a statically enforced way. Domination means that all *direct* accesses to objects inside an aggregate from outside are disallowed, making the dominator a single point of entry into an aggregate. As a consequence, any operation on an object inside an aggregate must be triggered by a method call on its dominating capability (directly or indirectly). This means that subordinate objects inherit the concurrency control of their dominator. Subordinate capabilities dominated by a **thread** capability inherit its thread-locality; subordinate capabilities dominated by a **locked** capability enjoys protection of its lock, etc. An implementation of a linked list with **subordinate** links inside a dominating list head guarantees that only a single thread at a time can mutate the links, while still allowing arbitrary internal aliasing inside of the data structure (e.g., doubly-linked, circular).

Figure 1 shows encapsulation in  $\mathcal{K}$  from dominating and subordinate capabilities. To enforce the encapsulation of subordinate objects, a subordinate capability (B and C) may not be returned from or passed outside of its dominating capability (A). There is no hierarchical decomposition of the heap (cf., [17]) and no notion of transitive ownership. However, compositions (cf. §3.3) of dominating and subordinate capabilities (B) create nested aggregates, *i.e.*, entire aggregates strongly encapsulated inside another. Pointers to external capabilities must all be to dominating capabilities. Thus, objects inside B can refer to A, but not to C.

### 3.3 Flat and Nested Composition

As usual in a trait-based system,  $\mathcal{K}$  constructs classes by composing traits, or rather capabilities. There are two forms of composition: disjunction ( $\oplus$ ) and conjunction ( $\otimes$ ). If A and B are capabilities, their disjunction  $A \oplus B$  provides the union of the methods of A and B and requires the union of their field requirements. Their conjunction  $A \otimes B$  does the same, but is only well-formed if A and B do not share mutable state which is not protected by concurrency



■ **Figure 2** Permitted sharing of fields and state across two capabilities A and B in a composite.

control. This means that  $A \otimes B$  allows A and B to be used in parallel. Figure 2 shows the composition constraints of disjunction and conjunction pictorially.

We use the term *flat composition* to mean disjunction or conjunction. When employing parametric polymorphism a form of *nested composition* appears. The nested capability  $A \langle B \rangle$  exposes that A contains zero or more B’s at the type level, allowing type-level operations on the composite capability. (This presentation uses a “dumbed down” version of parametric polymorphism using concrete types in place of polymorphic parameters for simplicity.)

A composite capability inherits all properties and constraints of its sub-capabilities. Linear capabilities must not be aliased at all. Subordinate capabilities must not leak outside their dominator. Consequently, a type which is both **subordinate** and **linear** is both a dominator (may encapsulate state) and a subordinate (is encapsulated), may not escape its enclosing aggregate and has transfer semantics when assigned (*cf.*, B in Figure 1).

Composition affects locking. A disjunction of two locked capabilities  $A \oplus B$  will be protected by a single lock. A conjunction  $A \otimes B$  of locked capabilities can use different locks for A and B, allowing each disjoint part to be locked separately. Furthermore, compositions of **read** and **locked** capabilities can be mapped to readers–writer locks.

An important invariant in  $\mathcal{K}$  is that all aliases are safe with respect to data-races or interference and can be used to the full extent of their types. If an alias can be created, any use of it will not lead to a bad race, either because it employs some kind of locking, because all aliases are read-only, or because the referenced object is exclusive to a particular thread.

## 4 Creating and Destroying Aliases = Concurrency Control

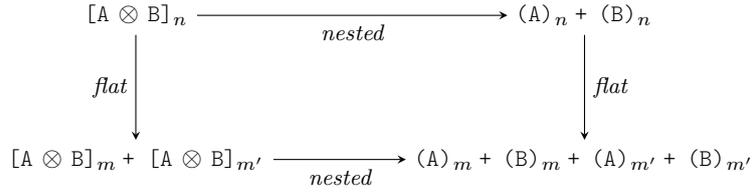
As aliasing is a prerequisite to sharing objects across possibly parallel computations, creating and destroying aliases is key to enabling parallelism while still guaranteeing race freedom in  $\mathcal{K}$ . Alias restrictions allows statically checkable non-interference, *i.e.*, without dynamic concurrency control (*e.g.*, locking). Programs that require objects that are aliased across threads must employ locks or avoid mutation.

Subordinate and thread-local capabilities may only be aliased from within certain contexts. Read, locked and unsafe capabilities have no alias restrictions. Finally, linear capabilities are alias-free. The following sections explore how linear types can be manipulated to create and destroy aliases (granting and revoking capabilities) while enjoying non-interference.

### 4.1 Packing and Unpacking

Conjunctions describe objects constructed from parts that can be manipulated in parallel without internal races. Unpacking breaks an object up into its sub-parts. A variable  $c$  with a handle to an instance of a class C, where  $\mathbf{class} \ C = A \otimes B$ , can be unpacked into two handles with types A and B using the + operator:  $\mathbf{var} \ a:A + b:B = c$ , nullifying  $c$  in the process.

Unpacking a disjunction is unsafe (and therefore disallowed) since its building blocks can



■ **Figure 3** Flat and nested unpacking, using arrays as an analogy.  $[A \otimes B]_n$ , an array of length  $n$  containing composite capabilities  $A \otimes B$  can be thought of as a matrix with rows as elements and whose columns are the elements' subparts,  $A$  and  $B$ . The matrix can be unpacked by rows (flat) or by columns (nested). Flat unpacking splits the array into subarrays of length  $m$  and  $m'$  such that  $n = m + m'$ . Nested unpacking requires that the containing object is not mutable, denoted by turning arrays into tuples,  $(A)_m$ . These compose in any order producing the same result.

share mutable state not mediated by concurrency control. The dual of unpacking is packing, which re-assembles an object by revoking (nullifying) its sub-capabilities: `var c:C = a + b`.

The packing and unpacking above is *flat*. Using an array as analogy, flat unpacking takes an array  $[A]_n$  with indexes  $[0, n)$  and turns it into two disjoint equi-typed sub-arrays with indexes  $[0, m)$  and  $[m, n)$  where  $m \leq n$ .  $\mathcal{K}$  also allows *nested unpacking*, which in the array analogy means that  $[A \otimes B]_n$  can be unpacked into two tuples  $(A)_n$  and  $(B)_n$  with the same length and indexes. Turning the array into tuples, *i.e.*, immutable arrays of mutable values, is necessary as the aliases could otherwise be used to perform conflicting operations, *e.g.*, updating the B-part of element  $i$  in one thread and nullifying element  $i$  in another thread.

While safe capabilities can always be shared, unpacking allows a linear capability to be split into several aliases that can safely be used concurrently. When restoring the original capability through packing, there may be no residual aliases. We implement this here by preserving linearity in the unpacked capabilities. Figure 3 shows flat and nested unpacking and how they combine and commute. §5.2 shows how unpacking can be used to implement both data parallelism and task parallelism.

In this paper, we only consider packing and unpacking as operations at the level of types: their purpose is to statically guarantee non-interference, not construct new objects from other parts. Thus, packing can be efficiently compiled into an identity check or removed by a compiler provided that handles do not escape the scope in which they were unpacked.

## 4.2 Bounding Capabilities to the Stack

Linearity is often overly restrictive since it prevents even short-lived aliases that do not break any invariants. To remedy this,  $\mathcal{K}$  employs *borrowing* [8]: temporarily relaxing linearity as long as the original capability is not accessible in the same scope, and all aliases are destroyed at the end of the scope. Borrowed capabilities in  $\mathcal{K}$  are *stack-bound*, denoted by a type wrapper  $\mathbf{S}()$ . For example,  $\mathbf{S}(\mathbf{linear} \text{ Cell})$  denotes a capability which is identical to the  $\mathbf{linear} \text{ Cell}$  capability except that it may not be stored in a field, and thus is revoked once the scope exits.  $\mathcal{K}$  supports two forms of borrowing:

**Forward Borrowing** A  $\mathbf{linear}$  capability in a stack variable can be converted into a stack-bound capability for a certain scope, destructively read and then safely reinstated at the end of the scope. This allows *e.g.*, passing a linear capability as an argument to a method, reinstating it on return. In conjunction with the borrowing it may optionally be converted to a **thread**, allowing it to be freely aliased until reinstated.

```

class Pair = (linear Fst  $\otimes$  linear Snd)  $\oplus$  linear Swap { var fst:int; var snd:int; }
trait Fst {
  require var fst:int;

  def setFst(i:int) : void {
    this.fst = i;
  }
  def getFst() : int {
    this.fst;
  } }
trait Snd {
  require var snd:int;

  def setSnd(i:int) : void {
    this.snd = i;
  }
  def getSnd() : int {
    this.snd;
  } }
trait Swap {
  require var fst:int;
  require var snd:int;

  def swap() : void {
    var tmp:int = this.fst;
    this.fst = this.snd;
    this.snd = tmp;
  } }

```

■ **Figure 4** A pair class constructed from capabilities, Fst, Snd and Swap.

**Reverse Borrowing** A method of a **linear** capability may non-destructively read and return a stack-bound alias of a field of **linear** type. This allows linear elements of a data structure to be accessed without removing them, which is safe as long as the capability holding the field is not accessed during borrowing. To prevent multiple reverse borrowings of the same value (which would break linearity), the returned value may not be stored in fields or local variables but must be used immediately, *e.g.*, as an argument to a method call.

Borrowing simplifies programming with linear capabilities as it removes the need to explicitly consume and reinstate values when aliasing is benign, avoiding unnecessary memory writes. See §5.2 for an example of both forward and reverse borrowing in action.

### 4.3 Forgetting and Recovering Sub-Capabilities

Unpacking a disjunction is unsafe as its building blocks may have direct access to the same state without any concurrency control. As an example, consider the simple `Pair` class created from the capabilities `Fst`, `Snd` and `Swap` shown in Figure 4.

If we could unpack the pair, it would allow `fst` and `snd` to be updated independently. However, this is unsafe in the presence of the `Swap` capability, which accesses both fields. For example, the result of calling `swap()` concurrently with `setFst()` depends on the timing of the threads. A crude solution is simply upcasting `Pair` to **linear** `Fst`  $\otimes$  **linear** `Snd`. This *forgets* the `Swap` capability and enables unpacking—but as a consequence `Swap` is lost forever.

To facilitate recovering a more specific type,  $\mathcal{K}$  provides a means to temporarily stash capabilities inside a *jail* which precludes their use except for recovering a composite type:

```

var p:Pair = ...;
var j:J(Pair|Fst  $\otimes$  Snd) + k:(Fst  $\otimes$  Snd) = p; // (1)
var f:Fst + s:Snd = k; // flat unpacking
... // use f and s freely
p = j + (f + s); // flat packing, twice, and getting out of jail (2)

```

At (1), the type of `j`,  $\mathbf{J}(\text{Pair}|\text{Fst} \otimes \text{Snd})$ , denotes a jail storing a `Pair` which is unusable (the interface of a jailed capability is empty) until it is unlocked by providing the `Fst`  $\otimes$  `Snd` capability of the corresponding resource as key. Thus `j` serves as a witness to the existence of the full `Pair` capability, including `Swap`. At (2), we recover `k` from `f` and `s`, nullifying both variables. We use the resulting value to open the jail `j` and store the result in `p`. As for packing, checking whether a key “fits” at run-time (*i.e.*, if `f` and `s` are aliases of the jail) is a simple pointer identity check, which could often be optimised away using escape analysis.

## 5 Applying Capabilities to the Case Studies in § 2.1–2.3

### 5.1 Simple Counters

This example demonstrated the problem of distinguishing objects shared across threads from thread-local or unaliased objects, and pointed at the trickiness of locking correctly. In  $\mathcal{K}$ , a counter might be described as a simple trait `Counter`:

```
trait Counter {
  require var cnt : int;
  def inc() : void { this.cnt = this.cnt + 1; }
  def value() : int { return this.cnt; } }
```

To get a capability from the trait, what is missing is to add the mode declaration, which controls aliasing and sharing across threads. Out of the six possible mode annotations, five are allowed for the `Counter` trait:

**linear** A globally unaliased counter.

**thread** A thread-local counter. It can be aliased, but aliases cannot cross into other threads.

**locked** A counter protected by a lock, sharable across threads.

**subordinate** This type denotes a counter nested inside another object from which it cannot escape. It thus inherits data-race freedom or non-interference of the enclosing object.

**unsafe** A sharable, unprotected counter that requires the client to perform synchronisation at use-site: `c.inc()` will not compile unless wrapped inside a synchronisation block, which changes the type of `c` from **unsafe** to **locked**.

Using the mode **read** would denote a read-only counter, sharable across threads. Assigning this mode to the trait is rejected by the compiler because of the mutable `cnt` field.

Modes communicate how counters may be aliased: not at all, by a single thread, or across threads. In the latter case modes also communicate how concurrent accesses are made safe: by locks, by only allowing reads (not applicable here), by relying on some containing object or by delegating responsibility to the client.

Differently synchronised counters can be defined almost without code duplication, *e.g.*:

```
class LocalCounter = thread Counter { var cnt:int; }
class SharedCounter = locked Counter { var cnt:int; }
```

### 5.2 Data/Task Parallelism

This example demonstrated the need for reasoning about aliasing in order to determine what parts of an interface can be safely accessed concurrently.

A binary tree can be constructed as the conjunction of capabilities giving access to the left and right subtrees and the current element (full code in the appendix).

```
class Tree<T> = linear Left<T> ⊗ linear Right<T> ⊗ linear Element<T>
```

We employ nesting to show that the tree contains capabilities of type `T`, the type of the element value held by the `Element` capability. The conjunction allows parallel operations on subparts of a tree and requires that parts do not overlap, modulo safe capabilities. Since the tree type must be treated linearly, the fact that the `Left` and `Right` subtrees do not overlap follows from the requirement that `Left` and `Right` manipulate fields of different names.

To perform data-parallel operations on a tree, we can construct a recursive procedure that takes a tree, splits it into its separate components and operates on them in parallel.

```
def foreach(t:S(Tree<T>), f:T → T) : void {
  var l:S(linear Left<T>) + r:S(linear Right<T>) + e:S(linear Element<T>) = t; // 0
```

```

finish {
  async { foreach(l.getLeft(), f); } // 1
  async { foreach(r.getRight(), f); } // 1
  e.apply(f); } } // 2

```

At (0) the splitting implicitly consumes the original tree capability. At (1) we recurse on the left and right subtrees. At (2) we pass the function argument  $f$  to the element capability to be performed on its  $T$ -typed value. For simplicity, we omit the check for whether  $l$  or  $r$  is **null**. The implementation requires a tree to be constructed from linear building blocks to guarantee that no parts of the tree are ever shared across multiple threads.  $T$  does not need to be linear.

This code illustrates both forward and reverse borrowing. The tree argument to `foreach()` is forward borrowed and stack-bound, which is why there is no need to pack  $l$ ,  $r$  and  $e$  to recover  $t$ — $t$  is still accessible at the call-site, where it was buried [8] during the call.

Calls to `getLeft()` and `getRight()` return two reversely borrowed linear values (of type  $S(\text{Tree}\langle T \rangle)$ ) which we can pass as arguments to the recursive calls. Hence, all trees manipulated by this code will be stack-bound. If we remove the stack-boundedness, `foreach()` may not update the subtrees in-place, and must recover and return  $t$  at the end, reminiscent of functional programming. This would cause lines marked (1) to change thus:

```

async { l.setLeft( foreach( l.getLeft(), f ) ); }
async { l.setRight( foreach( l.getRight(), f ) ); }

```

which allows *replacing* the tree as opposed to updating it, plus a return: **return**  $l + r + e$ .

We may extend the `Tree` type with a disjunction on a capability `Visit` which provides a read-only view of the entire tree. Elements may not be swapped for other elements, but modified if  $T$  allows it. This allows multiple threads to access the same tree in parallel provided that `Left`, `Right` and `Element` are temporarily forgotten.

```

class Tree<T> = read Visit<T>  $\oplus$  (linear Left<T>  $\otimes$  linear Right<T>  $\otimes$  linear Element<T>)

```

Let the type of our tree be  $\text{Tree}\langle A \otimes B \rangle$  for linear capabilities  $A$  and  $B$ . Turning this capability into  $\text{Visit}\langle A \otimes B \rangle$  is possible by forgetting every other capability in the tree type. While read-only capabilities can be aliased freely, creating multiple aliases typed  $\text{Visit}\langle A \otimes B \rangle$  would provide multiple paths to supposedly linear  $A \otimes B$  capabilities. Composition must thus adhere to all alias restrictions in the composite capability, just like flat composition. Therefore,  $\text{Visit}\langle A \otimes B \rangle$  is a linear capability. Unpacking however allows us to turn  $\text{Visit}\langle A \otimes B \rangle$  into two handles typed  $\text{Visit}\langle A \rangle$  and  $\text{Visit}\langle B \rangle$ , which preserves linearity across all paths. This allows us to specify a task-parallel operation which implements column-based access:

```

def map(t:S(Tree<A  $\otimes$  B>), f:S(A)  $\rightarrow$  void, g:S(B)  $\rightarrow$  void) : void {
  var ta:S(read Visit<A>) + tb:S(read Visit<B>) = t; // 3
  finish {
    async { ta.preorder(f); } // 4
    async { tb.preorder(g); } } // 4

```

In this code we create two immutable views of the spine of the tree using `Visit` and then proceed to apply  $f$  and  $g$  to all elements of the tree in parallel. At (3) the rest of the capabilities of `Tree` are forgotten. If we wanted to restore them after the parallel operations we would jail them at (3) and restore them after (4).

While the data-parallel version is more scalable than the task-parallel version, there may be cases when the latter is preferred. Further, their combination is possible in either order—apply  $f$  and  $g$  in parallel to each element at (2) above, or start by unpacking the tree into multiple immutable trees and then process the sub-elements in parallel in each tree, equivalent to calling a version of `foreach` instead of `preorder` at (4) (*cf.*, Figure 3).

### 5.3 Vector vs. ArrayList in Java

This example demonstrated that building synchronisation into a data structure can cause too much overhead and destroy parallelism. In  $\mathcal{K}$ , a list might be described using capabilities (full code in the appendix):

- Add\_Del for adding and removing elements
- Get for looking up elements

Add\_Del might be split into two capabilities allowing for more flexibility, for example granting a client only the ability to add elements but not delete them. As the two capabilities operate on some shared state (the links), their combination must be a disjunction:  $\text{Add\_Del} \oplus \text{Get}$ .

To express the difference between the array list and vector, we would write

```
class ArrayList = unsafe Add_Del  $\oplus$  unsafe Get // Needs external synchronisation
class Vector = locked Add_Del  $\oplus$  locked Get // Has synchronisation built in
```

Specifying use of readers–writer locks to access an object is straightforward and allows sharing a list across threads for reading, causing concurrent write operations to block:

```
class ArrayList = unsafe Add_Del  $\oplus$  read Get
class Vector = locked Add_Del  $\oplus$  read Get
```

The use of **unsafe** in the definition of the array list class pushes the synchronisation from within the called methods to the outside, *e.g.*, calling `list.add(element)` we must first take a (write-)lock on `list`. Requiring external synchronisation also allows acquiring, holding and releasing a lock once to perform several operations, like an iteration, without fear of interleaving accesses from elsewhere.

The type **thread**  $\text{Add\_Del} \oplus \text{read Get}$  denotes a list confined to its creating thread. The type **linear**  $\text{Add\_Del} \oplus \text{read Get}$  denotes a list that can mediate between being mutated from one alias or read-only from several aliases. This type is similar to a readers–writer lock, except relying on alias restrictions instead of locks (*cf.*, [9]), removing locking overhead. The ability to reuse traits for different concurrency scenarios is an important contribution of  $\mathcal{K}$ .

### Concluding Remarks for §3–5

Linear and thread-local capabilities give *non-interference* by restricting aliases to a single thread. Locked and unsafe capabilities can be shared across threads and employ locks at declaration-site or at use-site to *avoid data-races*. Read capabilities can be shared across threads and do not allow causing or directly witnessing mutation. When a read capability is extracted from a linear composite, no mutating aliases exist, guaranteeing *non-interference*. When extracted from a locked composite, locks are used to guarantee *data-race freedom*.

The assignment of modes to traits at inclusion site allow a single definition to be reused across multiple concurrency scenarios. Composition captures how different parts of an object’s interface interact and defines the safe aliasing of an object.

Subordinate capabilities inherit the protection of their enclosing dominating capabilities. Thus, operations on encapsulated objects are atomic in  $\mathcal{K}$ , in the sense that all side-effects of a method call on an aggregate are made visible to other threads atomically. Operating atomically on several objects which are not encapsulated in the same aggregate is possible by locking them together using nested synchronisation (for **unsafe** capabilities) or by structuring a call-chain on **locked** capabilities.

Invariantly, all well-typed aliases can coexist without risking data-races. The type system guarantees that all accesses to an object will either be exclusive or only perform operations that cannot clash with any other possible concurrent operations to the same object.

$P ::= Cds Tds e$	<i>(Program)</i>
$Cd ::= \mathbf{class} C = K \{ Fds \}$	<i>(Class definition)</i>
$Fd ::= \mathit{mod} f : \tau$	<i>(Field definition)</i>
$\mathit{mod} ::= \mathbf{var} \mid \mathbf{val}$	<i>(Mutable and immutable fields)</i>
$K ::= k T \mid k \mathbf{I} \mid K(K) \mid (K \odot K)$	<i>(Capabilities and composition)</i>
$\odot ::= \otimes \mid \oplus$	<i>(Conjunction and disjunction)</i>
$Td ::= k \mathbf{trait} T(t) \{Rs Mds\} \mid \mathbf{trait} T(t) \{Rs Mds\}$	<i>(Trait definition)</i>
$R ::= \mathbf{require} Fd$	<i>(Field requirement)</i>
$Md ::= \mathbf{def} m(x : t) : t \{e\}$	<i>(Method definition)</i>
$e ::= v \mid \mathbf{let} x = e \mathbf{in} e \mid \mathbf{pack} x = y + z \mathbf{in} e \mid \mathbf{unpack} x + y = z \mathbf{in} e \mid x.m(e) \mid x \mid x.f$ $\mid x.f = e \mid \mathbf{new} C \mid \mathbf{consume} x \mid \mathbf{consume} x.f \mid (t) e \mid \mathbf{sync} x \mathbf{as} y \{e\}; e$ $\mid \mathbf{bound} x \{e\}; e \mid \mathbf{finish} \{ \mathbf{async} \{e\} \mathbf{async} \{e\} \}; e$	<i>(Expression)</i>
$v ::= \mathbf{null}$	<i>(Literal)</i>
$t ::= K \mid C \mid B(K)$	<i>(Type)</i>
$B ::= \mathbf{J}_K \mid \mathbf{S}$	<i>(“Boxed” types, i.e., jailed or stack-bound)</i>
$k ::= \mathbf{linear} \mid \mathbf{locked} \mid \mathbf{read} \mid \mathbf{safe} \mid \mathbf{subordinate} \mid \mathbf{thread} \mid \mathbf{unsafe}$	<i>(Modes)</i>

■ **Figure 5** Syntax of  $\mathcal{K}$ .  $T$  is a trait name;  $\mathbf{I}$  is the incapability;  $C$  is a class name;  $m$  is a method name;  $f$  is a field name;  $x, y$  are variable names, including **this**.  $Ds ::= D_1, \dots, D_n$  for  $D \in \{Cd, Td, Fd, R, Md\}$ .

## 6 Formalising $\mathcal{K}$

We formalise the static semantics of  $\mathcal{K}$ . We define a flattening translation into a language without traits,  $\mathcal{K}_F$ , whose static and dynamic semantics is found in the appendix.  $\mathcal{K}_F$  is a simple object-oriented language with structured parallelism and locking, that uses classes and interfaces which are oblivious to the existence of  $\mathcal{K}$  capabilities. The translation from  $\mathcal{K}$  to  $\mathcal{K}_F$  inserts locking and unlocking operations when translating **locked** capabilities and conjunctions of **locked** and **read** capabilities. The locks are reentrant readers-writer locks controlling access to parts of objects. Other locking schemes are possible.

The syntax of  $\mathcal{K}$  is shown in Figure 5. We make a few simplifications, none of which are critical for the soundness of the approach:

1. We use let-bindings and explicit pack/unpack constructs. Targets of method calls must be stack variables. Aliasing stack-bounds requires a method-call indirection.
2. We consider finish/async parallelism rather than unstructured creation of threads.
3. Classes only contain fields and no methods.
4. We omit the treatment of constructors. Fields are initialised with **null** on instantiation.
5. We use objects to model higher-order functions and omit these from the formalism.
6. Only a single method parameter and a single nested type are supported.

We introduce a **safe** capability, which abstracts **read** and **locked** to allow mode subtyping. The **safe** mode is only allowed in types, not in declarations. The *incapability* type  $\mathbf{I}$  does not contain any fields or methods and simply allows holding a reference to an object.

Our main technical result is the proof that a  $\mathcal{K}_F$  program translated from a well-typed  $\mathcal{K}$  program enjoys safe aliasing and strong encapsulation (*cf.* § 7.2) in a way that implies thread-safety (*cf.* § 7.3). We verify our definition of thread-safety by proving that it implies data-race freedom and, when certain capabilities are excluded, also non-interference (*cf.* § 7.3).

### 6.1 Helper Predicates and Functions

The functions **fields**, **vals**, **vars** and **msigs** return a map from names to types or method signatures. We use helper predicates of the form  $k(K)$  to assess whether a capability  $K$  has

mode  $k$ . The predicates **linear**, **subord**( $\text{inate}$ ) and **unsafe** hold if there exists *at least one* sub-capability in  $K$  of that mode. The predicates **read**( $K$ ) and **encaps**( $K$ ) hold if *all* sub-capabilities in  $K$  are **read** or **subordinate**, respectively. **locked**( $K$ ) holds if one or more sub-capabilities are locked, and the remainder **safe**.

## 6.2 Well-Formed $\kappa$ Programs (Figure 6)

A well-formed program consists of classes, traits, and an initial expression (WF-PROGRAM). Traits without manifest mode are type-checked as if they were subordinate (WF-T-TRAIT). To reduce the number of rules, we require all traits to have exactly one nested capability (a concrete type “parameter”), and use  $T$  as shorthand for  $T\langle I \rangle$ , where  $I$  is the empty capability. A trait is well-formed if its field requirements and methods are well-typed given the self-type of the current trait and the nested type. The latter is tracked by the special variable  $\rho$  which may not appear anywhere in the program source (WF-T-TRAIT-MFST). Fields are either mutable (**var**) or stable (**val**). We assume that names of classes and traits are unique in a program and the names of fields and methods are unique in classes and traits.

A well-formed class consists of well-typed **var** fields that satisfy the requirements from its traits, and a defined equivalence to a well-formed composite capability. We allow covariance for **val** fields (WF-CLASS). Only immutable fields holding **safe** capabilities are allowed in **read** capabilities (WF-REQ-\*, WF-FD), unless the type of the field is exposed through nesting (WF-FD-NST). Fields may not store stack-bound capabilities and fields holding thread-local values are only allowed if the containing object is also thread-local (WF-FD).

## 6.3 Well-Formed Types (Figure 7)

Capabilities corresponding to traits with manifest modes are trivially well-formed (T-TRAIT-MFST). Traits without a manifest mode can be given any mode (T-TRAIT). Well-formed **read** capabilities may only contain **safe** val fields. The empty capability  $I$  can be given any mode (T-I). Composing capabilities with  $I$  thus affects the mode of the composite, but not the interface (*cf.*, § 6.4).

Two well-formed capabilities can form a nested capability type (T-NESTING). A composite capability is well-formed if its sub-capabilities are well-formed and their shared fields are composable (T-COMPOSITION). We also require that two subordinate fields appearing on opposing sides of a conjunction  $K_1 \otimes K_2$  are not both accessible from some other trait  $K'$  in the same composite (T-REGIONS). Such a field would act as a channel that could be used to share subordinate state across the supposedly disjoint representations of  $K_1$  and  $K_2$ .

The rules of the form  $Fd_1 \odot Fd_2$  govern field overlaps between capabilities in a composite, where  $\odot \in \{\otimes, \oplus\}$  denotes the composition of the capabilities containing the fields (*cf.*, Figure 2). Disjunctions may overlap freely (C-DISJUNCTION). Disjoint fields do not overlap (C-DISJOINT). If a field appearing on both sides of a composition is mutable on one side and immutable on the other, the mutable field’s type must be more precise (C-VAR-VAL). An immutable field may appear on both sides of a composition only if its type is safe or unsafe (C-SHARABLE) or if the fields have types whose conjunction is well-formed (C-VAL-VAL). If the sharing capabilities are conjunctive, the field must not be subordinate.

## 6.4 Type Equivalence, Packing and Subtyping (Figure 8)

Class names are aliases for composite capabilities (T-EQ-CLASS-TRAIT). The order of the operands in composition of *a single kind* does not matter (T-EQ-COMMUTATIVE) and (T-

$\vdash P : t \quad \vdash Td \quad \Gamma \vdash Td \quad \vdash Cd$		<i>(Well-formedness top-level declarations)</i>	
$\frac{\text{WF-PROGRAM} \quad \forall Cd \in Cds . \vdash Cd \quad \forall Td \in Tds . \vdash Td \quad \epsilon \vdash e : t}{\vdash Cds \ Tds \ e : t}$	$\frac{\text{WF-T-TRAIT} \quad \vdash \mathbf{subord \ trait} \ T\langle t \rangle \{ Rs \ Mds \}}{\vdash \mathbf{trait} \ T\langle t \rangle \{ Rs \ Mds \}}$	$\frac{\text{WF-T-TRAIT-MFST} \quad \rho : t \vdash k \mathbf{trait} \ T\langle t \rangle \{ Rs \ Mds \}}{\vdash k \mathbf{trait} \ T\langle t \rangle \{ Rs \ Mds \}}$	
$\frac{\text{WF-T-INNARDS} \quad k \neq \mathbf{safe} \quad \Gamma, \mathbf{this} : kT \vdash Rs \quad \Gamma, \mathbf{this} : kT \vdash Mds}{\Gamma \vdash k \mathbf{trait} \ T\langle t \rangle \{ Rs \ Mds \}}$	$\frac{\text{WF-CLASS} \quad \vdash K \quad \forall Fd \in Fds . \mathbf{this} : K \vdash Fd \quad \forall \mathbf{var} f : t_1 \in \mathbf{fields}(K) . \exists \mathbf{var} f : t_2 \in Fds . t_2 \equiv t_1 \quad \forall \mathbf{val} f : t_1 \in \mathbf{fields}(K) . \exists \mathbf{var} f : t_2 \in Fds . t_2 <: t_1}{\vdash \mathbf{class} \ C = K \{ Fds \}}$		
$\Gamma \vdash R, Rs \quad \Gamma \vdash Mds, Md \quad \Gamma \vdash Fd \quad \vdash \Gamma$		<i>(Well-formed body parts)</i>	
$\frac{\text{WF-REQ-FD} \quad \Gamma \vdash Fd}{\Gamma \vdash \mathbf{require} \ Fd}$	$\frac{\text{WF-REQ-FDS} \quad \Gamma \vdash R \quad \Gamma \vdash Rs}{\Gamma \vdash Rs, R}$	$\frac{\text{WF-MDS} \quad \Gamma \vdash Md \quad \Gamma \vdash Mds}{\Gamma \vdash Mds, Md}$	$\frac{\text{WF-M-TRAIT} \quad \Gamma, x : t_1 \vdash e : t_2}{\Gamma \vdash \mathbf{def} \ m(x : t_1) : t_2 \{ e \}}$
$\frac{\text{WF-FD-NST} \quad \vdash \Gamma \quad \Gamma(\rho) = K}{\Gamma \vdash \mathbf{val} \ f : K}$	$\frac{\text{WF-FD} \quad \Gamma(\mathbf{this}) = K \quad \vdash t \quad t \neq \mathbf{S}(\_) \quad \vdash K \quad \mathbf{thread}(t) \Rightarrow \mathbf{thread}(K) \quad \mathbf{read}(K) \Rightarrow (\mathbf{mod} \equiv \mathbf{val} \wedge \mathbf{safe}(t))}{\Gamma \vdash \mathbf{mod} \ f : t}$		$\frac{\text{ENV-VAR} \quad \vdash \Gamma \quad \vdash t \quad \text{ENV-E} \quad x \notin \mathbf{dom}(\Gamma)}{\vdash \epsilon \quad \vdash \Gamma, x : t}$

■ **Figure 6** Well-formed declarations.  $\Gamma ::= \epsilon \mid \Gamma, x : t$ , ( $x$  incl.  $\rho$ ).

EQ-ASSOCIATIVE). Equivalent types in jail or bound to the stack are still equivalent (T-EQ-BOXED). A **read** with a nested conjunction can be unpacked into two **read** capabilities with nested capabilities from the unpacked conjunction (T-EQ-NESTING). Incapabilities can be added and removed from a type as long as modes are preserved (T-EQ-I).

Jailing allows creating a conjunction from a disjunction (T-JAIL). The full capability that can be unlocked from the jail is written as a subscript  $K$ . Jailing requires that all modes on the composite are preserved by the extracted capability, modulo the rules  $k_{<}$ : for unpacking to be sound. For example, **locked**  $A \oplus \mathbf{subord} \ B$  denotes a (partially) subordinate capability that is strongly encapsulated inside an aggregate. If we are allowed to forget the subordinate mode of the type, the locked sub-capability could be leaked. We employ (T-EQ-I) to this end. For example, we can compose **subord**  $I$  with **locked**  $A \oplus \mathbf{subord} \ B$  and then apply (T-JAIL) to extract **locked**  $A \oplus \mathbf{subord} \ I$  which satisfies mode preservation.

Subtyping is structural on capabilities. Subtyping must preserve modes, or encapsulation, domination or exclusivity could be lost. The rules (T-SUB-\*) allow **locked** and **read** to be abstracted by the **safe** mode.

## 6.5 Well-Typed Expressions (Figure 9)

### 6.5.1 Packing & Unpacking

The rules (E-PACK) and (E-UNPACK) govern the packing and unpacking of capabilities. They rely on the rules (T-JAIL) and (T-PACK) in Figure 8 which allow introducing aliases to

$\boxed{\vdash t} \quad (well\text{-formedness of types})$ $\frac{\text{T-CLASS} \quad \text{class } C = K \{ \_ \} \in P}{\vdash C}$ $\frac{\text{T-TRAIT-MFST} \quad k \text{ trait } T(K) \{ \_ \} \in P}{\vdash kT}$ $\frac{\text{T-BOXED} \quad \vdash K}{\vdash B(K)}$ $\frac{\text{T-NESTING} \quad \vdash K_1 \quad \vdash K_2}{\vdash K_1 \langle K_2 \rangle}$ $\frac{\text{T-TRAIT} \quad k \equiv \text{read} \Rightarrow \text{this} : \text{read } T \vdash Rs \quad \text{trait } T(K) \{ Rs \_ \} \in P}{\vdash kT}$ $\frac{\text{T-COMPOSITION} \quad \vdash K_1 \quad \vdash K_2 \quad \forall Fd_1 \in \mathbf{fields}(K_1), Fd_2 \in \mathbf{fields}(K_2) . Fd_1 \odot Fd_2 \quad \mathbf{wfRegions}(K_1, K_2) \quad \mathbf{wfRegions}(K_2, K_1)}{\vdash K_1 \odot K_2}$ $\frac{\text{T-I} \quad \vdash k\mathbf{I}}{\vdash k\mathbf{I}}$ $\frac{\text{T-REGIONS} \quad \forall K_1 \otimes K_2 \in K . \text{mod}_1 f_1 : t_1 \in \mathbf{fields}(K_1) \wedge \text{mod}_2 f_2 : t_2 \in \mathbf{fields}(K_2) \quad \wedge f_1 \neq f_2 \wedge \mathbf{subord}(t_1) \wedge \mathbf{subord}(t_2) \Rightarrow \neg (f_1 \in \mathbf{fields}(K') \wedge f_2 \in \mathbf{fields}(K'))}{\mathbf{wfRegions}(K, K')}$	$\boxed{Fd_1 \odot Fd_2} \quad (sharing fields across traits)$ $\frac{\text{C-SHARABLE} \quad \mathbf{safe}(t) \vee \mathbf{unsafe}(t)}{\mathbf{val } f : t \otimes \mathbf{val } f : t}$ $\frac{\text{C-VAR-VAL} \quad t_1 <: t_2}{\mathbf{var } f : t_1 \oplus \mathbf{val } f : t_2}$ $\frac{\text{C-VAL-VAL} \quad \vdash K_1 \otimes K_2 \quad \odot = \otimes \Rightarrow \neg \mathbf{subord}(K_1 \otimes K_2)}{\mathbf{val } f : K_1 \odot \mathbf{val } f : K_2}$ $\frac{\text{C-DISJOINT} \quad f_1 \neq f_2}{\text{mod}_1 f_1 : t_1 \odot \text{mod}_2 f_2 : t_2}$ $\frac{\text{C-DISJUNCTION} \quad \text{mod}_1 f : t \oplus \text{mod}_2 f : t}{\text{mod}_1 f : t \oplus \text{mod}_2 f : t}$
--	--

■ **Figure 7** Well-formed types. Conjunctions and disjunctions of traits are governed by the rules for overlapping fields. For simplicity, we omit (C-VAL-VAR) which is isomorphic with (C-VAR-VAL).

discrete capabilities of a conjunction and turning a disjunction into a conjunction by jailing the overlapping parts.

## 6.5.2 Linearity

**linear** capabilities are destructively read to maintain alias freedom and allow ownership transfer (E-CONS-VAR, E-CONS-FD). Method calls do not destroy **linear** receivers as an object's **this** cannot be consumed. Thus, linear capabilities are externally unique [18].

## 6.5.3 Finish–Async and Sync

Parallelism in  $\mathcal{K}$  is modelled using a scoped finish/async construct (abbreviated as **f** and **a** respectively) (E-FJ). A finish–async forks two parallel computations and waits until they have both completed. Forking a larger number of threads can be simulated using nested finish/async blocks. For simplicity we do not allow async blocks outside of a finish block, but extending the system to support unstructured parallelism or active objects is possible. We employ a frame rule that guarantees that no variable is visible in both asyncs, and that subordinate objects are only accessible to the first of the asyncs, (FRAME). This models the current thread running the first async (with access to the current **this**), and another thread running the second async block.

$$\frac{\text{FRAME} \quad \mathbf{dom}(\Gamma_2) \cap \mathbf{dom}(\Gamma_3) \equiv \emptyset \quad \nexists x : t \in \Gamma_3 . (\mathbf{subord}(t) \vee \mathbf{thread}(t)) \quad \forall x . (\Gamma_2(x) = t \Rightarrow \Gamma_1(x) = t) \wedge (\Gamma_3(x) = t \Rightarrow \Gamma_1(x) = t)}{\Gamma_1 = \Gamma_2 + \Gamma_3}$$

$t_1 \equiv t_2$				<i>(type equivalence)</i>
$\frac{\text{T-EQ-CLASS-TRAIT}}{\text{class } C = K \{ \_ \} \in P}{C \equiv K}$	$\frac{\text{T-EQ-COMMUTATIVE}}{K_1 \odot K_2 \equiv K_2 \odot K_1}$	$\frac{\text{T-EQ-ASSOCIATIVE}}{(K_1 \odot K_2) \odot K_3 \equiv K_1 \odot (K_2 \odot K_3)}$		
$\frac{\text{T-EQ-BOXED}}{K_1 \equiv K_2}{B(K_1) \equiv B(K_2)}$	$\frac{\text{T-EQ-NESTING}}{\text{read}(K_1)}{K_1 \langle K_2 \otimes K_3 \rangle \equiv K_1 \langle K_2 \rangle \otimes K_1 \langle K_3 \rangle}$	$\frac{\text{T-EQ-I}}{k(K)}{K \odot k \mathbf{I} \equiv K}$	$\frac{\text{T-EQ-TRANS}}{K_1 \equiv K_2 \quad K_2 \equiv K_3}{K_1 \equiv K_3}$	
$t_1 \rightleftharpoons t_2 \otimes t_3$				<i>(packing and unpacking)</i>
$\frac{\text{T-PACK}}{K_1 \equiv K_2 \otimes K_3}{\text{subord}(K_2) \rightleftharpoons \text{subord}(K_3)}$	$\frac{\text{T-JAIL}}{K_1 \equiv K_2 \oplus K_3}{\forall k . k(K_1) \rightleftharpoons k(K_3)}$	$\frac{\text{T-PACK-BOUND}}{K_1 \equiv K_2 \otimes K_3}{\mathbf{S}(K_1) \rightleftharpoons \mathbf{S}(K_2) \otimes \mathbf{S}(K_3)}$		
$\frac{\text{T-SUB-STRUCTURAL}}{\forall k . k(K_1 \odot K_2) \Rightarrow k_{<}(K_1)}{K_1 \odot K_2 <: K_1}$	$\frac{\text{T-SUB-BOXED}}{K_1 <: K_2}{B(K_1) <: B(K_2)}$	$\frac{\text{T-SUB-EQ}}{t_1 \equiv t_2 \quad t_2 <: t_3}{t_1 <: t_3}$	$\frac{\text{T-SUB-ID}}{t <: t}$	
$\frac{\text{T-SUB-K}}{k(K)}{k_{<}(K)}$	$\frac{\text{T-SUB-RD}}{\text{safe}(K)}{\text{read}_{<}(K)}$	$\frac{\text{T-SUB-LOCK}}{\text{safe}(K)}{\text{locked}_{<}(K)}$		

■ **Figure 8** Type equivalence, packing/unpacking, subtyping.

The **sync** keyword temporarily converts an **unsafe** (and therefore unusable) capability into a **locked** capability, acquiring and releasing locks at the entry and exit of  $e_1$  (E-SYNC).

### 6.5.4 Borrowing

Forward borrowing allows turning capabilities into stack-bound capabilities non-destructively, possibly relaxing a **linear** to a **thread** and allows splitting reads (E-FORWARD). The helper predicate **boundable**( $K_1, K_2$ ) holds in either of three cases:

1.  $K_1 = K_2$
2.  $K_1[\mathbf{linear} \mapsto \mathbf{thread}] = K_2$  (relaxing linearity to thread-affinity)
3.  $K_1 = K_2 \odot K_3$  s.t.  $K_3$  is neither **subordinate** nor **thread**, and all capabilities in  $K_2$  are **read**. The last case allows relaxing alias restrictions for stack-bound **read** capabilities which enables mediating from single writer to multiple readers across multiple threads without dynamic concurrency control for a clearly defined scope.

Reverse borrowing allows non-destructively reading a **linear** capability into a stack-bound value (E-REVERSE). Since only one value can be returned by a method, multiple reverse borrowing of the same field in the same method is innocuous. Non-linear capabilities never need to be reverse borrowed as they can always be returned normally without consuming.

### 6.5.5 Self Typing

Modulo traits with manifest modes, **this** inside a trait is always subordinate (WF-T-TRAIT). This reflects the fact that on the inside of a capability, exclusive access of a single thread is already guaranteed (*e.g.*, because the accessing thread was forced to acquire a lock to enter

$\Gamma \vdash e : t$				<i>(expression typing)</i>
$\frac{\text{E-UPCAST} \quad \Gamma \vdash e : t_2 \quad t_2 <: t_1}{\Gamma \vdash (t_1)e : t_1}$	$\frac{\text{E-NEW} \quad \vdash \Gamma \quad \text{class } C = \mathbb{K}\{\_ \} \in P \quad \text{subord}(\mathbb{K}) \Rightarrow \text{this} \in \text{dom}(\Gamma)}{\Gamma \vdash \text{new } C : \mathbb{K}}$	$\frac{\text{E-NULL} \quad \vdash \Gamma \quad \vdash t}{\Gamma \vdash \text{null} : t}$	$\frac{\text{E-LET} \quad \Gamma \vdash e_1 : t_1 \quad t_1 \neq \mathbf{S}(\_) \quad \Gamma, x : t_1 \vdash e_2 : t_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : t_2}$	
$\frac{\text{E-UNPACK} \quad \Gamma \vdash z : t_1 \quad t_1 \rightleftharpoons t_2 \otimes t_3 \quad \Gamma, x : t_2, y : t_3 \vdash e : t_4}{\Gamma \vdash \text{unpack } x + y = z \text{ in } e : t_4}$	$\frac{\text{E-PACK} \quad \Gamma \vdash y : t_1 \quad \Gamma \vdash z : t_2 \quad \text{linear}(t_1) \quad \text{linear}(t_2) \quad t_3 \rightleftharpoons t_1 \otimes t_2 \quad \Gamma, x : t_3 \vdash e : t_4}{\Gamma \vdash \text{pack } x = y + z \text{ in } e : t_4}$	$\frac{\text{E-SELECT} \quad \Gamma \vdash \text{this} : t_1 \quad \text{fields}(t_1)(f) = t_2 \quad \neg \text{linear}(t_2)}{\Gamma \vdash \text{this}.f : t_2}$		
$\frac{\text{E-CONS-FD} \quad \Gamma \vdash \text{this} : t_1 \quad \text{vars}(t_1)(f) = t_2}{\Gamma \vdash \text{consume this}.f : t_2}$	$\frac{\text{E-UPDATE} \quad \Gamma \vdash \text{this} : t_1 \quad \text{vars}(t_1)(f) = t_2 \quad \Gamma \vdash e : t_2}{\Gamma \vdash \text{this}.f = e : t_2}$	$\frac{\text{E-VAR} \quad \vdash \Gamma \quad \Gamma(x) = t \quad \neg \text{linear}(t)}{\Gamma \vdash x : t}$	$\frac{\text{E-CONS-VAR} \quad \vdash \Gamma \quad x \neq \text{this} \quad \Gamma(x) = t}{\Gamma \vdash \text{consume } x : t}$	
$\frac{\text{E-CALL} \quad \text{linear}(t_1) \Rightarrow x \notin \text{freeVars}(e) \quad \Gamma(x) = t_1 \quad \neg \text{unsafe}(t_1) \quad \text{msigs}(t_1)(m) = z : t_2 \rightarrow t_3 \quad \Gamma \vdash e : t_2 \quad (\text{subord}(t_2) \vee \text{subord}(t_3)) \Rightarrow \text{encaps}(t_1) \vee x \equiv \text{this}}{\Gamma \vdash x.m(e) : t_3}$		$\frac{\text{E-FJ} \quad \Gamma = \Gamma_1 + \Gamma_2 \quad \Gamma \vdash e : t \quad \Gamma_1 \vdash e_1 : \_ \quad \Gamma_2 \vdash e_2 : \_}{\Gamma \vdash \mathbf{f} \{ \mathbf{a} \{ e_1 \} \mathbf{a} \{ e_2 \} \}; e : t}$		
$\frac{\text{E-SYNC} \quad \Gamma \vdash x : \text{unsafe T} \quad \Gamma, y : \mathbf{S}(\text{locked T}) \vdash e_1 : \_ \quad \Gamma \vdash e_2 : t}{\Gamma \vdash \text{sync } x \text{ as } y \{ e_1 \}; e_2 : t}$	$\frac{\text{E-REVERSE} \quad \Gamma \vdash \text{this} : t \quad \text{linear}(t) \quad \text{fields}(t)(f) = \mathbb{K}}{\Gamma \vdash \text{this}.f : \mathbf{S}(\mathbb{K})}$	$\frac{\text{E-FORWARD} \quad \text{boundable}(\mathbb{K}_1, \mathbb{K}_2) \quad \Gamma, x : \mathbf{S}(\mathbb{K}_2) \vdash e_1 : \_ \quad \Gamma, x : \mathbb{K}_1 \vdash e_2 : t}{\Gamma, x : \mathbb{K}_1 \vdash \text{bound } x \{ e_1 \}; e_2 : t}$		

■ **Figure 9** Typing of expressions. Note that all fields are “private”.

the object, or place a linear entry point in a stack variable, which is analogous).

Viewing **this** as subordinate allows an object to be aliased freely from *inside* its own enclosure, including objects of linear capabilities. Thus, linear capabilities in  $\mathcal{K}$  are externally unique [18]. Traits with manifest modes have more information about themselves internally.

## 7 Meta-Theoretic Evaluation

In this section, we describe the key invariants of our capabilities in a well-typed  $\mathcal{K}$  program. The full proofs and definitions have been relegated to the appendix. This section aims to explain the key properties, and sketching why they hold.

### 7.1 $\mathcal{K}_F$ and the Dynamic Semantics of $\mathcal{K}$ Programs

To execute  $\mathcal{K}$  programs, they are translated into a simpler language,  $\mathcal{K}_F$ , with classes and interfaces without traits or capabilities. The semantics of  $\mathcal{K}_F$  is straightforward, and only the most relevant details are presented here.

Flattening of traits [40] is performed similar to other trait-based languages: A class is translated by copying the methods from its traits, traits are translated into interfaces with the equivalent signature. For each composition  $K_1 \odot K_2$ , an interface is created extending the interfaces corresponding to  $K_1$  and  $K_2$ . This preserves the same subtyping rules as in  $\mathcal{K}$ . For each  $\mathcal{K}$ -capability in a translated program, there is a single corresponding interface. Because of this one-to-one mapping, we can easily recover  $\mathcal{K}$  types from a  $\mathcal{K}_F$  program, which we use extensively in our proofs.

From field overlaps, we infer a set of regions for each class, and insert lock and unlock instructions at the start and end of methods to acquire and release the lock for the region touched by the method. Read locks are acquired in methods in read capabilities if they overlap with a locked or unsafe capability. Methods in locked capabilities acquire write locks. All locks are reentrant. Well-formedness of  $\mathcal{K}_F$  configurations requires that no two threads can hold the same writer lock, and that all locks held by a thread are also taken in the objects themselves. This assures mutual exclusion in all parts of a program that use locks.

A well-formed  $\mathcal{K}$  program will always translate into a well-formed  $\mathcal{K}_F$  program. This is easy to prove as most type rules for  $\mathcal{K}_F$  are subsumed by the  $\mathcal{K}$  type rules. Thus, by proving type soundness (progress and preservation) of  $\mathcal{K}_F$ , we show that a translated  $\mathcal{K}$  program will never get stuck (modulo deadlocks, which we distinguish from unsound stuck states).

$\mathcal{K}_F$  imposes no restrictions on aliasing nor does it provide any guarantees about race freedom. However,  $\mathcal{K}_F$  programs translated from well-formed  $\mathcal{K}$  programs will always be data-race free. As  $\mathcal{K}_F$  itself provides no data-race guarantees, the invariants given by  $\mathcal{K}$  are defined independent of well-formed configurations of  $\mathcal{K}_F$  (see §7.2 and §7.3).

We have a fully mechanised specification of  $\mathcal{K}_F$  in Coq, including a proof of type soundness, but not data-race freedom [15]. In our hand-written proof of data-race freedom we also extend  $\mathcal{K}_F$  with means of tracking types and stack-boundedness of values. This has no effect on the execution or typechecking of  $\mathcal{K}_F$  programs and is thus excluded from the mechanised version. Specifying all of  $\mathcal{K}$  in Coq is future work.

## 7.2 Aliasing and Encapsulation

This section details the invariants on aliasing and encapsulation in well-typed  $\mathcal{K}$  programs.

### 7.2.1 Safe Aliasing

One of the main technical results of this work is the proof that  $\mathcal{K}$  programs enjoy *safe aliasing*, *i.e.*, two paths to the same mutable field are local to the same thread, or protected by the same lock which must be acquired before access. Informally, aliasing is safe if the following is true for all aliases  $x, y$  on the stack or heap:

- $x$  and  $y$  have composable types, meaning they point to different parts of the same object, modulo safe **val** fields, corresponding to  $\vdash t_x \otimes t_y$  in  $\mathcal{K}$  (*cf.*, Figure 7).
- $x$  and  $y$  are protected, *i.e.*, read-only aliases, or safe aliases that use locks internally, or unsafe aliases whose accesses must be wrapped in locks.
- $x$  and  $y$  are both local to the same thread, corresponding to the **thread** capabilities of  $\mathcal{K}$ .
- $x$  and  $y$  both have subordinate types, meaning that any thread accessing them must currently have exclusive access to their dominator.
- If  $x$  or  $y$  is **linear**, at least one of the aliases must be stack-bound to prevent introducing aliases of linear values on the heap.
- If one of the aliases is stack-bound, the origins of the borrowed value must be buried, to prevent multiple accessible references to the same linear value.

## 7.2.2 Proof

Part of the proof of thread-safety (*cf.*, §H.7).

## 7.2.3 Strong Encapsulation of Subordinate Capabilities

Another invariant preserved by  $\mathcal{K}$  programs is that references in fields of subordinate type point to objects dominated by the dominator of the current enclosure. This is what grants subordinate capabilities strong encapsulation, similar to ownership types [17] and external uniqueness [18].

At run-time in  $\mathcal{K}_F$ , instances know the identity of their dominator. This identity is invariant, even under ownership transfer, because transfer operates on linear capabilities and instances of classes without a subordinate capability are their own dominators.

Let  $\rightarrow$  denote “refers to” and  $\iota.\text{dom}$  denote the dominator for an object with id  $\iota$  in the heap  $H$ . Now,  $\forall \iota, \iota' \in \text{dom}(H)$ ,  $\iota \rightarrow \iota'$  s.t.  $\iota \neq \iota'$ , either one of the following holds:

1.  $\iota'.\text{dom} = \iota.\text{dom}$  (a pointer between subordinates in the same enclosure)
2.  $\iota'.\text{dom} = \iota$  (a dominator pointing to one of its subordinates)
3.  $\iota.\text{dom} = \iota'$  (a subordinate pointing to its dominator)

or  $\iota'$  is a top-level object, *i.e.*,  $\iota'.\text{dom} = \iota'$ .

## 7.2.4 Proof

Part of the proof of thread-safety (*cf.*, §H.7).

## 7.3 Data-Race Freedom and Non-Interference

This section describes the invariants of  $\mathcal{K}$  for concurrent and parallel programming.

### 7.3.1 Thread-Safety

Safe aliasing and the encapsulation guarantees mentioned above are both part of a bigger notion of a *thread-safe* (**TS**) configuration. A configuration is thread-safe if no two possible reductions can cause interference—if a possible reduction of a configuration has one thread writing to a field, there cannot be another reduction of the same configuration where the same field is read or written to by another thread.

In addition to safe aliasing, a number of constraints apply to the elements of thread-safe configurations that deal specifically with aliasing across threads:

- references in fields of subordinate type point to objects dominated by the dominator of the current enclosure;
- values of type **thread** were created by the thread that can access them;
- all local variables of **subordinate** type are dominated by the closest dominating **this** on the current stack;
- if a stack-bound **linear** value aliases a value on the heap, the value on the heap is effectively buried, *i.e.*, the only path to it is rooted on the stack;
- all accesses to values not wrapped in locks are **linear**, **thread**, **subord** or **read**.

Having defined thread-safety, we then prove that the initial configuration is **TS** and that evaluation preserves this property in a program translated from  $\mathcal{K}$  to  $\mathcal{K}_F$ .

► **Preservation of Thread-Safety.** In a well-formed program translated from  $\mathcal{K}$ , if a thread-safe configuration  $cfg$  can step to  $cfg'$ , then  $cfg'$  is also thread-safe.

$$\forall \Gamma, cfg, cfg'. \\ \Gamma \vdash \mathbf{TS}(cfg) \wedge cfg \hookrightarrow cfg' \Rightarrow \exists \Gamma'. \Gamma' \vdash \mathbf{TS}(cfg') \wedge \Gamma \subseteq \Gamma'$$

### 7.3.2 Proof

We prove thread-safety by induction over the thread structure ( $cf$ , §H.7). The proof is similar in structure to the type preservation proof of  $\mathcal{K}_F$  (and relies on type preservation to find  $\Gamma'$ ).

### 7.3.3 Data-Race Freedom

The dynamic semantics of  $\mathcal{K}_F$  tracks effect footprints in terms of reads ( $\mathbf{rd}(\_)$ ) and writes ( $\mathbf{wr}(\_)$ ) of object fields. This allows us to define and prove data-race freedom, which verifies that our notion of thread-safety is sound.

► **Data-Race Freedom.** If a safe configuration  $cfg$  steps to two different configurations causing effects  $Eff_1$  and  $Eff_2$  respectively, then these effects are non-conflicting:

$$\forall \Gamma, cfg, cfg' \text{ } cfg''. \\ \Gamma \vdash \mathbf{TS}(cfg) \wedge cfg \hookrightarrow^{Eff_1} cfg' \wedge cfg \hookrightarrow^{Eff_2} cfg'' \Rightarrow Eff_1 \# Eff_2 \vee cfg' = cfg''$$

where  $\#$  denotes that two effects are disjoint or a read–read conflict:

$$\begin{array}{l} \mathbf{rd}(l.f) \# \mathbf{rd}(l'.f') \\ \varepsilon \# \_ \end{array} \quad \begin{array}{l} \_ (l.f) \# \mathbf{wr}(l'.f') \\ Eff_1 \# Eff_2 \end{array} \text{ iff } l \neq l' \vee f \neq f' \\ \text{ iff } Eff_2 \# Eff_1$$

### 7.3.4 Proof

The proof is straightforward and performed by case analysis on the thread structure, showing that any interference contradicts thread-safety ( $cf$ , §I).

### 7.3.5 Corollary: Non-Interference

Parallel  $\mathcal{K}$  expressions that do not use locked or unsafe capabilities are *free from interference and therefore deterministic*. The only way threads can affect each other is by writing shared mutable locations, which requires taking a lock. Without locked or unsafe capabilities there are no locks, meaning that any data that is shared between threads can only be read, and that no threads are ever blocked in their execution.

### 7.3.6 Corollary: Thread-Affinity of Thread Capabilities

Implied by **TS**,  $\mathcal{K}$  **thread** capabilities are thread-affine. Let  $creator(\iota)$  return the id of the thread creating  $\iota$ . From **TS** follows that in a thread  $tid$  with local variables  $V$  and expression  $e$ ,  $\forall x. V(x) = \iota \wedge \Gamma(x) = t \wedge \mathbf{thread}(t) \Rightarrow creator(\iota) = tid$  and  $\forall \iota. \iota \in \mathbf{locations}(e) \wedge \Gamma(x) = t \wedge \mathbf{thread}(t) \Rightarrow creator(\iota) = tid$ . Thus **thread** capabilities are only visible to their creating threads. The key elements in the type system are the  $\mathbf{thread}(t) \Rightarrow \mathbf{thread}(\mathbb{K})$  constraint in the  $\mathcal{K}$  rule (**WF-FD**) which restrict fields of type **thread** to only appear inside manifestly **thread** capabilities and the  $\mathcal{K}$  rule (**FRAME**) which does not allow **thread** capabilities to be visible in the second **asyn**c of a **finish**.

## 8 Related Work

An original source of inspiration for this work was Boyland *et al.*'s “Capabilities for Sharing” which introduces a system of reference capabilities, such as immutability or ownership in a dynamic system, not amenable to static typing [10]. Similarly,  $\kappa$  brings together ideas from many different areas in a single system, but fully statically typed. To the best of our knowledge, the selection and integration of the features in  $\kappa$  are unique in an object setting:

1. Linear capabilities are similar to uniqueness [28, 35] or permissions [9, 43, 39] and enable ownership transfer [18].
2. Subordinate capabilities enable strong encapsulation similar to ownership types [17] or Universes [37, 36].  $\kappa$ 's combination of subordinate and dominating capabilities give arbitrary nesting, but nested aggregates may not refer to subordinate objects in their enclosing aggregate, nor does  $\kappa$  support incoming read-only references (owners-as-modifiers [37, 36])—both enable data-races.
3. Thread capabilities resemble thread-local heaps in Loci [44] and ownership for actors [19].
4. That linear capabilities view themselves as subordinate capabilities internally gives a form of external uniqueness [18, 27].
5. The combination of locked and read capabilities express readers–writer locks, with a compile-time guarantee that readers will not write.
6. The **safe** mode, abstracting over **read** and **locked**, avoids code duplication for traits that are agnostic to why objects are safe, similar to type qualifier generics [23, 46].
7. The combination of locked and subordinate capabilities empowers a single lock to range over an entire aggregate with a compile-time guarantee of correctness (*cf.*, owners-as-locks in *e.g.*, [17]), it also allows enforcing a crude form of lock ordering in combination with readers-writer locks by connecting lock order to nesting order (*cf.*, [7]).
8. Nested capabilities are essentially storable permissions [9, 43] but without breaking abstraction—the names of fields etc. of the object storing permissions can be kept secret.
9. The flat composition of capabilities and the packing/unpacking marries ideas from fractional permissions [9] with ownership and substructural types [13], similar to [30]. Composites of read and linear capabilities support mediation between readers and writers, using stack-bounding to identify where sharing starts and stops.

### 8.1 Ownership Systems

Aliasing of mutable state in object-oriented programming is a mature research field: categorisations of alias management techniques [29], ownership types [17], universe types [23, 36], external uniqueness [18], balloons [2], as well as multiple flavours of references [28, 4, 35, 3, 10, 8, 12] etc. (*cf.* [17] for a broad coverage of many aspects). Banning aliasing is usually abandoned in favour of alias control, which commonly prescribes a certain shape on the program [17, 23, 44, 19] possibly combined with an effect system to coordinate accesses to shared data across multiple program locations [20, 5, 14].  $\kappa$  does not prescribe a certain topology for shared capabilities. With the shift to ubiquitous parallelism, ideas from these fields have been applied to the simplification of concurrent and parallel programming (*e.g.*, [7, 19, 27, 25, 38, 22]).

Abadi *et al.* [1] propose RaceFree Java where field declarations are associated with locks and an effect system tracks how locks are acquired and released. Classes can be parameterised with external locks. The combination of locked and subordinate capabilities in  $\kappa$  seem to be able to express the same, but without ghost variables or an effect system. Zhao [45]

constructs a system similar to Abadi *et al.* [1] but based on fractional permissions. It uses method annotations with read/write effects and locks taken and also considers deadlocks through lock ordering, similar to [7].

The recent surge of interest in Mozilla’s Rust language provides anecdotal evidence to the value of languages with data-race freedom built in. In Rust, values mediate between linear/mutable and sharable/immutable. Linear values use transfer semantics and Rust uses borrowing to simplify programming. Rust support flat unpacking of arrays, but not nested unpacking and not unpacking of other types. Rust does not have strong encapsulation meaning an aggregate’s innards is not protected by the aggregate’s single entry-point and no aliasing of mutable objects is allowed inside the aggregate.

## 8.2 Substructural Systems

$\kappa$  is close in spirit to work by Caires and Seco [13] as well as work by Pottier *et al.* [39] on Mezzo, both in the context of ML-like languages. Caires and Seco formalise a fine-grained capability system for reasoning about interference caused by aliasing or concurrent accesses to aliased data with explicit synchronisation. There is no support for read-sharing or strong non-linear encapsulation.

Militão *et al.* use a substructural type system for specifying rely-guarantee protocols in a functional context [33]. Protocols capture the view of shared state from one particular alias—our capabilities are similar. Ownership transfer and recovery for linear values is supported.

The functional language Alms [41] explores the design space of practical programming with substructural types. Alms separates capabilities from references and operations that require a capability must have the capability passed in as an argument. Capabilities in Alms are lower level than in  $\kappa$  and can be used to express many of our capabilities. There is no unification of capabilities with building blocks like traits, or composition of capabilities.

## 8.3 Capability Systems and Permission Systems

Miller uses capabilities for access control and concurrency control in a distributed setting in the seminal E language [34], employing a more dynamic approach (than  $\kappa$ ) with optional soft typing.

Mediation between different views of an object is similar to fractional permissions [9].  $\kappa$  supports going from a single writer to multiple enumerable disjoint writers or readers, and in the case of readers to an unbounded number of stack-bound aliases via (E-FORWARD). Fractional permissions with nesting [11, 43] is similar to subordinate capabilities in allowing one permission to act as guard to another. These system allows turning an entire nested structure read-only.  $\kappa$ ’s subordinate capabilities are less restricted, but also not transferable.  $\kappa$ ’s **read** capabilities also provide abstraction as they allow fields remain private.

Bocchino’s Deterministic Parallel Java [5] uses an effect system to guarantee deterministic parallelism for operations that have no overlapping writes. When the effect system is not enough, the user can annotate trusted operations as commuting.  $\kappa$  provides similar determinism guarantees when excluding locking capabilities, but resorts to locking (and non-determinism) rather than using unchecked annotations for more complex operations.

Clebsch *et al.* [21] use “deny capabilities” to provide safe sharing of objects between actors. Their capabilities always grant exclusive write access to entire objects, while  $\kappa$ ’s also allows accessing parts of an object, as well as permitting multiple parallel writers.

Westbrook *et al.* [43] formalise and implement a gradual extension to HabaneroJava, HJp, in the form of a permission system. Permissions in HJp always govern access to entire objects,

and there is no notion of encapsulation modulo storing linear permission in fields which only supports tree-shaped data. When there is not enough permission information, dynamic checks are inserted which may fail, but which also allow unconstrained aliasing. Splitting a single write-permission into multiple read permissions is similar to **read** capabilities.

Chalice by Leino *et al.* [31] is a language for specification and verification of concurrent software that uses permissions to statically track aliasing. Since  $\mathcal{K}$  is concerned only with data-race freedom, it trades some of the more fine-grained control (*e.g.*, full method specification) for simplicity (*e.g.*, no need for manual permission tracking).

## 9 Discussion & Future Work

We currently require programmers to explicitly manage substructural operations manually through packing and unpacking and jails. Building simpler-to-use constructs on top of these is possible. For example, a combination of **bound** and **unpack** would remove the need for packing, at the cost of enforcing a nested structure to unpacking and packing. Employing inference to automate this to a large extent seems possible and is a direction for future work.

Implementation of  $\mathcal{K}$  is on-going in the actor-based language Encore. There, actors replace locks as a means of pessimistic concurrency control. Capabilities protect actors' state while allowing ownership-transfer due to linear values. A larger case study evaluating the full expressiveness of  $\mathcal{K}$  is planned for future work.

### 9.1 Unstructured vs. Structured

We have purposely supported *unstructured* packing and unpacking of capabilities. This allows granting capabilities to other threads (possibly on other machines) without requiring the capabilities to be returned or tying their return to a particular local scope. This removes limitations inherent in effect systems (*e.g.*, [26, 20, 5]), which requires computations to be nested. Unstructured locking is important in some applications, for example to implement hand-over-hand locking, and is a possible direction for future work.

### 9.2 Revocation

We only consider “cooperative revocation”, *i.e.*, there is no built-in mechanism to arbitrarily revoke a given capability. In the security setting from which the capability idea stems, this is a major concern but it makes less sense in our setting as revoking a capability from another thread at an unfortunate point in time might cause system-wide inconsistencies.

### 9.3 Other Capabilities

The only form of dynamic concurrency control considered in this paper is locks. In ongoing work, the set of modes are extended with **async** (objects are actors), **atomic** (objects use transactional memory) and **lockfree** (lock-free programming). In this richer setting, we aim to address more programmer-friendly forms of multi-object atomicity.

## 10 Conclusions

Creating and destroying aliases enables and constrains parallelism and is key to establishing data-race freedom and non-interference. By capturing how data is shared and accessed through modes, and by introducing a structured approach to creating and destroying aliases

through the combination of capabilities that make up classes, data-race freedom can be guaranteed statically with or without dynamic concurrency control.  $\kappa$ 's invariants are similar to what an effect system can give, but avoids complicated effect annotations which propagate through the program and constrain inheritance. Tracking modes in types provides machine-checked documentation about alias freedom and sharing which localises reasoning.

The unification of traits and capabilities allows a single trait to serve multiple concurrency scenarios, which extends trait-based reuse. It also simplifies programming as trait implementers may safely assume data-race freedom. Ultimately,  $\kappa$  brings together a broad spectrum of prior work in a unified system.

**Acknowledgements** We are grateful for the comments from Sophia Drossopoulou, the SLURP reading group at Imperial College, Dave Clarke, and the anonymous reviewers of ECOOP.

---

## References

- 1 M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, Mar. 2006.
- 2 P. S. Almeida. *Control of Object Sharing in Programming Languages*. PhD thesis, Imperial College London, June 1998.
- 3 D. F. Bacon, R. E. Strom, and A. Tarafdar. Guava: a Dialect of Java Without Data Races. In *OOPSLA*, pages 382–400, 2000.
- 4 H. G. Baker. 'Use-once' Variables and Linear Objects – Storage Management, Reflection and Multi-Threading. *ACM SIGPLAN Notices*, 30(1), Jan. 1995.
- 5 R. Bocchino. An Effect System and Language for Deterministic-By-Default Parallel Programming, 2010. PhD thesis, University of Illinois at Urbana-Champaign.
- 6 S. Borkar and A. A. Chien. The future of microprocessors. *CACM*, 54(5):67–77, 2011.
- 7 C. Boyapati, R. Lee, and M. C. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*, pages 211–230, 2002.
- 8 J. Boyland. Alias burying: Unique variables without destructive reads. *Software—Practice and Experience*, 31(6):533–553, May 2001.
- 9 J. Boyland. Checking interference with fractional permissions. In *SAS*, pages 55–72, 2003.
- 10 J. Boyland, J. Noble, and W. Retert. Capabilities for Sharing: A Generalisation of Uniqueness and Read-Only. In *ECOOP*. Springer, 2001.
- 11 J. T. Boyland. Semantics of fractional permissions with nesting. *ACM Trans. Program. Lang. Syst.*, 32(6):22:1–22:33, Aug. 2010.
- 12 J. T. Boyland and W. Retert. Connecting Effects and Uniqueness with Adoption. In *POPL*, pages 283–295, 2005.
- 13 L. Caires and J. a. C. Seco. The Type Discipline of Behavioral Separation. In *POPL*, 2013.
- 14 N. R. Cameron, S. Drossopoulou, J. Noble, and M. J. Smith. Multiple ownership. In *OOPSLA*, 2007.
- 15 E. Castegren. Coq sources of KappaF mechanisation. <https://github.com/EliasC/kappaf>.
- 16 E. Castegren and T. Wrigstad. Reference Capabilities for Concurrency Control. In *ECOOP*, 2016.
- 17 D. Clarke, J. Östlund, I. Sergey, and T. Wrigstad. Ownership Types: A Survey. In *Aliasing in Object-Oriented Programming*, volume 7850 of *LNCS*. Springer, 2013.
- 18 D. Clarke and T. Wrigstad. External uniqueness is unique enough. In *ECOOP*, 2003.
- 19 D. Clarke, T. Wrigstad, J. Östlund, and E. Johnsen. Minimal ownership for active objects. In *Programming Languages and Systems*, volume 5356 of *LNCS*. Springer, 2008.

- 20 D. G. Clarke and S. Drossopoulou. Ownership, Encapsulation and the Disjointness of Type and Effect. In *OOPSLA*, pages 292–310, 2002.
- 21 S. Clebsch, S. Drossopoulou, S. Blessing, and A. McNeil. Deny capabilities for safe, fast actors. In *AGERE*, 2015.
- 22 D. Cunningham, S. Drossopoulou, and S. Eisenbach. Universe Types for Race Safety. In *VAMP*, 2007.
- 23 W. M. Dietl. *Universe Types: Topology, Encapsulation, Genericity, and Tools*. Ph.D., Department of Computer Science, ETH Zurich, Dec. 2009.
- 24 M. Fähndrich and R. DeLine. Adoption and Focus: Practical Linear Types for Imperative Programming. In *PLDI*, pages 13–24, 2002.
- 25 C. S. Gordon, M. J. Parkinson, J. Parsons, A. Bromfield, and J. Duffy. Uniqueness and Reference Immutability for Safe Parallelism. In *OOPSLA*, pages 21–40, 2012.
- 26 A. Greenhouse and J. Boyland. An object-oriented effects system. In *ECOOP*, 1999.
- 27 P. Haller and M. Odersky. Capabilities for uniqueness and borrowing. In *ECOOP*, 2010.
- 28 J. Hogg. Islands: Aliasing Protection in Object-Oriented Languages. In *OOPSLA*, 1991.
- 29 J. Hogg, D. Lea, A. Wills, D. de Champeaux, and R. Holt. The Geneva Convention on the Treatment of Object Aliasing. *OOPS Messenger*, 3(2), Apr. 1992.
- 30 N. R. Krishnaswami, A. Turon, D. Dreyer, and D. Garg. Superficially Substructural Types. In *ICFP*, New York, NY, USA, 2012. ACM.
- 31 K. R. M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. In *Foundations of Security Analysis and Design V*. 2009.
- 32 H. Levy, editor. *Capability Based Computer Systems*. Digital Press, 1984.
- 33 F. Militão, J. Aldrich, and L. Caires. Rely-guarantee protocols. In *ECOOP*, 2014.
- 34 M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, 2006.
- 35 N. Minsky. Towards alias-free pointers. In *ECOOP*, July 1996.
- 36 P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer, 2002.
- 37 P. Müller and A. Poetzsch-Heffter. Universes: a type system for controlling representation exposure. Technical Report 263, 1999. Fernuniversität Hagen.
- 38 P. Permandla, M. Roberson, and C. Boyapati. A type System for Preventing Data Races and Deadlocks in the Java Virtual Machine Language. In *LCTES*, pages 1–10, 2007.
- 39 F. Pottier and J. Protzenko. Programming with Permissions in Mezzo. In *ICFP*, pages 173–184, Sept. 2013.
- 40 N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behaviour. In *ECOOP*, 2003.
- 41 J. A. Tov. *Practical Programming with Substructural Types*. PhD thesis, Northeastern University, 2012.
- 42 P. Wadler. Linear Types Can Change the World! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods*. North Holland, 1990.
- 43 E. Westbrook, J. Zhao, Z. Budimli, and V. Sarkar. Practical permissions for race-free parallelism. In *ECOOP*, 2012.
- 44 T. Wrigstad, F. Pizlo, F. Meawad, L. Zhao, and J. Vitek. Loci: Simple thread-locality for Java. In *ECOOP*, 2009.
- 45 Y. Zhao. *Concurrency Analysis Based On Fractional Permission System*. PhD thesis, University of Wisconsin – Milwaukee, 2007.
- 46 Y. Zibin, A. Potanin, S. Artzi, et al. Object and reference immutability using Java generics. In *ESEC/FSE*. 2007.

## Appendix

### Overview of the Additional Material

**Figure 10–13** Helper predicates and functions.

**§A** Full code for the tree example from §5.

**§B** Additional example: Linked list.

**§C** Additional example: Initialisation of a shared data structure.

**§D** Static semantics of  $\mathcal{K}_F$ .

**§E** Dynamic semantics of  $\mathcal{K}_F$ .

**§F** Overview of proof of type soundness of  $\mathcal{K}_F$  (see also Coq sources).

**§G** Translation of  $\mathcal{K}$  programs into  $\mathcal{K}_F$  programs.

**§H** Formal definition and proof of thread-safe configuration and safe aliasing.

**§I** Formal definition and proof of data-race freedom.

**§J** Proof of strong encapsulation.

$k(t)$	<i>(Type has capability k)</i>		
	$\frac{\text{K-BOXED } k(K)}{k(B(K))}$		
$\mathbf{linear}(t)$	<i>(A capability is linear if any subcapability is linear, including nesting)</i>		
$\frac{\text{K-LINEAR}}{\mathbf{linear}(\mathbf{linear} T)}$	$\frac{\text{K-LI-FLAT } \mathbf{linear}(K_1) \vee \mathbf{linear}(K_2)}{\mathbf{linear}(K_1 \odot K_2)}$	$\frac{\text{K-LI-NESTING } \mathbf{linear}(K_1) \vee \mathbf{linear}(K_2)}{\mathbf{linear}(K_1 \langle K_2 \rangle)}$	
$\mathbf{read}(t)$	<i>(A capability is read if it is composed of read capabilities only, modulo nesting)</i>		
$\frac{\text{K-READ}}{\mathbf{read}(\mathbf{read} T)}$	$\frac{\text{K-RD-FLAT } \mathbf{read}(K_1) \quad \mathbf{read}(K_2)}{\mathbf{read}(K_1 \odot K_2)}$	$\frac{\text{K-RD-NESTING } \mathbf{read}(K_1)}{\mathbf{read}(K_1 \langle K_2 \rangle)}$	
$\mathbf{locked}(t)$	<i>(A capability is locked if it is safe and composed of at least one locked capability)</i>		
$\frac{\text{K-LOCKED}}{\mathbf{locked}(\mathbf{locked} T)}$	$\frac{\text{K-LO-FLAT-L } \mathbf{locked}(K_1) \quad \mathbf{safe}(K_2)}{\mathbf{locked}(K_1 \odot K_2)}$	$\frac{\text{K-LO-FLAT-R } \mathbf{safe}(K_1) \quad \mathbf{locked}(K_2)}{\mathbf{locked}(K_1 \odot K_2)}$	$\frac{\text{K-LO-NESTING } \mathbf{locked}(K_1)}{\mathbf{locked}(K_1 \langle K_2 \rangle)}$
$\mathbf{subord}(t)$	<i>(A capability is subordinate if any subcapability is subordinate, including nesting)</i>		
$\frac{\text{K-SUBORDINATE}}{\mathbf{subord}(\mathbf{subord} T)}$	$\frac{\text{K-SUB-FLAT } \mathbf{subord}(K_1) \vee \mathbf{subord}(K_2)}{\mathbf{subord}(K_1 \odot K_2)}$	$\frac{\text{K-SUB-NESTING } \mathbf{subord}(K_1) \vee \mathbf{subord}(K_2)}{\mathbf{subord}(K_1 \langle K_2 \rangle)}$	
$\mathbf{unsafe}(t)$	<i>(A capability is unsafe if any subcapability is unsafe, modulo nesting)</i>		
$\frac{\text{K-UNSAFE-ATOM}}{\mathbf{unsafe}(\mathbf{unsafe} T)}$	$\frac{\text{K-UNSAFE-FLAT } \mathbf{unsafe}(K_1) \vee \mathbf{unsafe}(K_2)}{\mathbf{unsafe}(K_1 \odot K_2)}$	$\frac{\text{K-UNSAFE-NESTING } \mathbf{unsafe}(K_1)}{\mathbf{unsafe}(K_1 \langle K_2 \rangle)}$	
$\mathbf{thread}(t)$	<i>(A capability is thread if any subcapability is thread, including nesting)</i>		
$\frac{\text{K-THREAD-ATOM}}{\mathbf{thread}(\mathbf{thread} T)}$	$\frac{\text{K-THREAD-FLAT } \mathbf{thread}(K_1) \vee \mathbf{thread}(K_2)}{\mathbf{thread}(K_1 \odot K_2)}$	$\frac{\text{K-THREAD-NESTING } \mathbf{thread}(K_1) \vee \mathbf{thread}(K_2)}{\mathbf{thread}(K_1 \langle K_2 \rangle)}$	

■ **Figure 10** Helper predicates for determining capability modes.

**encaps**( $t$ ) *(A capability is encapsulated if all non-nested subcapabilities are subordinate)*

$$\frac{\text{K-ENCAPS-ATOM}}{\text{encaps}(\text{subord } T)} \quad \frac{\text{K-ENCAPS-FLAT} \quad \text{encaps}(K_1) \quad \text{encaps}(K_2)}{\text{encaps}(K_1 \odot K_2)} \quad \frac{\text{K-ENCAPS-NESTING} \quad \text{encaps}(K_1)}{\text{encaps}(K_1 \langle K_2 \rangle)}$$

**safe**( $t$ ) *(A capability is safe if it is any combination of read, locked and safe)*

$$\frac{\text{K-SA-ATOM}}{\text{safe}(\text{safe } T)} \quad \frac{\text{K-SA-LOCKED} \quad \text{locked}(K_1)}{\text{safe}(K_1)} \quad \frac{\text{K-SA-READ} \quad \text{read}(K_1)}{\text{safe}(K_1)} \quad \frac{\text{K-SA-FLAT} \quad \text{safe}(K_1) \quad \text{safe}(K_2)}{\text{safe}(K_1 \odot K_2)}$$

■ **Figure 11** Helper predicates for determining capability modes, continued.

**dom**( $V$ ) =  $\{x_1, \dots, x_n\}$  *(domain of stack)*

$$\frac{\text{STACK-DOM-E}}{\text{dom}(\epsilon) = \emptyset} \quad \frac{\text{STACK-DOM-VAR}}{\text{dom}(V, x \mapsto v) = \text{dom}(V) \cup \{x\}}$$

**dom**( $\Gamma$ ) =  $\{x_1, \dots, x_n\}$  *(domain of environment)*

$$\frac{\text{ENV-DOM-E}}{\text{dom}(\epsilon) = \emptyset} \quad \frac{\text{ENV-DOM-VAR}}{\text{dom}(\Gamma, x : t) = \text{dom}(\Gamma) \cup \{x\}}$$

**msigs**( $t$ ) =  $\{m_1 :: x_1 : t_1 \rightarrow t_2, \dots, m_n :: x_n : t_1 \rightarrow t_2\}$  *(Method lookup from a type)*

$$\frac{\text{M-TRAIT} \quad \text{trait } T \langle K \rangle \{ \_Md_1, \dots, Md_n \} \in P}{\text{msigs}(k T \langle K' \rangle) = \{ \text{extract}(K, K', Md_1), \dots, \text{extract}(K, K', Md_n) \}}$$

$$\frac{\text{M-MANIFEST} \quad k \text{ trait } T \langle K \rangle \{ \_Md_1, \dots, Md_n \} \in P}{\text{msigs}(k T \langle K' \rangle) = \{ \text{extract}(K, K', Md_1), \dots, \text{extract}(K, K', Md_n) \}}$$

$$\frac{\text{M-COMPOSITE}}{\text{msigs}(K_1 \odot K_2) = \text{msigs}(K_1) \cup \text{msigs}(K_2)} \quad \frac{\text{M-CLASS} \quad \text{class } C = K \{ \_ \} \in P}{\text{msigs}(C) = \text{msigs}(K)} \quad \frac{\text{M-I}}{\text{msigs}(k \mathbf{I}) = \emptyset}$$

$$\frac{\text{M-JAILED}}{\text{msigs}(\mathbf{J}_{K_1}(K_2)) = \emptyset} \quad \frac{\text{M-STACKBOUND}}{\text{msigs}(\mathbf{S}(K)) = \text{msigs}(K)}$$

$$\frac{\text{M-SIGN} \quad t' = t[K \mapsto K']}{\text{extract}(K, K', \text{def } m(x_1 : t_1) : t \{ e \}) = m :: x_1 : t_1 \rightarrow t'}$$

■ **Figure 12** Helper functions.

$\mathbf{fields}(t) = \{f_1 : t_1, \dots, f_n : t_n\}$  *(Looking up all fields from a type)*

$$\begin{array}{c} \text{F-BOXED} \\ \hline \mathbf{fields}(B(K)) = \emptyset \end{array} \quad \begin{array}{c} \text{F-TRAIT} \\ \mathbf{trait } T\langle K \rangle \{ \mathbf{require } Fd_1, \dots, \mathbf{require } Fd_n \} \in P \\ \hline \mathbf{fields}(kT) = \{Fd_1, \dots, Fd_n\} \end{array} \quad \begin{array}{c} \text{F-I} \\ \hline \mathbf{fields}(k\mathbf{I}) = \emptyset \end{array}$$

$\mathbf{vals}(Rs) = \{f_1 : t_1, \dots, f_n : t_n\}$  *(Extracting immutable fields from a set of required fields)*

$$\begin{array}{c} \text{REQS-E} \\ \hline \mathbf{vals}(\epsilon) = \emptyset \end{array} \quad \begin{array}{c} \text{REQS-VAR} \\ \hline \mathbf{vals}(Rs, \mathbf{require } \mathbf{var } f : t) = \mathbf{vals}(Rs) \end{array}$$

$$\begin{array}{c} \text{REQS-VAL} \\ \hline \mathbf{vals}(Rs, \mathbf{require } \mathbf{val } f : t) = \{f : t\} \cup \mathbf{vals}(Rs) \end{array}$$

$\mathbf{vals}(t) = \{f_1 : t_1, \dots, f_n : t_n\}$  *(Looking up immutable fields from a type)*

$$\begin{array}{c} \text{V-BOXED} \\ \hline \mathbf{vals}(B(K)) = \emptyset \end{array} \quad \begin{array}{c} \text{V-COMPOSITE} \\ \hline \mathbf{vals}(K_1 \odot K_2) = \mathbf{vals}(K_1) \cup \mathbf{vals}(K_2) \end{array} \quad \begin{array}{c} \text{V-TRAIT} \\ \mathbf{trait } T\langle K \rangle \{ Rs\_ \} \in P \\ \hline \mathbf{vals}(kT) = \mathbf{vals}(Rs) \end{array}$$

$$\begin{array}{c} \text{V-TRAIT-MFST} \\ k \mathbf{trait } T\langle K \rangle \{ Rs\_ \} \in P \\ \hline \mathbf{vals}(kT) = \mathbf{vals}(Rs) \end{array} \quad \begin{array}{c} \text{V-I} \\ \hline \mathbf{vals}(k\mathbf{I}) = \emptyset \end{array}$$

$\mathbf{vars}(t) = \{f_1 : t_1, \dots, f_n : t_n\}$  *(Looking up mutable fields from a type)*

$$\begin{array}{c} \text{V-VARS} \\ \hline \mathbf{vars}(t) = (\mathbf{fields}(t) \setminus \mathbf{vals}(t)) \end{array}$$

■ **Figure 13** Helper functions, continued

## A Code for the Tree in §5.2

The example in §5.2 focused on reading and operating on the elements of the tree, while completely ignoring how the tree was built. The complete definition of the `Tree` class needs an `Add` capability with write access to all parts of the tree to be included in disjunction with the other capabilities.

The syntax of the code is a slightly modified (to look more like the language presented in this paper) version of the syntax of the `Encore` programming language ([www.upscale-project.eu](http://www.upscale-project.eu)) in which the  $\mathcal{K}$  capabilities are currently being implemented.

For simplicity, we use an `int` key to order the elements.

```
class Tree<T> =
  linear Add<T> ⊕ read Visit<T> ⊕
  linear Left<T> ⊗ linear Right<T> ⊗ linear Element<T> {

  var left:Tree<T>;
  var right:Tree<T>;
  var k:int;
  var v:T;
}

trait Visit<T> {
  require val left:Visit<T>;
  require val right:Visit<T>;
  require val k:int;
  require val v:T;

  def preorder(f:S(T) → void) : void {
    if this.left != null
      then this.left.preorder(f);

    f(this.v);

    if this.right != null
      then this.right.preorder(f);
  }

  def get(k:int) : S(T) {
    if k == this.k
      then this.v
    else if k < this.k and this.left != null
      then this.left.get(k)
      else if this.right != null
        then this.right.get(k)
        else null
  }
}
```

The `Add` trait adds support for inserting elements in the tree.

```
trait Add<T> {
  require var left:Tree<T>;
  require var right:Tree<T>;
  require var k:int;
  require var v:T;

  def add(k:int, v:T) : void {
```

```

    if k == this.k
    then this.v = v // or 'consume v' if T is linear (and similar below)
    else if k < this.k
        then this.addLeft(k, v)
        else this.addRight(k, v)
}

def addLeft(k:int, v:T) : void {
    if this.left == null
    then { this.left = new Tree<T>; this.left.set(k, v) }
    else { this.left.add(k, v) }
}

def addRight(k:int, v:T) : void {
    if this.right == null
    then { this.right = new Tree<T>; this.right.set(k, v) }
    else { this.right.add(k, v) }
}
}

```

The code for Left, Right and Element is trivial. Reverse borrowing allows non-destructively reading the left and right fields (as long as the containing object is **linear**) and return their value for its immediate use (but not for storage in a variable). If T is linear, so is Left<T>. Thus,  $y = x.\text{getLeft}()$ ;  $\text{op}(y)$  is not allowed, but  $\text{op}(x.\text{getLeft}())$  is. It does not break linearity since  $x$  is not aliased elsewhere in the system, and since it appears in the argument expression to  $\text{op}()$  it cannot visible inside  $\text{op}()$ . This is more flexible than a field look-up (since  $\text{getLeft}()$  could contain arbitrary computation), and preserves abstraction since the name(s) of the field(s) accessed is hidden.

For clarity, we also show a destructive version of  $\text{getLeft}()$ ,  $\text{getLeft\_destructive}()$ .

```

trait Left<T> {
    require var left:Tree<T>;

    def getLeft() : S(Tree<T>) {
        return this.left;
    }

    def getLeft_destructive() : Tree<T> {
        return consume this.left;
    }

    def setLeft(t:Tree<T>) {
        this.left = consume t;
    }
}

trait Right<T> {
    require var right:Tree<T>;

    def getRight() : S(Tree<T>) {
        return this.right;
    }

    def getRight_destructive() : Tree<T> {
        return consume this.right;
    }
}

```

```

def setRight(t:Tree<T>) {
  this.right = consume t;
}
}

trait Element<T> {
  require var k:int;
  require var v:T;

  def set(k:int, v:T) : void {
    this.k = k;
    this.v = v; // or 'consume v' if T is linear
  }

  def apply(f:T → T) : void {
    this.v = f(this.v);
  }
}

```

## B Sharing, Encapsulation and Linearity

The code below shows a linked list constructed from three separate capabilities: Add, Del and Get. It showcases the interaction between two classes. For simplicity, we ignore some unimportant details, such as checking for **null**-termination.

This linked list design stores elements of type  $P$ , which is omitted. Adding support for parametric polymorphism,  $P$  could be a type parameter. Notably,  $P$  is linear.

```

class List = locked Add ⊕ locked Del ⊕ locked Get {
  var first : Link; // subordinate
}

trait Add {
  require var first : Link;

  def prepend(v : P) : void {
    var tmp : Link = new Link();
    tmp.setNext(this.first);
    tmp.setValue(consume v); // destructive read
    this.first = tmp;
  }
}

trait Del {
  require var first : Link;

  def remove(i : int) : P {
    if i == 0 then{
      var tmp : P = this.first.getValue();
      this.first = this.first.getNext();
      return consume tmp;
    } else {
      return this.first.remove(i);
    }
  }
}

```

```

trait Get {
  require val first : Iterator;

  def lookup(i : int) : P {
    return this.first.get(i);
  }
}

```

Each trait for List class has a corresponding trait for the internal Link class:

```

class Link = subordinate Node  $\oplus$  subordinate Remove  $\oplus$ 
  subordinate Iterator {
  var elmt : P;
  var next : Link;
}

```

```

trait Remove {
  require var next : Link;

  def remove(i : int) : P {
    if i - 1 == 0 then {
      var tmp : P = this.next.getValue();
      this.next = this.next.getNext();
      return consume tmp;
    } else {
      return this.next.remove(i - 1);
    }
  }
}

```

```

trait Node {
  require var elmt : P;
  require var next : Link;

  def setNext(n : Link) : void { this.next = n; }
  def setValue(e : P) : void { this.elmt = consume e; }
  def getNext() : Link { return this.next; }
  def getValue() : P { return consume this.elmt; }
}

```

```

trait Iterator {
  require var elmt : P;
  require val next : Iterator;

  def get(i : int) : P {
    if i == 0
    then return consume this.elmt
    else return this.next.get(i - 1);
  }
}

```

A problem facing any linear type system (most of the related work on substructural types, uniqueness, etc.) is iteration over this structure. Since links are **subordinate**, links may be freely aliased, so already inside the list, there may be multiple paths leading to a supposedly linear value. Thus, to protect its linearity, we force the values to be destructively read. This means that iterating over the list consumes its values (note the **consume** in `Iterator`).

To access the values non-destructively, we could employ reverse borrowing, but then we also need to make `Iterator` **linear**, either by giving it the **linear** mode or by exposing the linear type `P` via nesting: `Iterator<P>`.

The reason we may not reversely borrow from a subordinate capability is that we cannot rule out the existence of multiple paths to a single link, as any two subordinate capabilities may alias (modulo type compatibility). For example, we cannot prove inside `Iterator` that `this.elmt` and `this.next.elmt` are not aliases (this could happen if `this == this.next`). This is the same reasoning that disallows accessing two different subordinates in different **asyncs**.

Making `Iterator` **linear** will in turn will propagate to make `Link` **linear**, which of course has effects for the design of the list, for example precluding turning it into a doubly linked list. A doubly linked list storing linear values can only be expressed using non-linear, subordinate `Links` whose exclusive access is guarded by a dominating (e.g., **locked**) `List` class. Iterating over such a structure without leaving it destroyed is still possible by lending the elements to a higher-order function, and then immediately reinstating them (here shown for the existing `Link` class):

```
trait App {
  require var elmt : P;
  require val next : App; // subordinate
  def app(f : S(P) → void) : void {
    let tmp = consume this.elmt;
    f(tmp); // borrowing
    this.elmt = consume tmp;
    if this.next != null
      then this.next.app(f);
  }
}
```

In an implementation, we could infer that the higher-order function `f` cannot have access to the subordinate state of the `App` trait, and that reversely borrowing `this.f` (i.e., without destructive reads) would therefore be safe.

## Readers–Writer Locks

Removing the linearity on `P` allows turning `Iterator` and therefore also `Get` into **read** capabilities. Composing these capabilities with **locked** versions of the mutating list capabilities would make the list be protected by a readers–writer lock acquiring the read lock on accesses to `Get` and the write-lock on accesses to `Add` or `Del`:

```
class List = locked Add ⊕ locked Del ⊕ read Get

class Link = subordinate Node ⊕ subordinate Remove ⊕
             read Iterator

trait Iterator {
  require val elmt : P; // Note that P is no longer linear
  require val next : read Iterator;

  def get(i : int) : P {
    if i == 0
      then return this.elmt
    else return this.next.get(i - 1);
  }
}
```

Since we have a guarantee that **read** capabilities will not write to the links of the list (Iterator sees next as another **read** Iterator and has only **val** fields) concurrent accesses to Iterator are benign. Note that Iterator *can* call methods on `elmt`, which is also safe since P has its own mode of concurrency control.

## C Initialisation of a Shared Data Structure

Any composition with a linear capability will require the entire composite to be treated linearly. Below we show an example of a property class created from a read-only key value store and a linear initialiser. Instances of Property will thus be local to a single thread, and can even be transferred to another thread to add more data. Finally, the linear part of the data structure can be *discarded* and the resulting type, **read** KeyValueStore is free to be shared, but not updated. A **locked** mode would also have been possible to allow the shared object to be updated.

```
class Property =
  read KeyValueStore ⊕ linear Initialisable {
    var dict:Hashtable;
  }

trait KeyValueStore {
  require val dict:ImmutableHashtable;

  def get(key:String) : String { ... }
  def hasKey(key:String) : boolean { ... }
}

trait Initialisable {
  require var dict:Hashtable;

  def addDataFromDisk(file:String) : void { ... }
  def addDataFromHashtable(t:Hashtable) : void { ... }
}
```

Note that the types of `dict` differs in `KeyValueStore` and `Initialisable`. The composition into `Property` is well-typed using the (C-VAR-VAL) rule that forces the type on the **var** side to be a subtype of the type on the **val** side. Thus, for this class definition to be well-typed, `Hashtable` must be a subtype of `ImmutableHashtable`, *e.g.*, the class `Hashtable` is a composition involving an `ImmutableHashtable`.

## Adding Support for Dropping Capabilities

In the interest of keeping things simple, we omitted support for dropping of a capability in the type system in the main paper. This ability is specific to linear capabilities as it is easy to guarantee that no aliases' view of how the object can be safely accessed is destroyed due to transfer semantics of a single capability (*cf.*, ownership transfer or object-reclassification).

To enable this behaviour, we can simply add a weakening rule to the subtyping rules:

$$\frac{\text{T-SUB-DROP-LINEAR}}{K \odot \text{linear } T <: K}$$

To drop the `Initialisable` part from a property object, simply:

```
var p : Property ... ;
```

```
var r1 : KeyValueStore = (KeyValueStore) consume p;
var r2 : KeyValueStore = r1; // alias
```

Note the support for temporarily dropping Initialisable via the **bound** construct. The following is well-typed by (E-FORWARD) and the definition of the **boundable** helper predicate:

```
var p : Property ... ;
bound p {
  var r2 : KeyValueStore = p; // no consume, so r2 is an alias of p
}
// p is again linear
```

Inside the **bound** block,  $p$  is stack-bound, so no aliases can survive the block.

## D Formal Account of $\mathcal{K}_F$

In this section, we describe  $\mathcal{K}_F$ , the target language to which we translate  $\mathcal{K}$  programs (*cf.*, §7.1).  $\mathcal{K}_F$  is a simple object-oriented language that uses interfaces for subtyping. It has support for structured parallelism and re-entrant readers-writer locks.  $\mathcal{K}_F$  does not know about capabilities, stack-bounding, etc. We make use of an atomic **consume** operation (like a CAS with **null**, although since  $\mathcal{K}$  guarantees freedom from data-races, this does not need to be atomic), and a pointer equality assertion to encode the necessary dynamic machinery for  $\mathcal{K}$ . The translation function from  $\mathcal{K}$  to  $\mathcal{K}_F$  is defined in §G.

Each lock belongs to some object and guards a particular *region* of the object. Each field belongs to a specific region  $r$ , denoted in field declarations as  $f : t$  **in**  $r$ . Disjoint fields from a conjunction  $A \otimes B$  in  $\mathcal{K}$  will end up as fields in different regions in the corresponding class in  $\mathcal{K}_F$ . Locks can be taken as a write lock or a read lock. At most one thread can hold a lock in write mode, while an unbounded number of threads can hold a lock in read mode.

$\mathcal{K}_F$  has been formalised and proven type sound in Coq [15]. The material presented here varies slightly from the mechanised version, but only in ways that improves readability by ignoring uninteresting details like generation of fresh names and similar technicalities. Another difference is the addition of a mechanism for tracing the dynamic types of locations and tainting locations (using the **taint** operator). The tainting is used to track stack-boundedness, and they are both used only in the meta-theoretical reasoning. They have no effect on the execution or type checking of  $\mathcal{K}_F$ -programs, and is thus excluded from the mechanised version.

The syntax of  $\mathcal{K}_F$  is in Figure 14.  $x, y, z, f$  and **this** are variable names.  $r$  range over region names.  $C$  is a class name.  $I$  is an interface name (not to be confused with the “incapability” **I** in  $\mathcal{K}$ ).  $m$  is a name of a method.  $\iota$  is an abstract location in the dynamic semantics of  $\mathcal{K}_F$ .  $l$  is such a location  $\iota$  subscripted with the (static) type of the expression from which this value was obtained. References can be tainted with a superscript  $*$ , *i.e.*, tainted  $\iota$  is written  $\iota^*$ . We use tainting to mark values as stack-bound in the translation from  $\mathcal{K}$  to simplify the meta-theoretical reasoning. Tainting does not affect the execution or type-checking of a  $\mathcal{K}_F$  program—the rules simply propagate taint information and it never gets lost. In all  $\mathcal{K}_F$  rules, wherever  $\iota$  appears,  $\iota^*$  can be used instead.

The expression **wlock**( $x, r$ ) **in**  $e$  locks region  $r$  in the object pointed to by  $x$  for the duration of  $e$  (and similarly for **rlock**). While a locked expression is executed in the dynamic semantics, it appears as **wlocked**<sub>( $\iota, r$ )</sub>{ $e$ } (or **rlocked** respectively). The expression **assert**( $x == y$ );  $e$  asserts that  $x$  and  $y$  are equal (throwing an exception if they are not), and then executes  $e$ .

```

P ::=
    | Cds Ids e

Cd ::=
    | class C implements I { Fds Mds }

Fd ::=
    | f : t in r

Id ::=
    | interface I { Msigs }
    | interface I extends I1, I2

Msig ::=
    | m(x : t1) : t2

Md ::=
    | def m(x : t1) : t2 { e }

e ::=
    | let x, y = e1 in e2
    | x.m(e)
    | x
    | x.f
    | x.f = e
    | new C
    | consume x
    | consume x.f
    | (t)e
    | v
    | finish { async { e1 } async { e2 } }; e3
    | wlock(x, r) in e
    | rlock(x, r) in e
    | wlocked(ι, r){e}
    | rlocked(ι, r){e}
    | assert (x == y); e
    | taint x
    | taint x.f
    | E[e]

l ::=
    | ι
    | ι*

v ::=
    | null
    | lt

t ::=
    | C
    | I
    | Unit

```

■ **Figure 14** Syntax of  $\mathcal{K}_F$ .  $E[e]$  is described in Figure 15

## D.1 Well-Formed Program

A well-formed program consists of well-formed classes and well-formed interfaces plus a well-typed starting expression.

$$\boxed{\vdash P : t \quad \vdash Id \quad \vdash Cd \quad \vdash Fd \quad \vdash Md} \quad (WF \text{ decls.})$$

$$\frac{\text{WF-PROGRAM} \quad \forall Id \in Ids. \vdash Id \quad \forall Cd \in Cds. \vdash Cd \quad \epsilon \vdash e : t}{\vdash Cds \ Ids \ e : t}$$

A non-empty interface is well-formed if its method signatures are well-formed (WF-INTERFACE), and an empty interface is well-formed if the interfaces it extends are well-formed (WF-INTERFACE-EXTENDS).

$$\frac{\text{WF-INTERFACE} \quad \forall m(x : t) : t' \in Msigs. \vdash t \wedge \vdash t'}{\vdash \mathbf{interface} \ I \ \{ Msigs \}}$$

$$\frac{\text{WF-INTERFACE-EXTENDS} \quad \vdash I_1 \quad \vdash I_2}{\vdash \mathbf{interface} \ I \ \mathbf{extends} \ I_1, I_2}$$

By (WF-CLASS), a class is well-formed if it implements all the methods in its interfaces, and all its methods are mentioned in an interface. Further, all fields and methods must be well-formed. We make similar assumptions here as in  $\mathcal{K}$ : we assume that field names are unique in each class and that method names are unique in interfaces.

$$\frac{\text{WF-CLASS} \quad \forall m(x : t) : t' \in \mathbf{msigs}(I). \mathbf{def} \ m(x : t) : t' \ \{ e \} \in Mds \quad \forall \mathbf{def} \ m(x : t) : t' \ \{ e \} \in Mds. m(x : t) : t' \in \mathbf{msigs}(I) \quad \forall Fd \in Fds. \vdash Fd \quad \forall Md \in Mds. \epsilon, \mathbf{this} : C \vdash Md}{\vdash \mathbf{class} \ C \ \mathbf{implements} \ I \ \{ Fds \ Mds \}}$$

A field is well-formed if its type is well-formed. Any region is well-formed (WF-FIELD).

$$\frac{\text{WF-FIELD} \quad \vdash t}{\vdash f : t \ \mathbf{in} \ r}$$

A method is well-formed if its body has the type specified as the method's return type under an environment containing the single parameter and the type of the current this (WF-METHOD).

$$\frac{\text{WF-METHOD} \quad \epsilon, \mathbf{this} : C, x : t \vdash e : t'}{\epsilon, \mathbf{this} : C \vdash \mathbf{def} \ m(x : t) : t' \ \{ e \}}$$

## D.2 Well-Formed Types

Since  $\mathcal{K}_F$  programs only have Java-style interfaces and classes, the rules for well-formedness of types are much simpler than in  $\mathcal{K}$ . Each class in  $\mathcal{K}_F$  corresponds to a class in  $\mathcal{K}$ , and each interface corresponds to a capability or a composite capability. The interface  $I_1$  **extends**  $I_2, I_3$  corresponds to a composition (conjunction or disjunction) between the capabilities corresponding to  $I_2$  and  $I_3$  respectively. The **Unit** interface is an (empty) interface on the top of the interface hierarchy.

$$\boxed{\vdash t} \quad (Well\text{-}formed\ types)$$

$$\begin{array}{c}
 \text{T-WF-CLASS} \\
 \text{class } C \text{ implements } I\{\_\} \in P \\
 \hline
 \vdash C
 \end{array}
 \quad
 \begin{array}{c}
 \text{T-WF-INTERFACE} \\
 \text{interface } I\{\_\} \in P \\
 \hline
 \vdash I
 \end{array}
 \quad
 \begin{array}{c}
 \text{T-WF-INTERFACE-EXTENDS} \\
 \text{interface } I \text{ extends } I_1, I_2 \in P \\
 \hline
 \vdash I
 \end{array}$$

$$\begin{array}{c}
 \text{T-WF-UNIT} \\
 \hline
 \vdash \mathbf{Unit}
 \end{array}$$

## D.3 Subtyping

The structural subtyping from  $\mathcal{K}$  is turned into nominal subtyping by creating a hierarchy of interfaces that matches capability composition. An interface therefore has zero (for example in the case for the root interface **Unit**) or two super interfaces, since composition has two operands. Modulo this peculiarity, the subtyping rules for interfaces in  $\mathcal{K}_F$  are the same as in *e.g.*, Java 1.2. As  $\mathcal{K}$  does not have class inheritance, neither does  $\mathcal{K}_F$ .

$$\boxed{t_1 <: t_2} \quad (Subtyping)$$

$$\begin{array}{c}
 \text{T-SUB-SUB-CLASS} \\
 \text{class } C \text{ implements } I\{\_\} \in P \\
 \hline
 C <: I
 \end{array}
 \quad
 \begin{array}{c}
 \text{T-SUB-SUB-INTERFACE-LEFT} \\
 \text{interface } I \text{ extends } I_1, I_2 \in P \\
 \hline
 I <: I_1
 \end{array}$$

$$\begin{array}{c}
 \text{T-SUB-SUB-INTERFACE-RIGHT} \\
 \text{interface } I \text{ extends } I_1, I_2 \in P \\
 \hline
 I <: I_2
 \end{array}
 \quad
 \begin{array}{c}
 \text{T-SUB-SUB-TRANS} \\
 \frac{t_1 <: t_2 \quad t_2 <: t_3}{t_1 <: t_3}
 \end{array}
 \quad
 \begin{array}{c}
 \text{T-SUB-SUB-EQ} \\
 \frac{\vdash t}{t <: t}
 \end{array}$$

## D.4 Well-Formed Environment

In  $\mathcal{K}_F$ , a well-formed environment  $\Gamma$  has variables of well-formed types and locations of valid class types.

$$\boxed{\vdash \Gamma} \quad (Well\text{-}formed\ environment)$$

$$\begin{array}{c}
 \text{WF-ENV} \\
 \frac{\forall x : t \in \Gamma. \vdash t \quad \forall \iota : C \in \Gamma. \vdash C}{\vdash \Gamma}
 \end{array}$$

## D.5 Environment Subsumption

The relation  $\Gamma_1 \subseteq \Gamma_2$  between environments says that one environment  $\Gamma_2$  subsumes another  $\Gamma_1$  if all facts in  $\Gamma_1$  are also in  $\Gamma_2$ . This relation is crucial in the proof of preservation.

$$\boxed{\Gamma_1 \subseteq \Gamma_2}$$

(*Environment Subsumption*)

$$\frac{\text{WF-SUBSUMPTION} \quad \forall x : t \in \Gamma. \Gamma'(x) = t}{\Gamma \subseteq \Gamma'}$$

## D.6 Frame Rule

The frame rule of  $\mathcal{K}_F$  is similar to the frame rule for  $\mathcal{K}$ , modulo capabilities since they do not exist in  $\mathcal{K}_F$ .

$$\boxed{\Gamma_1 = \Gamma_2 + \Gamma_3}$$

(*Frame Rule*)

$$\frac{\text{WF-FRAME} \quad \begin{array}{l} \forall x. \Gamma_2(x) = t \Rightarrow \Gamma_1(x) = t \\ \forall x. \Gamma_3(x) = t \Rightarrow \Gamma_1(x) = t \\ (\text{dom}(\Gamma_2) \cap \text{dom}(\Gamma_3)) \equiv \emptyset \end{array}}{\Gamma_1 = \Gamma_2 + \Gamma_3}$$

## D.7 Expression Typing

The rules for expression typing in  $\mathcal{K}_F$  are very similar to the corresponding type rules from  $\mathcal{K}$ , except they are simplified since they do not know about capabilities. A **consume** still performs a destructive read, but the type rules do not impose use of **consume** for linear capabilities since no such notion exists, and no special rules for borrowing are needed. Correctness with respect to exclusivity (etc.) comes from the correct translation from the more restrictive  $\mathcal{K}$ .

There are seven new rules in  $\mathcal{K}_F$  that have no correspondence in  $\mathcal{K}$ , namely (WF-LOC), which deals with typing locations inserted into the running program in the dynamic semantics, (WF-ASSERT), which is used to assert equality of two variables (and throw an exception on failure), (WF-TAINT-\*), which is used to mark a location as stack-bound (in practice a no-op), and (WF-LOCK), (WF-RLOCK), (WF-LOCKED), and (WF-RLOCKED) that deal with locks and are straightforward. The last two rules are only used in the dynamic semantics to ensure that locks are not acquired on non-existing objects. In this presentation, we omit checking that regions exist and refer to the mechanised specification of  $\mathcal{K}_F$  for details.

$$\boxed{\Gamma \vdash e : t}$$

(*Typing Expressions*)

$$\frac{\text{WF-CAST} \quad \Gamma \vdash e : t'}{\Gamma \vdash (t)e : t}$$

$$\frac{\text{WF-LET} \quad \Gamma \vdash e_1 : t_1 \quad \Gamma, x : t_1, y : t_1 \vdash e_2 : t}{\Gamma \vdash \text{let } x, y = e_1 \text{ in } e_2 : t}$$

$$\frac{\text{WF-CALL} \quad \Gamma(x) = t_1 \quad \Gamma \vdash e : t_2 \quad \text{msigs}(t_1)(m) = z : t_2 \rightarrow t}{\Gamma \vdash x.m(e) : t}$$

$$\frac{\text{WF-VAR} \quad \vdash \Gamma \quad \Gamma(x) = t}{\Gamma \vdash x : t}$$

$$\begin{array}{c}
\text{WF-LOC} \\
\frac{\Gamma(\iota) = t_2 \quad t_2 <: t}{\Gamma \vdash \iota_t : t_2} \\
\\
\text{WF-NULL} \\
\frac{\vdash \Gamma \quad \vdash t}{\Gamma \vdash \mathbf{null} : t} \\
\\
\text{WF-SELECT} \\
\frac{\Gamma \vdash x : t_1 \quad \mathbf{fields}(t_1)(f) = t_2}{\Gamma \vdash x.f : t_2} \\
\\
\text{WF-UPDATE} \\
\frac{\Gamma \vdash x : t_1 \quad \mathbf{fields}(t_1)(f) = t \quad \Gamma \vdash e : t}{\Gamma \vdash x.f = e : \mathbf{Unit}} \\
\\
\text{WF-NEW} \\
\frac{\vdash \Gamma \quad \mathbf{class } C \mathbf{ implements } I \{ \_ \} \in P}{\Gamma \vdash \mathbf{new } C : C} \\
\\
\text{WF-CONS-VAR} \\
\frac{\vdash \Gamma \quad \Gamma(x) = t}{\Gamma \vdash \mathbf{consume } x : t} \\
\\
\text{WF-CONS-FD} \\
\frac{\Gamma \vdash x : t_1 \quad \mathbf{fields}(t_1)(f) = t}{\Gamma \vdash \mathbf{consume } x.f : t} \\
\\
\text{WF-FJ} \\
\frac{\Gamma = \Gamma_1 + \Gamma_2 \quad \Gamma_1 \vdash e_1 : t_1 \quad \Gamma_2 \vdash e_2 : t_2 \quad \Gamma \vdash e : t}{\Gamma \vdash \mathbf{finish} \{ \mathbf{async} \{ e_1 \} \mathbf{async} \{ e_2 \} \}; e : t} \\
\\
\text{WF-WLOCK} \\
\frac{\Gamma \vdash x : t_2 \quad \Gamma \vdash e : t}{\Gamma \vdash \mathbf{wlock}(x, r) \mathbf{in } e : t} \\
\\
\text{WF-WLOCKED} \\
\frac{\Gamma \vdash e : t \quad \Gamma(\iota) = t_2}{\Gamma \vdash \mathbf{wlocked}_{(\iota, r)} \{ e \} : t} \\
\\
\text{WF-RLOCK} \\
\frac{\Gamma \vdash x : t_2 \quad \Gamma \vdash e : t}{\Gamma \vdash \mathbf{rlock}(x, r) \mathbf{in } e : t} \\
\\
\text{WF-RLOCKED} \\
\frac{\Gamma \vdash e : t \quad \Gamma(\iota) = t_2}{\Gamma \vdash \mathbf{rlocked}_{(\iota, r)} \{ e \} : t} \\
\\
\text{WF-TAINT-VAR} \\
\frac{\Gamma(x) = t}{\Gamma \vdash \mathbf{taint } x : t} \\
\\
\text{WF-TAINT-FIELD} \\
\frac{\Gamma(x) = t_1 \quad \mathbf{fields}(t_1)(f) = t_2}{\Gamma \vdash \mathbf{taint } x.f : t_2} \\
\\
\text{WF-ASSERT} \\
\frac{\Gamma(x) = t_1 \quad \Gamma(y) = t_2 \quad t' <: t_1 \quad t' <: t_2 \quad \Gamma \vdash e : t}{\Gamma \vdash \mathbf{assert}(x == y); e : t}
\end{array}$$

## E The Dynamic Semantics of $\mathcal{K}_F$

Syntax for run-time constructs of  $\mathcal{K}_F$  is found in Figure 15. The dynamic semantics is a small-step reduction semantics that uses evaluation contexts, a well-known technique. A configuration  $\langle H; V; T \rangle$  contains a heap  $H$ , a variable map  $V$ , and a collection of threads  $T$ . A heap  $H$  maps abstract locations to objects. Objects map field names to values and regions to lock-statuses (write-locked by a single thread, or read-locked by  $n$  threads). Each object also carries the identifier of the thread that created it. A stack map  $V$  variable names to values. The thread collection  $T$  is more involved:  $T_1 || T_2 \triangleright e$  denotes two parallel asyncs  $T_1$  and  $T_2$  who must reduce fully before evaluation proceeds to  $e$ .  $(\mathcal{L}, e)_{tid}$  is a set of write-locks acquired by the current thread, its evaluating expression, and a thread identifier. The initial configuration is  $\langle \epsilon; \epsilon; (\emptyset, e) \rangle$ , where  $e$  is the initial expression of the program.

We subscript object locations with types to track the static view a particular location. For example, a successful downcast of  $\iota$  to type  $t$  will result in  $\iota_t$ . This is necessary in to prove data-race freedom as it allows us to track aliasing of disjoint parts of an object. For example,  $\iota_t$  and  $\iota_{t'}$  are aliases to the same object, but their types  $t$  and  $t'$  combine to  $t \otimes t'$  in the original  $\mathcal{K}$  program, meaning that although the locations denote the same object, they cannot be used to access overlapping parts that are not sufficiently protected.

Each reduction is indexed by an effect  $Eff$  which can be empty,  $\mathbf{rd}(\iota.f)$  or  $\mathbf{wr}(\iota.f)$  for reading or writing the field  $f$  of the object in location  $\iota$  respectively. An omitted effect means the empty effect. The effects are used to reason about data-races in §H.

$E[\bullet]$	$::=$		Evaluation ctx.
		<b>let</b> $x + y = \bullet$ <b>in</b> $e$	
		$x.m(\bullet)$	
		$x.f = \bullet$	
		$(t)\bullet$	
		<b>wlocked</b> $_{(\iota,r)}\{\bullet\}$	
		<b>rlocked</b> $_{(\iota,r)}\{\bullet\}$	
$H$	$::=$		Heap
		$\epsilon$	
		$H, \iota \mapsto obj$	
$V$	$::=$		Variables
		$\epsilon$	
		$V, x \mapsto v$	
$T$	$::=$		Threads
		$(\mathcal{L}, e)_{tid}$	
		$T_1 \parallel T_2 \triangleright e$	
		<b>EXN</b>	
$\mathcal{L}$	$::=$		Held locks
		$\emptyset$	
		$L_1, \dots, L_n$	
$L$	$::=$		Held lock
		$(\iota, r)$	
<b>EXN</b>	$::=$		Exceptions
		<b>NullPointerException</b>	
		<b>AssertionException</b>	
		<b>CastException</b>	
$obj$	$::=$		Object
		$(C, F, RL)_{tid}$	
$F$	$::=$		Fields
		$f_1 \mapsto v_1, \dots, f_n \mapsto v_n$	
$RL$	$::=$		Region Locks
		$\epsilon$	
		$RL, r \mapsto LS$	
$LS$	$::=$		Lock status
		<b>locked</b>	
		<b>0 readers</b>	
		<b><math>n</math> readers</b>	
$\Gamma$	$::=$		Environment
		$\epsilon$	
		$\Gamma, x : t$	
		$\Gamma, \iota : C$	
$Eff$	$::=$		Effect
		<b>wr</b> $(\iota.f)$	
		<b>rd</b> $(\iota.f)$	

■ **Figure 15** Syntax for run-time constructs of  $\mathcal{K}_F$ .

By (WF-CFG), a  $\kappa_F$  configuration is well-formed if all its sub-components, heap  $H$  and stack  $V$  are well-formed, its collection of threads  $T$  is well-typed, and the current locking in the system is well-formed.

$$\boxed{\Gamma \vdash cfg : t} \quad (\text{Configuration is well-formed})$$

$$\frac{\text{WF-CFG} \quad \Gamma \vdash H \quad \Gamma \vdash V \quad \Gamma \vdash T : t \quad H \vdash_{\text{lock}} T}{\Gamma \vdash \langle H; V; T \rangle : t}$$

## E.1 Heap, Fields, Vars

By (WF-HEAP), a heap  $H$  is well-formed under a  $\Gamma$  if all locations in  $\Gamma$  correspond to objects in  $H$ , and all the fields of all the objects have the expected static type obtained from the class definition in  $\Gamma$  (via (WF-FIELDS)).

$$\boxed{\Gamma \vdash H \quad \Gamma; C \vdash F \quad \Gamma \vdash V} \quad (\text{WF heap, fields and stack})$$

$$\frac{\text{WF-HEAP} \quad \forall \iota : C \in \Gamma. H(\iota) = (C, F, RL)_{tid} \wedge \Gamma; C \vdash F \quad \forall \iota \in \text{dom}(H). \iota \in \text{dom}(\Gamma) \quad \vdash \Gamma}{\Gamma \vdash H}$$

The rule (WF-FIELDS) captures the well-formedness of the field compartment of an object. A set of fields mapping to values must correspond to the expected fields in the object's type and the values must correspond to the expected types, modulo subtyping.

$$\frac{\text{WF-FIELDS} \quad \mathbf{fields}(C) \equiv f_1 : t_1 \mathbf{in} r_1, \dots, f_n : t_n \mathbf{in} r_n \quad \Gamma \vdash v_1 : t_1, \dots, \Gamma \vdash v_n : t_n}{\Gamma; C \vdash f_1 \mapsto v_1, \dots, f_n \mapsto v_n}$$

A stack  $V$  is well-formed under a  $\Gamma$  if all variables in  $\Gamma$  map to a certain value in  $V$  s.t. the type of the value (looked up from  $\Gamma$ ) corresponds to the type of the variable (also looked up from  $\Gamma$ ).

$$\frac{\text{WF-VARS} \quad \forall x : t \in \Gamma. V(x) = v \wedge \Gamma \vdash v : t \quad \forall x \in \text{dom}(V). x \in \text{dom}(\Gamma) \quad \vdash \Gamma}{\Gamma \vdash V}$$

## E.2 Thread Collection

A configuration's  $T$  compartment holds a collection of concurrently executed threads. Each thread has a thread identifier  $tid$ . A thread can be done, in which case it is fully reduced (or finished with an exception) and will not take another step. The rules are in Figure 16.

$$\boxed{\Gamma; H \vdash T : t} \quad (\text{Well-formed threads})$$

$$\begin{array}{c}
\text{WF-T-ASYNC} \\
\frac{\Gamma \vdash e : t \quad \Gamma \vdash T_1 : t_1 \quad \Gamma \vdash T_2 : t_2}{\Gamma \vdash T_1 \parallel T_2 \triangleright e : t}
\end{array}
\quad
\begin{array}{c}
\text{WF-T-THREAD} \\
\frac{\Gamma \vdash e : t}{\Gamma \vdash (\mathcal{L}, e)_{tid} : t}
\end{array}
\quad
\begin{array}{c}
\text{WF-T-EXN} \\
\frac{\vdash t \quad \vdash \Gamma}{\Gamma \vdash \mathbf{EXN} : t}
\end{array}$$

$$\boxed{\Gamma; H \vdash_{\text{lock}} T} \quad (\text{Well-formed locking})$$

$$\begin{array}{c}
\text{WF-L-ASYNC} \\
\frac{\begin{array}{c}
\mathbf{heldLocks}(T_1) \cap \mathbf{heldLocks}(T_2) \equiv \emptyset \\
\forall (\iota, r) \in \mathbf{wlocks}(e). (\iota, r) \in \mathbf{heldLocks}(T_1) \\
\mathit{distinctWLocks}(e) \quad H \vdash_{\text{lock}} T_1 \quad H \vdash_{\text{lock}} T_2 \\
\forall (\iota, r) \in \mathbf{rlocks}(e). H(\iota) = (C, F, RL)_{tid} \wedge RL(r) = n \mathbf{readers} \wedge n \geq |\mathbf{rlocks}_{(\iota, r)}(e)|
\end{array}}{H \vdash_{\text{lock}} T_1 \parallel T_2 \triangleright e}
\end{array}$$

$$\begin{array}{c}
\text{WF-L-THREAD} \\
\frac{\begin{array}{c}
H \vdash \mathcal{L} \quad \mathit{distinctWLocks}(e) \\
\forall (\iota, r) \in \mathbf{wlocks}(e). (\iota, r) \in \mathcal{L} \\
\forall (\iota, r) \in \mathbf{rlocks}(e). H(\iota) = (C, F, RL)_{tid} \wedge RL(r) = n \mathbf{readers} \wedge n \geq |\mathbf{rlocks}_{(\iota, r)}(e)|
\end{array}}{H \vdash_{\text{lock}} (\mathcal{L}, e)_{tid}}
\end{array}$$

■ **Figure 16** Well-formedness of threads and locks.

### E.3 Locks

In  $\mathcal{K}_F$ ,  $\mathcal{L}$  denotes the write locks held by a single thread in the system. This is represented as a set of tuples  $(\iota, r)$  of locations and locked regions and is well-formed given a heap  $H$  if for all  $(\iota, r)$ , the object at the location  $\iota$  in  $H$  is in the locked state for  $r$ .

$$\boxed{\Gamma \vdash \mathcal{L}} \quad (\text{Well-formed locks})$$

$$\begin{array}{c}
\text{WF-LOCKS-EMPTY} \\
\frac{}{H \vdash \emptyset}
\end{array}
\quad
\begin{array}{c}
\text{WF-LOCKS} \\
\frac{\begin{array}{c}
H \vdash \mathcal{L} \quad (\iota, r) \notin \mathcal{L} \\
H(\iota) = (C, F, RL)_{tid} \\
RL(r) = \mathbf{locked}
\end{array}}{H \vdash \mathcal{L}, (\iota, r)}
\end{array}$$

The rules for how threads may lock are in Figure 16. By (WF-L-THREAD), the write-locks taken by the expression of a thread must appear in its set of held locks, and the corresponding lock on the heap must be locked (captured by  $H \vdash \mathcal{L}$ ). The number of read-locks corresponding to a single region  $r$  of some object  $\iota$  taken by the expression of a thread must be at most the same as the number of registered readers of  $r$  in  $\iota$  on the heap. The parallel case propagates these properties, and additionally requires that two parallel threads do not hold the same locks. Any write-locks held in the continuation  $e$  must be held by the first thread of the async. This represents the fact the first thread is the one that will continue execution after the threads join.

## E.4 Concurrency

We use interleaving semantics, *i.e.*, we model concurrency as a non-deterministic choice between what thread (async) to reduce. This is visible in (DYN-EVAL-ASYNC-LEFT) and (DYN-EVAL-ASYNC-RIGHT). Modulo details which are not visible in our model (such as a non-aligned store turned into two writes on adjacent words), this is WLOG in relation to a truly concurrent semantics. In several cases our translation of  $\mathcal{K}$  programs into  $\mathcal{K}_F$  programs expand a single (supposedly atomic) expression into several expressions, as is the case for **pack**. In these cases, all operations are on stack variables meaning that the expanded operation is effectively atomic wrt. other threads.

$$\boxed{cfg_1 \hookrightarrow cfg_2}$$

(Scheduling)

$$\frac{\text{DYN-EVAL-ASYNC-LEFT} \quad \langle H; V; T_1 \rangle \xrightarrow{\text{Eff}} \langle H'; V'; T'_1 \rangle}{\langle H; V; T_1 \parallel T_2 \triangleright e \rangle \xrightarrow{\text{Eff}} \langle H'; V'; T'_1 \parallel T_2 \triangleright e \rangle}$$

$$\frac{\text{DYN-EVAL-ASYNC-RIGHT} \quad \langle H; V; T_2 \rangle \xrightarrow{\text{Eff}} \langle H'; V'; T'_2 \rangle}{\langle H; V; T_1 \parallel T_2 \triangleright e \rangle \xrightarrow{\text{Eff}} \langle H'; V'; T_1 \parallel T'_2 \triangleright e \rangle}$$

Finish/async spawns one new thread for the second async and uses the current thread for the first. This means that the first async holds all the locks and access to subordinate capabilities (*cf.*, the frame rule of  $\mathcal{K}$  in §6).

$$\frac{\text{DYN-EVAL-SPAWN} \quad e = \mathbf{finish} \{ \mathbf{async} \{ e_1 \} \mathbf{async} \{ e_2 \} \}; e_3 \quad \mathit{tid}_2 \mathbf{fresh}}{\langle H; V; (\mathcal{L}, e)_{\mathit{tid}_1} \rangle \hookrightarrow \langle H; V; (\mathcal{L}, e_1)_{\mathit{tid}_1} \parallel (\emptyset, e_2)_{\mathit{tid}_2} \triangleright e_3 \rangle}$$

When asyncs have finished (a join), the second thread is removed along with all its locks, which have already been released (DYN-EVAL-ASYNC-JOIN).

$$\frac{\text{DYN-EVAL-ASYNC-JOIN}}{\langle H; V; (\mathcal{L}, v)_{\mathit{tid}_1} \parallel (\mathcal{L}', v')_{\mathit{tid}_2} \triangleright e \rangle \hookrightarrow \langle H; V; (\mathcal{L}, e)_{\mathit{tid}_1} \rangle}$$

(DYN-EVAL-SPAWN-CONTEXT) exercises the evaluation context rules forcing the full reduction of the parallel expressions to the left of  $\triangleright$  before continuing with  $e_3$ , which will be the value eventually placed in the hole of the evaluation context.

$$\frac{\text{DYN-EVAL-SPAWN-CONTEXT} \quad \langle H; V; (\mathcal{L}, e)_{\mathit{tid}} \rangle \hookrightarrow \langle H; V; (\mathcal{L}, e_1)_{\mathit{tid}} \parallel (\emptyset, e_2)_{\mathit{tid}_2} \triangleright e_3 \rangle}{\langle H; V; (\mathcal{L}, E[e])_{\mathit{tid}_1} \rangle \hookrightarrow \langle H; V; (\mathcal{L}, e_1)_{\mathit{tid}_1} \parallel (\emptyset, e_2)_{\mathit{tid}_2} \triangleright E[e_3] \rangle}$$

## E.5 Expressions

We use a single stack frame for the entire program and employ renaming to make sure that variables have unique names. We further subscript variable names by *frame indices*,  $n$ , to be able to reconstruct the ordering that an actual stack of frames would give for free, while allowing the stack to grow monotonically. In (DYN-EVAL-CALL), we use the helper function  $\sigma_n(e)$  to subscript all bound and free variables in  $e$  with  $n$  to tie them to a common stack frame. A fresh frame index is always higher than before, so  $x_n$  and  $y_m$  where  $n < m$  means that  $x$  is on a lower frame than  $y$ . This is important to reason about alias burying in the context of borrowing and is employed in the proof of thread-safety. Where a variable's frame index is irrelevant, it has been omitted.

The evaluation context  $E$  decides the order of evaluation.

$$\boxed{cfg_1 \hookrightarrow cfg_2} \quad (\text{Expressions})$$

$$\begin{array}{c} \text{DYN-EVAL-CONTEXT} \\ \frac{\langle H; V; (\mathcal{L}, e)_{tid} \rangle \xrightarrow{E\text{ff}} \langle H'; V'; (\mathcal{L}', e')_{tid} \rangle}{\langle H; V; (\mathcal{L}, E[e]_{tid}) \rangle \xrightarrow{E\text{ff}} \langle H'; V'; (\mathcal{L}', E[e']_{tid}) \rangle} \end{array}$$

Casts are handled dynamically in  $\kappa_F$  and can be both upcasts and downcasts. Failed casts are handled in §E.6.

$$\begin{array}{c} \text{DYN-EVAL-UPCAST} \\ \frac{t_2 <: t}{\langle H; V; (\mathcal{L}, (t)\iota_{t_2})_{tid} \rangle \hookrightarrow \langle H; V; (\mathcal{L}, \iota_t)_{tid} \rangle} \end{array}$$

$$\begin{array}{c} \text{DYN-EVAL-DOWNCAST} \\ \frac{H(t) = (C, F, RL)_{tid} \quad C <: t}{\langle H; V; (\mathcal{L}, (t)\iota_{t_2})_{tid} \rangle \hookrightarrow \langle H; V; (\mathcal{L}, \iota_t)_{tid} \rangle} \end{array}$$

$$\begin{array}{c} \text{DYN-EVAL-CAST-NULL} \\ \frac{}{\langle H; V; (\mathcal{L}, (t)\mathbf{null})_{tid} \rangle \hookrightarrow \langle H; V; (\mathcal{L}, \mathbf{null})_{tid} \rangle} \end{array}$$

To support unpacking (which creates two aliases from a common source) we use a **let** expression that supports multiple assignment. It adds two variables with freshly picked names to the frame using the same frame index as the variables they are replacing. Then the old variables are substituted for the new. For simplicity the same rule is used for normal variable introduction as well.

$$\begin{array}{c} \text{DYN-EVAL-LET} \\ \frac{x' \text{ fresh} \quad y' \text{ fresh} \quad V' = V[y'_n \mapsto v][x'_n \mapsto v] \quad e' = e[y \mapsto y'][x \mapsto x']}{\langle H; V; (\mathcal{L}, \mathbf{let } x_n, y_n = v \mathbf{ in } e)_{tid} \rangle \hookrightarrow \langle H; V'; (\mathcal{L}, e')_{tid} \rangle} \end{array}$$

Frame indices grow on method calls, which pick a fresh index for the new logical frame. The freshly picked frame index is always higher than any index picked before. A single “counter” is shared between all threads as we are only interested in knowing if an index is smaller, not how much smaller. As  $n$  is picked fresh, **this** <sub>$n$</sub>  is a fresh variable name. Note that the  $\sigma$  substitution changes all occurrences of **this** into **this** <sub>$n$</sub>  in  $e$ .

$$\begin{array}{c}
\text{DYN-EVAL-CALL} \\
V(x) = \iota_t \quad H(\iota) = (C, F, RL)_{tid} \\
\text{methods}(C)(m) = y : t_1 \rightarrow t_2, e \quad n \text{ fresh} \\
V' = V[\text{this}_n \mapsto \iota_t][y_n \mapsto v] \quad e' = \sigma_n(e) \\
\hline
\langle H; V; (\mathcal{L}, x.m(v))_{tid} \rangle \hookrightarrow \langle H; V'; (\mathcal{L}, e')_{tid} \rangle
\end{array}$$

Taking a write lock requires that there are no active write or read locks, indicated by 0 **readers**, and adds the locked object and region to the  $\mathcal{L}$  set of objects locked by the current thread. It also updates the object to reflect its locked status. The **wlocked** wrapper around  $e$  records the successful taking of the lock and is used to release the lock once  $e$  has been fully reduced.

$$\begin{array}{c}
\text{DYN-EVAL-WLOCK} \\
V(x) = \iota_t \quad H(\iota) = (C, F, RL)_{tid} \\
RL(r) = 0 \text{ readers} \quad (\iota, r) \notin \mathcal{L} \\
H' = H[\iota \mapsto (C, F, RL[r \mapsto \text{locked}])] \\
\mathcal{L}' \equiv \mathcal{L}, (\iota, r) \\
\hline
\langle H; V; (\mathcal{L}, \text{wlock}(x, r) \text{ in } e)_{tid} \rangle \hookrightarrow \langle H'; V; (\mathcal{L}', \text{wlocked}_{(\iota, r)}\{e\})_{tid} \rangle
\end{array}$$

Locks are reentrant, meaning that an attempt by a thread to acquire a lock already in its locked set will succeed.

$$\begin{array}{c}
\text{DYN-EVAL-WLOCK-REENTRANT} \\
V(x) = \iota_t \quad H(\iota) = (C, F, RL)_{tid} \\
RL(r) = \text{locked} \quad (\iota, r) \in \mathcal{L} \\
\hline
\langle H; V; (\mathcal{L}, \text{wlock}(x, r) \text{ in } e)_{tid} \rangle \hookrightarrow \langle H; V; (\mathcal{L}, e)_{tid} \rangle
\end{array}$$

Once an expression wrapped in a **wlocked** wrapper is fully reduced, the lock is released and the wrapper removed.

$$\begin{array}{c}
\text{DYN-EVAL-WLOCK-RELEASE} \\
H(\iota) = (C, F, RL)_{tid} \quad RL(r) = \text{locked} \\
(\iota, r) \in \mathcal{L} \\
H' = H[\iota \mapsto (C, F, RL[r \mapsto 0 \text{ readers}])] \\
\mathcal{L}' \equiv \mathcal{L} \setminus \{(\iota, r)\} \\
\hline
\langle H; V; (\mathcal{L}, \text{wlocked}_{(\iota, r)}\{v\})_{tid} \rangle \hookrightarrow \langle H'; V; (\mathcal{L}', v)_{tid} \rangle
\end{array}$$

The semantics of read locks is similar to write locks, but read locks are not exclusive so there may be multiple threads holding a read lock simultaneously. A read lock can be successfully taken if there is no active write lock.

$$\begin{array}{c}
\text{DYN-EVAL-RLOCK} \\
V(x) = \iota_t \quad H(\iota) = (C, F, RL)_{tid} \\
RL(r) = n \text{ readers} \\
H' = H[\iota \mapsto (C, F, RL[r \mapsto (n + 1) \text{ readers}])] \\
\hline
\langle H; V; (\mathcal{L}, \text{rlock}(x, r) \text{ in } e)_{tid} \rangle \hookrightarrow \langle H'; V; (\mathcal{L}, \text{rlocked}_{(\iota, r)}\{e\})_{tid} \rangle
\end{array}$$

Multiple acquires by the same thread bumps the readers counter. A thread that holds a write lock will always succeed in taking the corresponding read lock.

$$\begin{array}{c}
\text{DYN-EVAL-RLOCK-REENTRANT} \\
V(x) = \iota_t \quad H(\iota) = (C, F, RL)_{tid} \\
RL(r) = \mathbf{locked} \quad (\iota, r) \in \mathcal{L} \\
\hline
\langle H; V; (\mathcal{L}, \mathbf{rlock}(x, r) \mathbf{in} e)_{tid} \rangle \leftrightarrow \langle H'; V; (\mathcal{L}, e)_{tid} \rangle
\end{array}$$

Read locks are released like write locks.

$$\begin{array}{c}
\text{DYN-EVAL-RLOCK-RELEASE} \\
H(\iota) = (C, F, RL)_{tid} \quad RL(r) = n \mathbf{readers} \\
H' = H[\iota \mapsto (C, F, RL[r \mapsto (n-1) \mathbf{readers}])] \\
\hline
\langle H; V; (\mathcal{L}, \mathbf{rlocked}_{(\iota, r)}\{v\})_{tid} \rangle \leftrightarrow \langle H'; V; (\mathcal{L}', v)_{tid} \rangle
\end{array}$$

We have omitted constructors from this treatise. A new object has its fields initialised to **null**, has no locks taken, and is given a fresh abstract location on the heap.

$$\begin{array}{c}
\text{DYN-EVAL-NEW} \\
\mathbf{fields}(C) \equiv f_1 : t_1 \mathbf{in} r_1, \dots, f_n : t_n \mathbf{in} r_n \\
\forall r \in r_1 \dots r_n. RL(r) = 0 \mathbf{readers} \\
F \equiv f_1 \mapsto \mathbf{null}, \dots, f_n \mapsto \mathbf{null} \\
\iota \mathbf{fresh} \quad H' = H[\iota \mapsto (C, F, RL)_{tid}] \\
\hline
\langle H; V; (\mathcal{L}, \mathbf{new} C)_{tid} \rangle \leftrightarrow \langle H'; V; (\mathcal{L}, \iota_C)_{tid} \rangle
\end{array}$$

Variable look-up, field look-up and assignment, and the destructive read versions are straightforward.

$$\begin{array}{c}
\text{DYN-EVAL-VAR} \\
V(x) = v \\
\hline
\langle H; V; (\mathcal{L}, x)_{tid} \rangle \leftrightarrow \langle H; V; (\mathcal{L}, v)_{tid} \rangle
\end{array}$$

$$\begin{array}{c}
\text{DYN-EVAL-SELECT} \\
V(x) = \iota_t \quad H(\iota) = (C, F, RL)_{tid} \\
\mathbf{fields}(C)(f) = t \mathbf{in} r \quad F(f) = v \\
\hline
\langle H; V; (\mathcal{L}, x.f)_{tid} \rangle \xrightarrow{\mathbf{rd}^{(\iota, f)}} \langle H; V; (\mathcal{L}, v)_{tid} \rangle
\end{array}$$

$$\begin{array}{c}
\text{DYN-EVAL-UPDATE} \\
V(x) = \iota_t \quad H(\iota) = (C, F, RL)_{tid} \\
\mathbf{fields}(C)(f) = t \mathbf{in} r \quad H' = H[\iota \mapsto (C, F[f \mapsto v], RL)] \\
\hline
\langle H; V; (\mathcal{L}, x.f = v)_{tid} \rangle \xrightarrow{\mathbf{wr}^{(\iota, f)}} \langle H'; V; (\mathcal{L}, \mathbf{null})_{tid} \rangle
\end{array}$$

$$\begin{array}{c}
\text{DYN-EVAL-CONSUME-VAR} \\
V(x) = v \quad V' = V[x \mapsto \mathbf{null}] \\
\hline
\langle H; V; (\mathcal{L}, \mathbf{consume} x)_{tid} \rangle \leftrightarrow \langle H; V'; (\mathcal{L}, v)_{tid} \rangle
\end{array}$$

$$\begin{array}{c}
\text{DYN-CONSUME-FIELD} \\
V(x) = \iota_t \quad H(\iota) = (C, F, RL)_{tid} \\
\mathbf{fields}(C)(f) = t \mathbf{in} r \\
H' = H[\iota \mapsto (C, F[f \mapsto \mathbf{null}], RL)] \quad F(f) = v \\
\hline
\langle H; V; (\mathcal{L}, \mathbf{consume} x.f)_{tid} \rangle \xrightarrow{\mathbf{wr}^{(\iota, f)}} \langle H'; V; (\mathcal{L}, v)_{tid} \rangle
\end{array}$$

For meta-theoretical reasoning we use a taint function to mark a location as stack-bound. This function is only used when translating from  $\mathcal{K}$  to  $\mathcal{K}_F$ , and is in practice just like reading a variable or a field.

$$\frac{\text{DYN-EVAL-TAINT-VAR} \quad V(x) = \iota_t}{\langle H; V; (\mathcal{L}, \mathbf{taint} \ x)_{tid} \rangle \hookrightarrow \langle H; V; (\mathcal{L}, \iota_t^*)_{tid} \rangle}$$

$$\frac{\text{DYN-EVAL-TAINT-VAR-NULL} \quad V(x) = \mathbf{null}}{\langle H; V; (\mathcal{L}, \mathbf{taint} \ x)_{tid} \rangle \hookrightarrow \langle H; V; (\mathcal{L}, \mathbf{null})_{tid} \rangle}$$

$$\frac{\text{DYN-EVAL-TAINT-FIELD} \quad \begin{array}{l} V(x) = \iota_t' \quad H(\iota') = (C, F, RL)_{tid} \\ \mathbf{fields}(C)(f) = t \ \mathbf{in} \ r \quad F(f) = \iota_t \end{array}}{\langle H; V; (\mathcal{L}, \mathbf{taint} \ x.f)_{tid} \rangle \hookrightarrow \langle H; V; (\mathcal{L}, \iota_t^*)_{tid} \rangle}$$

$$\frac{\text{DYN-EVAL-TAINT-FIELD-NULL} \quad \begin{array}{l} V(x) = \iota_t \quad H(\iota) = (C, F, RL)_{tid} \\ \mathbf{fields}(C)(f) = t \ \mathbf{in} \ r \quad F(f) = \mathbf{null} \end{array}}{\langle H; V; (\mathcal{L}, \mathbf{taint} \ x.f)_{tid} \rangle \hookrightarrow \langle H; V; (\mathcal{L}, \mathbf{null})_{tid} \rangle}$$

Finally, to check that the two variables in a  $\mathcal{K}$ -level pack are aliases, an exception throwing assert statement is used.

$$\frac{\text{DYN-EVAL-ASSERT} \quad V(x) = \iota_{t_1} \quad V(y) = \iota_{t_2}}{\langle H; V; (\mathcal{L}, \mathbf{assert} \ (x == y); e)_{tid} \rangle \hookrightarrow \langle H; V; (\mathcal{L}, e)_{tid} \rangle}$$

## E.6 Exceptions

The exception rules are straightforward and exceptions terminate the entire program. The only point that warrants clarification is (DYN-EXCEPTION-CONTEXT) which abstracts the nature of an underlying exception to avoid rule duplication.

For readability we abbreviate **NullPointerException** as **NPE**. When we don't care about the kind of exception we write **EXN**.

$$\boxed{cfg_1 \hookrightarrow cfg_2}$$

(Exceptions)

$$\frac{\text{DYN-NPE-SELECT} \quad V(x) = \mathbf{null}}{\langle H; V; (\mathcal{L}, x.f)_{tid} \rangle \hookrightarrow \langle H; V; \mathbf{NPE} \rangle}$$

$$\frac{\text{DYN-NPE-UPDATE} \quad V(x) = \mathbf{null}}{\langle H; V; (\mathcal{L}, x.f = v)_{tid} \rangle \hookrightarrow \langle H; V; \mathbf{NPE} \rangle}$$

$$\begin{array}{c}
\text{DYN-NPE-CONSUME-FIELD} \\
\frac{V(x) = \mathbf{null}}{\langle H; V; (\mathcal{L}, \mathbf{consume } x.f)_{tid} \rangle \leftrightarrow \langle H; V; \mathbf{NPE} \rangle} \\
\text{DYN-NPE-CALL} \\
\frac{V(x) = \mathbf{null}}{\langle H; V; (\mathcal{L}, x.m(v))_{tid} \rangle \leftrightarrow \langle H; V; \mathbf{NPE} \rangle} \\
\text{DYN-NPE-WLOCK} \\
\frac{V(x) = \mathbf{null}}{\langle H; V; (\mathcal{L}, \mathbf{wlock}(x, r) \mathbf{in } e)_{tid} \rangle \leftrightarrow \langle H; V; \mathbf{NPE} \rangle} \\
\text{DYN-NPE-RLOCK} \\
\frac{V(x) = \mathbf{null}}{\langle H; V; (\mathcal{L}, \mathbf{rlock}(x, r) \mathbf{in } e)_{tid} \rangle \leftrightarrow \langle H; V; \mathbf{NPE} \rangle} \\
\text{DYN-NPE-TAINT} \\
\frac{V(x) = \mathbf{null}}{\langle H; V; (\mathcal{L}, \mathbf{taint } x.f)_{tid} \rangle \leftrightarrow \langle H; V; \mathbf{NPE} \rangle} \\
\text{DYN-EXN-ASSERT} \\
\frac{V(x) = \mathbf{null} \vee V(y) = \mathbf{null} \vee V(x) \neq V(y)}{\langle H; V; (\mathcal{L}, \mathbf{assert } (x == y); e)_{tid} \rangle \leftrightarrow \langle H; V; \mathbf{AssertionException} \rangle} \\
\text{DYN-EXN-CAST} \\
\frac{H(t) = (C, F, RL)_{tid} \quad \neg C <: t}{\langle H; V; (\mathcal{L}, (t)\iota_t)_{tid} \rangle \leftrightarrow \langle H; V; \mathbf{CastException} \rangle} \\
\text{DYN-EXCEPTION-CONTEXT} \\
\frac{\langle H; V; (\mathcal{L}, e)_{tid} \rangle \leftrightarrow \langle H'; V'; \mathbf{EXN} \rangle}{\langle H; V; (\mathcal{L}, E[e])_{tid} \rangle \leftrightarrow \langle H'; V'; \mathbf{EXN} \rangle} \\
\text{DYN-EXCEPTION-ASYNC-LEFT} \\
\frac{}{\langle H; V; \mathbf{EXN} \parallel T_2 \triangleright e \rangle \leftrightarrow \langle H; V; \mathbf{EXN} \rangle} \\
\text{DYN-EXCEPTION-ASYNC-RIGHT} \\
\frac{}{\langle H; V; T_1 \parallel \mathbf{EXN} \triangleright e \rangle \leftrightarrow \langle H; V; \mathbf{EXN} \rangle}
\end{array}$$

## E.7 Blocking

The blocking property of a configuration holds all its threads are either blocking on a lock or are done. This property is necessary to distinguish deadlocks from stuck states.

$\boxed{\text{Blocked}(cfg)}$

*(Configuration is blocked)*

**BLOCKED-W-BLOCKED**

$$\frac{V(x) = \iota_t \quad H(t) = (C, F, RL)_{tid} \quad (\iota, r) \notin \mathcal{L} \quad RL(r) = \mathbf{locked}}{\text{Blocked}(\langle H; V; (\mathcal{L}, \mathbf{wlock}(x, r) \mathbf{in } e)_{tid} \rangle)}$$

**BLOCKED-W-RBLOCKED**

$$\frac{V(x) = \iota_t \quad H(t) = (C, F, RL)_{tid} \quad RL(r) = n \mathbf{readers} \quad n > 0}{\text{Blocked}(\langle H; V; (\mathcal{L}, \mathbf{wlock}(x, r) \mathbf{in } e)_{tid} \rangle)}$$

**BLOCKED-RBLOCKED**

$$\frac{V(x) = \iota_t \quad H(t) = (C, F, RL)_{tid} \quad (\iota, r) \notin \mathcal{L} \quad RL(r) = \mathbf{locked}}{\text{Blocked}(\langle H; V; (\mathcal{L}, \mathbf{rlock}(x, r) \mathbf{in } e)_{tid} \rangle)}$$

**BLOCKED-BLOCKED-CONTEXT**

$$\frac{\text{Blocked}(\langle H; V; (\mathcal{L}, e)_{tid} \rangle)}{\text{Blocked}(\langle H; V; (\mathcal{L}, E[e])_{tid} \rangle)}$$

$$\begin{array}{c}
\text{BLOCKED-DEADLOCK} \\
\frac{\text{Blocked}(\langle H; V; T_1 \rangle) \quad \text{Blocked}(\langle H; V; T_2 \rangle)}{\text{Blocked}(\langle H; V; T_1 \parallel T_2 \triangleright e \rangle)} \\
\\
\text{BLOCKED-BLOCKED-LEFT} \\
\frac{\text{Blocked}(\langle H; V; T_1 \rangle)}{\text{Blocked}(\langle H; V; T_1 \parallel (\mathcal{L}, v)_{tid} \triangleright e \rangle)} \\
\\
\text{BLOCKED-BLOCKED-RIGHT} \\
\frac{\text{Blocked}(\langle H; V; T_2 \rangle)}{\text{Blocked}(\langle H; V; (\mathcal{L}, v)_{tid} \parallel T_2 \triangleright e \rangle)}
\end{array}$$

isDone( $T$ )

*(Thread has finished)*

DONE

$$\frac{}{\text{isDone}((\mathcal{L}, v)_{tid})}$$

## F

 Type Soundness of  $\kappa_F$ 

Here we state the type soundness theorems of  $\kappa_F$ . The formal Coq proofs are available online [15].

► **Progress.** A well-formed configuration is either done, has thrown an exception, has dead-locked, or can take one additional step:

$$\forall \Gamma, H, V, T, t. \Gamma \vdash \langle H; V; T \rangle : t \Rightarrow \text{isDone}(T) \vee T = \mathbf{EXN} \vee \text{Blocked}(\langle H; V; T \rangle) \vee \exists cf g', \langle H; V; T \rangle \hookrightarrow cf g'$$

► **Preservation.** If  $\langle H; V; T \rangle$  types to  $t$  under some typing environment  $\Gamma$ , and  $\langle H; V; T \rangle$  can step to some  $\langle H'; V'; T' \rangle$ , there exists a superset of  $\Gamma$  that types  $\langle H'; V'; T' \rangle$  to  $t$ .

$$\begin{array}{l}
\forall \Gamma, H, H', V, V', T, T', t. \\
\Gamma \vdash \langle H; V; T \rangle : t \wedge \langle H; V; T \rangle \hookrightarrow \langle H'; V'; T' \rangle \Rightarrow \\
\exists \Gamma'. \Gamma' \vdash \langle H'; V'; T' \rangle : t \wedge \Gamma \subseteq \Gamma'
\end{array}$$

Both theorems are proven by induction on the thread structure  $T$ . The single-threaded case is proven by induction over the typing relation on the current expression  $e$ . For the proof of preservation, there is a number of lemmas regarding locking that needs proving (*e.g.*, that a thread cannot steal a lock held by another thread). We refer to the Coq proofs for details.

## G

 Translation from  $\kappa$  to  $\kappa_F$ 

For each syntactic construct in  $\kappa$ , we define a translation function  $\mathcal{F}$  that takes that construct to constructs in  $\kappa_F$ .

### G.1 Programs

A  $\kappa$  program is translated by translating the list of classes and traits, as well as the starting expression:

$$\mathcal{F}(Cds Tds e) = \mathcal{F}(Cds) \mathcal{F}(Tds) \mathcal{F}(e)$$

## G.2 Classes and Regions

A  $\mathcal{K}$  class is translated to a  $\mathcal{K}_F$  class that implements an interface corresponding to the trait composition in the original class. Nesting parameters are discarded. From the trait composition we calculate a map  $\theta$  from field names to regions such that all traits that share the same fields map to the same region. For example, `Add`, `Del` and `Get` from our example in §B would all map to the same region because they all share a common field `first`. This means that if these traits used locks, they would all try to acquire the same lock at the start of each method (`Get` in read mode, and the others in write mode). If `Get` did not use a lock, it would be able to witness changes, *i.e.*, a read–write race.

Field declarations are translated and assigned regions using the auxiliary function  $\mathcal{F}_\theta$ . Similarly we calculate a map  $\Lambda$  that maps method names to locking instructions, *e.g.*, all methods defined in `Add` map to the region of `Add`. The methods are extracted from the traits and translated using  $\mathcal{F}_\Lambda$ , which wraps their bodies in locks if the method name is in the map.

For tracking reasons we extend each class with a field `dominator` pointing to the dominating object (which is itself, unless the object is a subordinate in  $\mathcal{K}$ ). This field is never used in the program and cannot be dereferenced as it has the top-type **Unit**. Therefore it also does not matter which region the field is in.

$$\begin{aligned} \mathcal{F}(\text{class } C = K \{Fds\}) = & \\ & \text{class } C \text{ implements } I_K \{ \\ & \quad \text{dominator} : \mathbf{Unit} \text{ in } r_0 \\ & \quad \mathcal{F}_\theta(Fds) \\ & \quad \mathcal{F}_\Lambda(Mds) \\ & \quad \} \\ & \text{where } Mds = \mathbf{methods}(K) \end{aligned}$$

It should be noted that  $\theta$  and  $\Lambda$  is sometimes over-cautious in how they assign regions. If  $A$  and  $B$  share fields, and  $B$  and  $C$  share fields, these traits have all their fields put in the same region, even if  $A$  and  $C$  could be operated on in parallel. This simplifies the formalism and prevents some deadlocks that could arise if  $B$  needed to take two locks before running a method. In an implementation, a more clever protocol for taking locks that avoids such deadlocks could be used, allowing more parallelism. We ignore this in the context of the formalism.

## G.3 Traits

A trait declaration is translated to a  $\mathcal{K}_F$  interface containing the same method signatures. Nesting parameters and manifest modes are discarded.

$$\begin{aligned} \mathcal{F}(\_ \text{trait } T \langle \_ \rangle \{ \_ Mds \}) = & \text{interface } I_T \{ Msigs \} \\ \text{where } Msigs_i = m(x : t) : t' & \\ \text{iff } Mds_i = \mathbf{def } m(x : t) : t' \{ \_ \} & \end{aligned}$$

For each possible flat combination of capabilities from a list of traits we generate an interface using the pattern  $K_1 \odot K_2 \Rightarrow \text{interface } I_{K_1 \odot K_2} \text{ extends } I_{K_1}, I_{K_2}$ .

## G.4 Types

Class types are translated as is. Capability atoms and flat compositions are translated to the corresponding interface type. Modes and nested types are discarded. Jails are translated

into the (empty) root interface **Unit**.

$$\mathcal{F}(t) = \begin{cases} C & \text{if } t = C \\ I_T & \text{if } t = k T \\ \mathcal{F}(K_1) & \text{if } t = K_1 \langle K_2 \rangle \\ \mathbf{Unit} & \text{if } t = \mathbf{J}_{K'}(K) \\ I_{K_1 \odot K_2} & \text{if } t = K_1 \odot K_2 \end{cases}$$

## G.5 Fields

Fields are translated by translating the field type. Access modifiers are discarded. A field gets its region from the region map that was calculated when translating the class:

$$\mathcal{F}_\theta(\text{mod } f : t) = f : t \text{ in } \theta(f)$$

## G.6 Methods

Methods are translated by translating the types and method body. If the method name appears in the region map  $\Lambda$ , the method body is wrapped in the corresponding lock.

$$\begin{aligned} \mathcal{F}_\Lambda(\text{def } m(x : t) : t' \{e\}) = & \\ & \text{def } m(x : \mathcal{F}(t)) : \mathcal{F}(t') \{ \mathbf{wlock}(\mathbf{this}, r) \text{ in } \mathcal{F}(e) \} \\ & \text{if } \Lambda(m) = \mathbf{wlock } r \\ & \text{def } m(x : \mathcal{F}(t)) : \mathcal{F}(t') \{ \mathbf{rlock}(\mathbf{this}, r) \text{ in } \mathcal{F}(e) \} \\ & \text{if } \Lambda(m) = \mathbf{rlock } r \\ & \text{def } m(x : \mathcal{F}(t)) : \mathcal{F}(t') \{ \mathcal{F}(e) \} \\ & \text{otherwise} \end{aligned}$$

## G.7 Expressions

Expressions are mostly translated as is. We assume the presence of an environment  $\Gamma$  containing the types of all local variables. Unpack and pack are translated into a **let** expression, the latter together with an assertion that the variables being packed are actually aliases and a downcast to the resulting type. **sync** is translated into taking a lock on the region  $r_0$ , which is the region all unsafe capabilities keep their fields in. Since the **bound** expression is only used to change types in  $\mathcal{K}$ , it is a sequence in  $\mathcal{K}_F$  where the value becoming stack-bound is tainted for tracking reasons. The cast rule needs to have its type translated.

To be able to reason about ownership, we let each object have a field dominator that is filled in on creation time. When creating an object that is not subordinate, the object is its own dominator. When creating a subordinate object, the dominator will be the same as the dominator of the current **this**. Note that such a **this** must always exist in a well-formed  $\mathcal{K}$  program, and therefore in its translation into  $\mathcal{K}_F$  (E-NEW).

The full translation function for expressions is found in Figure 17.

$$\mathcal{F}(e) = \left\{ \begin{array}{l}
(\mathcal{F}(t))\mathcal{F}(e_1) \\
\quad \text{if } e = (t)e_1 \\
\text{let } x, \_ = \mathcal{F}(e_1) \text{ in } \mathcal{F}(e_2) \\
\quad \text{if } e = \text{let } x = e_1 \text{ in } e_2 \\
\text{let } x, y = \text{consume } z \text{ in } \mathcal{F}(e) \\
\quad \text{if } e = \text{unpack } x + y = z \text{ in } e \\
\text{assert}(y==z); \\
\text{let } \_, \_ = \text{consume } y \text{ in} \\
\quad \text{let } x, \_ = (\mathcal{F}(t)) \text{ consume } z \text{ in } \mathcal{F}(e) \\
\quad \text{where } t = \Gamma(y) \otimes \Gamma(z) \\
\quad \text{if } e = \text{pack } x = y + z \text{ in } e \\
\text{let } \_, \_ = \\
\quad \text{let } y, \_ = x \text{ in wlock}(y, r_0) \text{ in } \mathcal{F}(e_1) \\
\quad \text{in } \mathcal{F}(e_2) \\
\quad \text{if } e = \text{sync } x \text{ as } y \{e_1\}; e_2 \\
\text{let } \_, \_ = \\
\quad \text{let } x, \_ = \text{taint } x \text{ in } \mathcal{F}(e_1) \\
\quad \text{in } \mathcal{F}(e_2) \\
\quad \text{if } e = \text{bound } x \{e_1\}; e_2 \\
x.m(\mathcal{F}(e_1)) \\
\quad \text{if } e = x.m(e_1) \\
x.f = \mathcal{F}(e_1) \\
\quad \text{if } e = x.f = e_1 \\
\text{finish} \{ \text{async} \{ \mathcal{F}(e_1) \} \text{ async} \{ \mathcal{F}(e_2) \} ; \mathcal{F}(e_3) \} \\
\quad \text{if } e = \text{finish} \{ \text{async} \{ e_1 \} \text{ async} \{ e_2 \} ; e_3 \} \\
\text{let } x, \_ = \text{new } C \text{ in} \\
\quad \text{let } \_, \_ = x.\text{dominator} = x \text{ in } x \\
\quad \text{if } e = \text{new } C \\
\quad \text{and } \neg\text{subord}(C) \\
\text{let } x, \_ = \text{new } C \text{ in} \\
\quad \text{let } \_, \_ = x.\text{dominator} = \text{this}.\text{dominator} \text{ in } x \\
\quad \text{if } e = \text{new } C \\
\quad \text{and } \text{subord}(C) \\
e \quad \text{otherwise}
\end{array} \right.$$

■ **Figure 17** Translating  $\mathcal{K}$  expressions into  $\mathcal{K}_F$ .

## H Thread-Safe Configurations in $\mathcal{K}_F$

As discussed in § 7, in any program translated from  $\mathcal{K}$  to  $\mathcal{K}_F$ , there is a one to one mapping between types. Thus, for simplicity, in the following definitions, we will write *e.g.*,  $t_1 \otimes t_2$  about types in  $\mathcal{K}_F$  to mean that the corresponding types in  $\mathcal{K}$  are safely composable. We will also write the full name **threadSafe** rather than abbreviating it to **TS**.

The definition of a thread-safe configuration is admittedly quite involved. We attempt to modularise the definition by describing it in stages below. The complexity of thread-safe configuration mirrors the complexities of maintaining data-race freedom in the presence of shared mutable state and the power of the invariants of  $\mathcal{K}$ —thread-affinity, linearity, strong encapsulation etc.—which are all underlying data-race freedom.

Notes:

1. Since the dominator field is not used in the program, it is excluded in  $f$  below.
2. To avoid clutter, we omit  $t$  subscripts of values where they are not necessary.

### H.1 Safe Configuration

We say that a configuration is safe if its heap and threads (including the stack) are safe.

$$\Gamma \vdash \mathbf{threadSafe}(\langle H; V; T \rangle) \equiv \left\{ \begin{array}{l} \mathbf{safeHeap}(H) \\ \wedge \\ \Gamma; H \vdash \mathbf{safeThread}(V; T) \end{array} \right.$$

We now move on to define these concepts below one by one.

### H.2 Safe Heap

In a safe heap, aliasing between thread-local objects requires that both objects are local to the same thread. Further, an alias in a field  $f$  with a statically subordinate type refers an object whose dominator is the same as the dominator of the object containing the  $f$  field. This captures that subordinate objects in an enclosure are only aliased on the heap by other subordinates in the same enclosure, or by the dominator of the enclosure. Last, no field contains a stack-bound value.

$$\begin{aligned} \mathbf{safeHeap}(H) &\equiv \\ &\forall \iota_1. \\ &H(\iota_1) = (C, F[f \mapsto \iota_2], \_)_{tid} \wedge \\ &\mathbf{fields}(C)(f) = t \wedge \\ &\mathbf{thread}(t) \Rightarrow \\ &\quad \mathbf{creator}(\iota_2) = tid \\ &\wedge \\ &\mathbf{subord}(t) \Rightarrow \\ &\quad \iota_2.\mathbf{dominator} \in \{\iota_1.\mathbf{dominator}, \mathbf{null}\} \\ &\wedge \\ &\neg \mathbf{S}(\iota_2) \end{aligned}$$

where  $\mathbf{creator}(\iota) = tid$  if  $H(\iota) = (\_, \_, \_)_{tid}$  and  $\mathbf{S}(v) \equiv v = \iota^*$ .

Note that we allow dominators to be **null**. This is because in  $\mathcal{K}_F$ , dominators are inserted after object creation. It is easy to see that all object creation in a  $\mathcal{K}_F$  program translated from

a  $\mathcal{K}$  program will immediately assign dominators before the objects are used. To “atomically” assign the correct dominator we could extend  $\mathcal{K}_F$  with constructors, which would solve the “problem”.

### H.3 Safe Threads and Forks

A single thread is safe if its expression is safe and the corresponding stack is safe.

$$\begin{aligned} \Gamma; H \vdash \mathbf{safeThread}(V; (\mathcal{L}, e)_{tid}) &\equiv \\ \Gamma; H \vdash \mathbf{safeStack}(V, tid) \wedge & \\ \Gamma \vdash \mathbf{safeExpr}(H; V; e; tid) & \end{aligned}$$

A fork is safe if both parallel threads are safe and do not share any variables and the expression to be run after the join is safe. Also, if **linear** capabilities alias across the stack, the conjunction of their types must be well-formed.

$$\begin{aligned} \Gamma; H \vdash \mathbf{safeThread}(V; T_1 \parallel T_2 \triangleright e) &\equiv \\ \Gamma = \Gamma_1 + \Gamma_2 &\Rightarrow \\ V = V_1 + V_2 &\Rightarrow \\ \Gamma_1; H; \vdash \mathbf{safeThread}(V_1; T_1) \wedge & \\ \Gamma_2; H; \vdash \mathbf{safeThread}(V_2; T_2) \wedge & \\ \forall x_n, y_n. V_1(x) = V_2(y) \wedge (\mathbf{linear}(\Gamma(x)) \vee \mathbf{linear}(\Gamma(y))) &\Rightarrow \\ \Gamma(x) \otimes \Gamma(y) & \\ \wedge & \\ \Gamma \vdash \mathbf{safeExpr}(H; V; e; tid) & \end{aligned}$$

where  $tid$  is the thread id of the leftmost thread in  $T_1$ .

This definition uses a frame-rule for variable maps, which works similarly to the frame rule for typing environments (*cf.*, (WF-FRAME)). Its purpose is to hide the variables which do not belong to a thread so that the recursive instances of **safeThread** can reason about “all variables” instead of “all variables that are currently accessible to this thread”.

### H.4 Safe Stack

A stack for a thread  $tid$  is safe if all variables of subordinate type point to objects dominated by the closest dominating **this** on an earlier stack frame (this means that inside a dominator, all reachable subordinate objects belong to that dominator), and all variables holding thread capabilities must point to objects which are local to  $tid$ .

$$\begin{aligned} \Gamma; H \vdash \mathbf{safeStack}(V, tid) &\equiv \\ \forall x_n. & \\ V(x_n) = v \wedge & \\ \Gamma(x_n) = t \wedge & \\ \mathbf{thread}(t) \Rightarrow v = \mathbf{null} \vee \mathbf{creator}(v) = tid \wedge & \\ \mathbf{subord}(t) \Rightarrow \mathbf{domination}(H; V; v) & \end{aligned}$$

where  $\text{domination}(H; V; \iota)$  is defined thus:

$$\begin{aligned}
& \mathbf{domination}(H; V; \iota_1) \\
& \iota_1.\mathbf{dominator} = \mathbf{null} \vee \\
& \exists \mathbf{this}_m . \\
& \quad V(\mathbf{this}_m) = \iota_2 \wedge \\
& \quad \iota_1.\mathbf{dominator} = \iota_2 \wedge \\
& \quad m \leq \mathbf{topframe}(V) \wedge \\
& \quad \forall i, \mathbf{this}_i . \\
& \quad \quad V(\mathbf{this}_i) = \iota_3 \wedge \\
& \quad \quad \iota_3.\mathbf{dominator} = \iota_3 \wedge \\
& \quad \quad i \leq m
\end{aligned}$$

where  $\mathbf{topframe}(V)$  extracts the current stack frame number (*i.e.*, the highest stack frame number on any variable in  $V$ ).

## H.5 Safe Expression

An expression is safe in the thread  $tid$  if all values embedded in it respect thread-locality and subordination analogously to **safeStack**. Further, dereferenced variables not wrapped in a corresponding **wlock** or **rlock** must not have types which require such locks to be taken (locked and unsafe). Note that **safeAliasing** is checked here because the locations “in registers”, *i.e.*, those that are embedded in the program, must be checked for aliasing against the variables and fields on the stack and heap.

$$\begin{aligned}
& \Gamma \vdash \mathbf{safeExpr}(H; V; e; tid) \equiv \\
& \Gamma \vdash \mathbf{safeAliasing}(H; V; e) \wedge \\
& \forall x \in \mathbf{unprotectedTargets}(e) . \\
& \quad \Gamma(x) = t \Rightarrow \\
& \quad \quad \neg \mathbf{locked}(t) \wedge \\
& \quad \quad \neg \mathbf{unsafe}(t) \wedge \\
& \forall \iota_t \in \mathbf{locations}(e) . \\
& \quad \mathbf{thread}(t) \Rightarrow \mathbf{creator}(\iota_t) = tid \\
& \quad \mathbf{subord}(t) \Rightarrow \mathbf{domination}(H; V; \iota) \\
& \exists \mathcal{L} . \\
& \quad \Gamma \vdash \langle H; V; (\mathcal{L}, e)_{tid} \rangle : t \text{ for some } t
\end{aligned}$$

where  $\mathbf{unprotectedTargets}(e)$  means all variables  $x$  such that  $x$  is dereferenced ( $x.m(e)$ ,  $x.f$ ,  $x.f = e$ , etc.) in  $e$  but not locked inside a **wlock**( $x, \_$ ) or **rlock**( $x, \_$ ).

## H.6 Safe Aliasing

Aliasing is safe if all aliases either: have composable types (meaning they point to different parts of the same object modulo safe **val** fields); are protected (*i.e.*, read-only aliases, or safe aliases that use locks internally or unsafe aliases whose accesses must be wrapped in locks); are local to the same thread; or the types are subordinate. If two linear values alias, at least one of them must be stack-bound. Also, origins of borrowed values are buried.

Burying is slightly involved: a stack-bound linear value  $v_1$  is either borrowed from a variable on the same stack, or from a linear value  $\iota$  on the heap in which case there is a path from the current stack to  $\iota$  which consists of linear references only (see def. of **buried**). This

guarantees that there is no accessible path in the program which may reach  $v_2$  while it is reversely borrowed on the stack.

$$\begin{aligned}
\Gamma \vdash \mathbf{safeAliasing}(H; V; e) &\equiv \\
&\forall v_1, v_2, t_1, t_2. \\
&\mathbf{onHeap}(H, v_1, t_1) \vee \mathbf{onStack}(\Gamma, V, v_1, t_1) \vee \mathbf{inReg}(e, v_1, t_1) \vee \\
&\mathbf{onHeap}(H, v_2, t_2) \vee \mathbf{onStack}(\Gamma, V, v_2, t_2) \vee \mathbf{inReg}(e, v_2, t_2) . \\
&v_1 = v_2 \Rightarrow \\
&v_1 = \mathbf{null} \vee \\
&t_1 \otimes t_2 \vee \\
&\mathbf{protected}(t_1) \wedge \mathbf{protected}(t_2) \vee \\
&\mathbf{thread}(t_1) \wedge \mathbf{thread}(t_2) \vee \\
&\mathbf{subord}(t_1) \wedge \mathbf{subord}(t_2) \vee \\
&\mathbf{linear}(t_1) \wedge \mathbf{linear}(t_2) \wedge \\
&\mathbf{S}(v_1) \vee \mathbf{S}(v_2) \\
&\wedge \\
&\mathbf{S}(v_1) \Rightarrow \\
&\exists y. V(y) = \iota \wedge \iota = v_1 \quad (\text{n.b. } \iota \text{ not tainted}) \\
&\vee \\
&\exists \iota . \\
&H(\iota) = (C, F[f \mapsto v_1], \_)_{tid} \wedge \\
&\mathbf{field}(C)(f) = t' \wedge \\
&\mathbf{linear}(t') \Rightarrow \\
&\exists y. V(y) = \iota' \wedge \mathbf{buried}(H, \iota', \iota) \wedge \Gamma(y) = t \wedge \mathbf{linear}(t)
\end{aligned}$$

where  $\mathbf{protected}(t) \Leftrightarrow \mathbf{safe}(t) \vee \mathbf{unsafe}(t)$  and  $\mathbf{S}(v) \equiv v = \iota^*$ .

To avoid unnecessary complexity, we assume  $v_1$  and  $v_2$  above are obtained from different fields/variables/expressions, since otherwise  $v_1$  and  $v_2$  are not aliases. For example, if  $\mathbf{onHeap}(H, v_1, t_1)$  and  $\mathbf{onHeap}(H, v_2, t_2)$ , the values were obtained from different fields.

$$\mathbf{onHeap}(H, v, t) \equiv \left\{ \begin{array}{l} \exists \iota, f. H(\iota) = (C, F[f \mapsto v], \_) \\ \wedge \\ \mathbf{fields}(C)(f) = t \end{array} \right.$$

$$\mathbf{onStack}(\Gamma, V, v, t) \equiv \left\{ \begin{array}{l} \exists x. V(x) = v \\ \wedge \\ \Gamma(x) = t \end{array} \right.$$

$$\mathbf{inReg}(e, v_t, t) \equiv v_t \in \mathbf{locations}(e)$$

where  $\mathbf{locations}(e)$  is the set of locations embedded in  $e$ .

Last,  $\mathbf{buried}(H, \iota, \iota')$  means that there is a path of linear references leading from  $\iota$  to  $\iota'$ :

$$\frac{\exists f. H(\iota) = (\_, F[f \mapsto \iota'_t], \_) \quad \mathbf{linear}(t)}{\mathbf{buried}(H, \iota, \iota')}$$

$$\frac{\exists f, \iota''. H(\iota) = (\_, F[f \mapsto \iota''_t], \_) \quad \mathbf{linear}(t) \quad \mathbf{buried}(H, \iota'', \iota')}{\mathbf{buried}(H, \iota, \iota')}$$

## H.7 Proof of H.1–H.6

The formulation of preservation of thread-safety (aka **TS**) is:

► **Preservation of Thread-Safety.** In a program translated from  $\mathcal{K}$ , if a thread-safe configuration  $\langle H; V; T \rangle$  can step to  $\langle H'; V'; T' \rangle$ , then  $\langle H'; V'; T' \rangle$  is also thread-safe.

$$\begin{aligned} & \forall \Gamma, H, H', V, V', T, T'. \\ & \Gamma \vdash \mathbf{threadSafe}(\langle H; V; T \rangle) \wedge \\ & \langle H; V; T \rangle \hookrightarrow \langle H'; V'; T' \rangle \Rightarrow \\ & \exists \Gamma'. \Gamma' \vdash \mathbf{threadSafe}(\langle H'; V'; T' \rangle) \wedge \Gamma \subseteq \Gamma' \end{aligned}$$

The proof is by induction over the thread structure.

### Initial Configuration is threadSafe

We begin by establishing that the initial configuration is thread-safe.

1. **safeHeap**( $\epsilon$ ) trivially holds.
2.  $\Gamma; \epsilon \vdash \mathbf{safeStack}(\epsilon, tid)$  trivially holds.
3.  $\Gamma \vdash \mathbf{safeAliasing}(\epsilon; \epsilon; e)$  trivially holds because there is not yet any location in  $e$ .
4. The requirement of **safeExpr** that all variables whose types are not **locked** or **unsafe** are wrapped in lock statements follows from the translation from  $\mathcal{K}$  to  $\mathcal{K}_F$ , and the  $\mathcal{K}$  rule (E-CALL) that disallows calling methods on **unsafe** receivers without a **sync**.
5.  $\Gamma \vdash \langle H; V; (\emptyset, e) \rangle : t$  holds, see initial configuration in preservation.
6. By 3–5.),  $\Gamma \vdash \mathbf{safeExpr}(\epsilon; \epsilon; e; tid)$ .
7. By 2.) and 6.),  $\Gamma; \epsilon \vdash \mathbf{safeThread}(\epsilon; (\emptyset, e)_{tid})$ .
8. By 1.) and 7.),  $\Gamma \vdash \mathbf{threadSafe}(\langle \epsilon; \epsilon; (\emptyset, e)_{tid} \rangle)$ .

### Case Analysis on the Shape of $T$

WLOG, to simplify matters slightly, we impose an additional constraint on (E-CALL):  $\neg \mathbf{linear}(t_1)$ . This disallows method calls on linear receivers and requires a borrowing to be inserted for each call. This does not affect the semantics of  $\mathcal{K}$  since a linear receiver is effectively borrowed and buried on the underlying stack-frame when target of a method call.

#### Case 1: $(t) v$

**Upcast** Immediate since no aliases are added. While  $v$  may change type,  $\mathcal{K}$  rules for upcasting requires all modes to be preserved (see (T-SUB-STRUCTURAL) etc.).

**Downcast** The only time  $\mathcal{K}$  uses downcasts are when packing. In packing, capabilities grow.

Thus, if two aliases  $v_1$  and  $v_2$  have types  $t_1$  and  $t_2$  such that  $t_1 \otimes t_2$ , downcasting  $v_1$  to enlarge its type may break  $t_1 \otimes t_2$ ,

However, from translation of (PACK) it is easy to see that pack only succeeds if  $v_1 = v_2$  and both  $v_1$  and  $v_2$  are consumed. Thus, if either  $t_1$  and  $t_2$  are linear, they will be consumed meaning that linearity is preserved.

#### Case 2: null

Immediate.

**Case 3: let  $x, y = v$  in  $e$** 

This case moves a location from the expression to the stack, and then creates another copy of the same location on the stack.

The alias constraints on  $V(x)$  (thread-locality, domination) in **safeStack** are identical to **safeExpr**.

However, the *additional* alias must not invalidate any properties of **safeStack** or **safeAliasing**.

Let  $t$  be the type of  $v$ .

From the translation from  $\mathcal{K}$  to  $\mathcal{K}_F$ , we know that the only case in which  $y$  is actually introduced is when translating an `unpack`. From (E-UNPACK),  $t \Rightarrow t' \otimes t''$ . Thus, **safeAliasing** is satisfied.

**Case 4: wlock( $x, r$ ) in  $e$** **Case 4.1: (DYN-EVAL-WLOCK)**

In this case, an  $x$  is removed from  $V$  and  $V(x)$  is added to  $e$ .

The alias constraints on  $V(x)$  (thread-locality, domination) in **safeExpr** are identical to **safeStack**.

**Case 4.2: (DYN-EVAL-WLOCK-REENTRANT)**

Immediate as no aliases are added.

**Case 5: rlock( $x, r$ ) in  $e$** 

Analogous to Case 4.

**Case 6: wlocked<sub>( $\iota, r$ )</sub>{ $e$ }**

Immediate since no aliases are added.

**Case 7: rlocked<sub>( $\iota, r$ )</sub>{ $e$ }**

Immediate since no aliases are added.

**Case 8: new  $C$** 

We add a fresh value  $\iota$  to the expression, pointing to an object whose fields are all **null**.

(DYN-EVAL-NEW) satisfies the constraints on creator ids.

The dominator field is **null**, which is permitted.

All fields are **null**, so no field holds a stack-bound.

**Case 9: taint  $x$** 

By **safeAliasing**, adding an additional stack-bound value to the expression is allowed, even when  $\Gamma(x) = t$  s.t. **linear**( $t$ ).

Let  $V(x) = v$ .

From the translation of  $\mathcal{K}$  into  $\mathcal{K}_F$ , we know that **taint** is only used in **bound**. By (E-FORWARD), only non-stack-bound variables (*i.e.*, with non-tainted values) on the stack may be tainted. Thus,  $\exists y. V(y) = \iota$  and  $\iota = v$  (not tainted) from **safeAliasing** is satisfied.

**Case 10: taint  $x.f$** 

Similar to Case 9 but with reverse borrowing. In this case, we introduce a tainted alias to a possibly linear field on the heap. Let  $v$  be such that  $H(V(x))(f) = v$ .

By (E-REVERSE), we may only reverse borrow from linear targets, so the type of  $x$  must also be linear. Thus **buried**( $H, V(x), v$ ), which together with the fact that the type of  $x$  is linear satisfies the reverse borrowing constraint from **safeAliasing**.

**Case 11: assert ( $x==y$ ); $e$** 

Immediate since no aliases are added.

**Case 12:**  $x$ 

This case adds an additional alias to the expression.

By (E-VAR), we know  $\Gamma(x) = t$  s.t.  $\neg\mathbf{linear}(t)$ , thus the alias constraints of **safeAliasing** are satisfied.

The alias constraints on  $V(x)$  (thread-locality, domination) in **safeExpr** are identical to **safeStack**.

**Case 13:** **consume**  $x$ 

Analogous to Case 12, but by (E-CONSUME), we know  $\Gamma(x) = t$  s.t.  $\mathbf{linear}(t)$ .

If  $x$  was aliased initially, this aliasing is safe by **safeAliasing**, **safeStack** and **safeThread**.

By (DYN-EVAL-CONSUME),  $x$  is nullified. Thus, no extra alias has been created, so regardless of whether  $V(x)$  was stack-bound or not, the aliasing constraints are still satisfied.

**Case 14:**  $x.f$ 

This case adds a new alias to the expression. Let  $\iota$  be s.t.  $H(V(x))(f) = \iota$ . We ignore the case when  $x.f$  is **null** as it is subsumed by the case below.

The alias constraints on  $V(x)$  (thread-locality, domination) in **safeExpr** are identical to **safeStack**.

By (E-SELECT), we know  $\mathbf{fields}(\Gamma(x))(f) = t$  s.t.  $\neg\mathbf{linear}(t)$ , thus the alias constraints of **safeAliasing** are satisfied.

For the **domination** property to hold for  $\iota$ , either  $\iota.\mathbf{dominator} = \mathbf{null}$  or the closest dominator  $\iota'$  on an earlier stack frame must be such that  $\iota.\mathbf{dominator} = \iota'$ .

By **safeHeap**, either  $\iota.\mathbf{dominator} = \mathbf{null}$  or  $\iota.\mathbf{dominator} = V(x).\mathbf{dominator}$ .

There are two possible cases:  $x$  is a dominator or  $x$  is not a dominator.

In the first case,  $x$  is clearly the closest dominator (since it is on the same stack frame, remember  $x = \mathbf{this}$  as variable access is only allowed on **this**), thus

$$V(x).\mathbf{dominator} = V(x).$$

In the second case,  $V(x)$ 's dominator must be the closest dominator,  $\iota'$  (follows from the use of **domination** in **safeStack** since  $x$  is a variable on the stack), thus

$$\iota.\mathbf{dominator} = V(x).\mathbf{dominator}, \text{ i.e., } \iota'.$$

For the thread-locality property to hold,  $\mathbf{creator}(\iota)$  must be the identity of the current thread,  $\mathbf{tid}$ .

From (WF-FD) and **safeHeap**, if  $\mathbf{thread}(t)$ , then  $\Gamma(x) = t'$  s.t.  $\mathbf{thread}(t')$  and  $\mathbf{creator}(\iota) = \mathbf{creator}(V(x))$ . From **safeStack**,  $\mathbf{thread}(t') \Rightarrow \mathbf{creator}(V(x)) = \mathbf{tid}$ .

**Case 15:** **consume**  $x.f$ 

Analogous to Case 14 but the type can be linear. However, by

(DYN-EVAL-CONSUME-FIELD),  $x.f$  is destructively read, so even if there is an alias to  $x.f$  somewhere, the value *transferred* to the expression satisfies the constraints of **safeAliasing**, since **safeAliasing** holds for the initial state by the induction hypothesis.

**Case 16:**  $x.f = v$ 

The case is analogous to Case 14. The rules (WF-FD) and (E-UPDATE) statically prevents us from assigning a stack-bound to a field (subtyping rules does not allow adding or dropping a “box”). From the translation function and preservation, only stack-bound values are tainted at run-time.

**Case 17:** **finish** { **async** {  $e_1$  } **async** {  $e_2$  } } ;  $e_3$ 

We must satisfy **safeThread** which splits the stack up into two parts for which linear values may only alias if the conjunction of their types is well-formed.

To break this constraint, we must have two aliasing linear variables  $x$  and  $y$  in  $V$  typed  $t_x$  and  $t_y$  respectively s.t.  $\neg(t_x \otimes t_y)$  on the same stack frame.

**safeAliasing** allows this when at least one variable is stack-bound. However, by (E-FORWARD), borrowing a variable buries (hides) the original, and it is not permitted to **bound** an already stack-bound value.

Hence, two aliasing linear stack variables must be on different stack frames.

**Case 18:**  $x.m(v)$

In this case  $V$  is extended by a new stack frame with a new **this<sub>n</sub>** (creating an alias from stack to stack on the same thread), and the argument (creating an alias from expression to stack on the same thread).

As the receiver of the call is already aliased on the stack before the call, **safeAliasing** holds. Here we make use of our simplification and rely on the fact that if **linear**( $\Gamma(x)$ ), then  $x$  is also borrowed, *i.e.*, **S**( $\Gamma(x)$ ) holds. This satisfies the constraints on the new alias in **safeStack**.

The alias constraints on the new stack-based alias to  $v$  (thread-locality, domination) in **safeStack** are identical to **safeExpr**.

The translation function from  $\mathcal{K}$  to  $\mathcal{K}_F$  inserts locks around all dereferences of **locked** or **unsafe** typed variables.

There are no locations in the new  $e$  added to the stack.

**Case 19:**  $T \parallel T' \triangleright e$

**Case 19.1** : Reduce left

Follows from induction hypothesis.

**Case 19.2** : Reduce right

Follows from induction hypothesis.

**Case 19.3** :  $T = (L, v)$  and  $T' = (L', v')$

Immediate since no aliases are created.

## I Data-Race Freedom

► **Data-Race Freedom.** If a safe configuration  $\langle H; V; T \rangle$  steps to two different configurations causing effects  $Eff_1$  and  $Eff_2$  respectively, then these effects are non-conflicting:

$$\begin{aligned} & \forall \Gamma, H, V, T, cfg' \text{ } cfg'' . \\ & \Gamma \vdash \mathbf{threadSafe}(\langle H; V; T \rangle) \wedge \\ & \langle H; V; T \rangle \xrightarrow{Eff_1} cfg' \wedge \\ & \langle H; V; T \rangle \xrightarrow{Eff_2} cfg'' \Rightarrow Eff_1 \# Eff_2 \vee cfg' = cfg'' \end{aligned}$$

where

$$\begin{aligned} & \mathbf{rd}(l.f) \# \mathbf{rd}(l'.f') \\ & \_ (l.f) \# \mathbf{wr}(l'.f') \quad \text{iff } l \neq l' \vee f \neq f' \\ & \varepsilon \# \_ \\ & Eff_1 \# Eff_2 \quad \text{iff } Eff_2 \# Eff_1 \end{aligned}$$

### Proof

We focus on the case where there are just two threads  $T_1$  and  $T_2$  running in parallel in a thread-safe configuration. This is without loss of generality as the interactions between any number of threads in a single step can be reduced to reasoning about the interaction between any pair of threads.

Assume that the next expression to be evaluated in  $T_1$  is  $x.f = \mathbf{null}$  (causing the effect  $\mathbf{wr}(l_x.f)$ ) and the next expression for  $T_2$  is  $y.f$  (causing the effect  $\mathbf{rd}(l_y.f)$ ). We show that in a thread-safe configuration, these two operations cannot race. (The write–write race is analogous.)

The case where  $x$  and  $y$  point to different objects is trivial, so we assume that they are aliases of the location  $l$ . Furthermore, let the type of  $x$  be  $t_x$  and the type of  $y$  be  $t_y$ . By the definition of **threadSafe**, we have that for any two aliasing variables in different threads, either  $t_x \otimes t_y$  or both  $t_x$  and  $t_y$  are **protected**, **thread**, **subord** or **linear**. We proceed by case analysis:

#### Case 1: $t_x \otimes t_y$

By the rules in Figure 7 two traits in a disjunction can only share **val** fields. But by (E-UPDATE) the expression  $x.f = \mathbf{null}$  is only allowed if  $f$  is a **var**-field. Thus, this case leads to a contradiction.

#### Case 2: $\mathbf{protected}(t_x) \wedge \mathbf{protected}(t_y)$

We proceed on the possible combinations of different safe capabilities:

##### Case 2.1: $\mathbf{locked}(t_x)$

By **safeExpr**,  $x$  must be protected by a write lock. By (WF-L-THREAD) and (WF-LOCKS) the lock for the region  $r$  of the field  $f$  must be taken, and the lock  $(l, r)$  must be in the lock set of  $T_1$ .

##### Case 2.1.1: $\mathbf{locked}(t_y)$

By the same rules as above, if  $t_y$  is locked, the lock  $(l, r)$  must be in the lock set of  $T_2$ . But (WF-L-ASYNC) requires that the lock sets of two parallel threads are disjoint. This combination of capabilities leads to a contradiction.

**Case 2.1.2: read( $t_y$ )**

If  $T_2$  can access  $\iota.f$  through a **read** reference at the same time as  $T_1$  accesses it through a **locked**, the class  $C$  of  $\iota$  must be (at least) a disjunction of a **read**-trait and a **locked**-trait. In this case the translation of  $C$  would have inserted a read-lock for all the methods of  $t_y$ . As we only access fields through **this**, the expression  $y.f$  must be wrapped in a read lock. By (WF-L-THREAD) the read lock for the region  $r$  of the field  $f$  must be taken, but this contradicts the assumption that the write lock of the same region is taken.

**Case 2.1.3: unsafe( $t_y$ )**

Analogous to **Case 2.1.1**.

**Case 2.2: read( $t_x$ )**

If  $t_x$  is **read**, all fields in the trait from which the current method was translated are **val**-fields. But, by (E-UPDATE) the expression  $x.f = \mathbf{null}$  is only allowed if  $f$  is a **var**-field. Thus, this case leads to a contradiction.

**Case 2.3: unsafe( $t_x$ )**

Analogous to **Case 2.1**

**Case 3: subord( $t_x$ )  $\wedge$  subord( $t_y$ )**

From the definition of **safeStack**, if  $t_x$  is **subord**, then there exists a dominator active as the current **this** on an earlier stack frame of the same thread, and likewise for  $t_y$ . Let **this<sub>x</sub>** and **this<sub>y</sub>** designate the variables holding these dominators.

Analysing when **this<sub>x</sub>** and **this<sub>y</sub>** can alias, we can recursively apply the same reasoning to these variables. WLOG, assume that the dominators have fields pointing to  $x$  and  $y$ . If the dominator is **linear**, then the types of **this<sub>x</sub>** and **this<sub>y</sub>** must form a conjunction by **safeThread**. However, type-compatible subordinates in a conjunction are not allowed (by (T-COMPOSITION) and (C-VAL-VAL)) leading to a contradiction.

If the dominator is **read**,  $x$  could not be of **subordinate** type since all fields of **read** capabilities are **safe** capabilities (WF-FD).

If the dominator is **thread**, then by **safeStack**, **this<sub>x</sub>** and **this<sub>y</sub>** must be held by the same thread, leading to a contradiction.

If the dominator is **locked** or **unsafe**, then the lock sets for both threads must hold the corresponding object. By preservation, this is only possible for read locks. However, inside **locked** or **unsafe** capabilities, write locks are always acquired.

Thus, we can conclude that two subordinate capabilities on different stacks may not alias and therefore  $x \neq y$ .

**Case 3: linear( $t_x$ )  $\wedge$  linear( $t_y$ )**

By **safeThread**,  $t_x \otimes t_y$ . Thus, Case 1 applies.

**J Strong Encapsulation of Subordinate Capabilities**

Strong encapsulation in  $\mathcal{K}$  is similar to ownership types [17] and external uniqueness [18].

At run-time in  $\mathcal{K}_F$ , instances know the identity of their dominator. This identity is invariant, even under ownership transfer, because transfer operates on linear capabilities and instances of classes without a subordinate capability are their own dominators.

Let  $\rightarrow$  denote “refers to” and  $\iota.\text{dom}$  denote the dominator for an object with id  $\iota$ . Now,  $\forall \iota, \iota' \in \text{dom}(H)$ ,  $\iota \rightarrow \iota'$  s.t.  $\iota \neq \iota'$ , either one of the following holds:

1.  $\iota'.\text{dom} = \iota.\text{dom}$  (a pointer between subordinates in the same enclosure)
2.  $\iota'.\text{dom} = \iota$  (a dominator pointing to one of its subordinates)

3.  $\iota.\text{dom} = \iota'$  (a subordinate pointing to its dominator)  
 or  $\iota'$  is a top-level object, *i.e.*,  $\iota'.\text{dom} = \iota'$ .

### Proof

Straightforward derivation from H.2 Safe Heap.

The first key ingredient to show this part of thread-safety is how the dominator of a new instance  $\iota$  of type  $t$  is assigned in a translated  $\mathcal{K}$  program: **subordinate**( $t$ ) means  $\iota$ 's dominator is the dominator of the current **this**; otherwise,  $\iota$  is its own dominator. The dominator is stored in a field which is never used in the program, except for the meta-theoretic reasoning.

The second key ingredient is the  $\mathcal{K}$  equivalent of the static visibility constraint [17], found in (E-CALL): (**subord**( $t_2$ )  $\vee$  **subord**( $t_3$ ))  $\Rightarrow$  **encaps**( $t_1$ )  $\vee$   $x \equiv$  **this** where  $t_2$  and  $t_3$  are the argument and return types of a method, and  $t_1$  the type of the receiver. This constrains methods taking or returning subordinate objects to only be called on other subordinates (*i.e.*, objects in the same enclosure) or when the receiver is **this** (*i.e.*, the dominator).