Dala: A Simple Capability-Based Dynamic Language Design For Data Race-Freedom

Kiko Fernandez-Reyes
Uppsala University
Sweden
kiko.fernandez@it.uu.se

Isaac Oscar Gariano
Victoria University of Wellington
New Zealand
isaac@ecs.vuw.ac.nz

James Noble
Victoria University of Wellington
New Zealand
kjx@ecs.vuw.ac.nz

Erin Greenwood-Thessman
Victoria University of Wellington
New Zealand
erin.greenwood-thessman@ecs.vuw.ac.nz

Michael Homer
Victoria University of Wellington
New Zealand
mwh@ecs.vuw.ac.nz

Tobias Wrigstad
Uppsala University
Sweden
tobias.wrigstad@it.uu.se

Abstract

Dynamic languages like Erlang, Clojure, JavaScript, and E adopted data-race freedom by design. To enforce data-race freedom, these languages either deep copy objects during actor (thread) communication or proxy back to their owning thread. We present Dala, a simple programming model that ensures data-race freedom while supporting efficient inter-thread communication. Dala is a dynamic, concurrent, capability-based language that relies on three core capabilities: immutable values can be shared freely; isolated mutable objects can be transferred between threads but not aliased; local objects can be aliased within their owning thread but not dereferenced by other threads. Objects with capabilities can co-exist with unsafe objects, that are unchecked and may suffer data races, without compromising the safety of safe objects. We present a formal model of Dala, prove data race-freedom and state and prove a dynamic gradual guarantee. These theorems guarantee data race-freedom when using safe capabilities and show that the addition of capabilities is semantics preserving modulo permission and cast errors.

CCS Concepts: • **Computing methodologies** → **Concurrent programming languages**; • **Theory of computation** → *Type theory*.

Keywords: concurrency, capability, permission, isolation, immutability

ACM Reference Format:
Kiko Fernandez-Reyes, Isaac Oscar Gariano, James Noble, Erin Greenwood-Thessman, Michael Homer, and Tobias Wrigstad. 2021. Dala: A Simple Capability-Based Dynamic Language Design For Data Race-Freedom. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '21), October 20–22, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3486607.3486747

Onward! '21, October 20–22, 2021, Chicago, IL, USA

1 Introduction

Most mainstream object-oriented languages do not rule out *data races* – read-write or write-write accesses to a variable by different threads without any interleaving synchronisation. This makes it hard to reason about the correctness, or in programming languages like C and C++ even the meaning of programs. Static languages such as Java or Go have higher level constructs to control concurrency but, ultimately, nothing strictly prevents data races from happening. Dynamic languages such as Ruby or Python repeat the same story: nothing prevents data races, not even a global interpreter lock.

Many object-oriented languages added concurrency constructs as an afterthought, and objects may suffer from data races. Data race free languages have implicit concurrent properties as part of the "object model" and guarantee data race-freedom. Examples of data race free languages are: E which uses object capabilities and far references to forbid access to (global) and un-owned resources [54], capability- or ownership-based languages such as Pony [18, 19] or Rust [53], or languages without mutable state, *e.g.,* Erlang [2].

Freedom from data races simplifies avoidance of *race conditions*, which happen when behaviour is controlled by factors outside of the program's control, such as the scheduling of two threads. Data race free languages thus have a leg up on "racy" languages in this respect, but data race freedom always comes at a cost: languages either deliver "efficient concurrency" or "simple concurrency" but not both.

Table 1 shows the features of eight data race free languages. (Dala is our proposal, and we discuss its features later in the paper.) The first four languages are statically typed. To maintain data race-freedom, these introduce new concepts which permeate a system to: ownership, capabilities, capability composition, capability subtyping, capability promotion and

Table 1. Summary of features of capability-based static languages

Complexity \ Languages	Pony	Rust	Encore	RefImm	E	Newspeak	AmbientTalk	Erlang	Dala
Capabilities	6	5	7+	4	✓	✓	✓	✗	3
Capability Subtyping	✓	✗	✓	✓	✗	✗	✗	✗	✗
Promotion, Recovery, Borrowing	✓	✓	✓	✓	✗	✗	✗	✗	✗
Compositional Capabilities	✓	✓	✓	✗	✗	✗	✗	✗	✗
Deep copying	✗	✗	✗	✗	✓	✓	✓	✓	✗
Far References	✗	✗	✗	✗	✓	✓	✓	✗	✗
Data-Race Freedom	✓	✓	✓	✓	✓	✓	✓	✓	✓

recovery, and viewpoint adaption [12, 17, 19, 30, 53] (though not necessarily all at once). This may have lead to a steep learning curve [49]. In return, these languages deliver efficient concurrency: they allow large object graphs to be shared or passed around safely by pointer, or allow multiple threads to access different places of a single data structure at once.

The next four languages are dynamically typed. These maintain data race-freedom implicitly and requires the programmer to do little or nothing: objects are either copied between "threads" [39] (which, while simplifying garbage collection, may be expensive [58] and loses object identity), or proxied back to the "thread" that owns them [11, 26, 55, 69], which adds latency and makes performance hard to reason about unless it is clear what operations are asynchronous.[1]

The goal of this work is to deliver a design that provides both simplicity and performance, and that may work both in dynamically and statically typed programming languages. To this end we present Dala,[2] a capability-based and dynamic approach to data race-freedom without mandating deep copying and without the typical complexity of capability systems. The Dala model allows mixing objects which are guaranteed to be safe from data-races with objects that are not and thereby supports the gradual migration of programs to using only "safe objects," by converting unsafe objects to safe objects one at a time. Dala uses three object capabilities to maintain data race-freedom: immutable values that can be shared freely; isolated mutable objects that cannot be aliased but can be transferred between threads, and thread-local objects that can be aliased across threads but only dereferenced by the thread that created them. In this paper, we study our capabilities in a very simple setting: an untyped object-based language, leaving optional static typing for future work.

To support the design of Dala we contribute Dalarna, a formal model of the core of Dala. We use this model to prove that objects with capabilities cannot be subject to data races, nor can they observe a data race except via explicitly unsafe parameters passed to their methods. We also show that Dala supports a form of gradual guarantee [67, 68]: the addition of capabilities preserves the dynamic semantics, modulo runtime errors that check that the program behaves according to the programmer's explicitly stated intentions, *e.g.,* disallowing a write to an immutable object. The Dala model can be embedded into a wide range of garbage-collected object-oriented languages, such as Java, TypeScript, Ruby, OCAML, Swift, Scala, Go etc. To demonstrate that Dala has the potential to be practicable, for both statically and dynamically typed programs, we have a proof-of-concept implementation, Daddala, which embeds Dala in Grace [3, 41, 57], built on top of Moth VM [61].

Contributions and Outline.

1. We overview three inherent problems in race unsafe and safe programming languages, and discuss the current approaches in ownership- and capability-based systems (§2).
2. We introduce the Dala capabilities that allow safe interaction between programs containing data races from parts that should remain data race-free (§3).
3. We show how Dala tackles the three inherent problems in §2 (§4).
4. We provide a formal description of Dala and its core properties (§§5 and 5.3).
5. We provide Daddala, a proof-of-concept implementation that (anecdotally) shows the relative ease of embedding Dala in an existing system ([32]).

§6 places Dala in the context of related work, and §8 concludes.

2 Background: Perils of Concurrent Programming

To set the scene for this paper we discuss common problems in race unsafe (*e.g.,* Java) and safe concurrent languages (*e.g.,* Rust). First we discuss balancing complexity and performance (§2.1); then how tying safety to particular concurrency abstractions leads to a one-size-fits-all model which leads to problems with compositionality of concurrent abstractions (§2.2); last we discuss the problem of providing escape hatches to permit behaviour that is not supported by the programming language (§2.3).

[1] E, Erlang, and AmbientTalk are also distributed languages which motivates their copying and proxied reference approaches.

[2] A "Dala Horse" is a small carved wooden toy horse — a handmade little pony [19, 31, 62]).

2.1 Balancing Safety, Complexity and Performance

Safe languages have (implicit) concurrency mechanisms to prevent data races. From Table 1, we argue that the constructs or systems fall into three categories: *complex and efficient*, *simple and inefficient*, and *complex and inefficient*.

Complex and Efficient. Pony, Rust, Encore as well as Gordon's work on reference immutability are race-safe by controlling access to shared data, rather than banning its existence [8, 18, 19, 35, 53]. This is achieved by providing concepts that are not common to every day developers, such as capability and ownership type systems. These allow both *efficient* and *data race-safe* sharing and transfer of ownership with reference semantics. This allows *e.g.,* passing a large data structure across actors by reference, which subsequently allows fast synchronous access by the receiving actor. Implementing a concurrent hashmap in these systems requires up-front thinking about how keys and values may be accessed across different threads, and mapping the intended semantics onto the types/capabilities that the languages provide. For example, in Pony, a concurrent hashmap will be an actor; to allow multiple threads to know the existence of the same keys, the keys must be immutable; etc.

The cost of safe efficient data sharing is complexity: capability type systems introduce complex semantics such as capability promotion, capability subtyping, capability recoverability (*e.g.,* getting back a linear reference after it was shared), compositional capability reasoning (*i.e.,* combining capabilities to produce new capabilities), and viewpoint adaption (*i.e.,* how to view an object from another object's perspective, *e.g.,* Pony uses viewpoint adaption to write parametric polymorphism [51]). Understanding these concepts is key to write code that is efficient and free from data races.

Some form of unique/affine/linear reference is often the cornerstone of many of the static capability systems. Such references are extremely powerful: they provide reasoning power, they can often be converted into other capabilities (*e.g.,* to create cyclic immutable data structures) or to transfer ownership of objects across threads. However, polymorphic behaviour is typically a source of great pain for these systems. For example, consider a simple hashmap – concurrent or not. If values can be unique, a lookup must remove the value out of the data structure to preserve uniqueness (and the associated entry to reflect this in the hashmap). If values are not unique, this behaviour is counter-productive. Behaviours like this force duplication of code.

Simple and Inefficient. ASP [10], ProActive [9], Erlang [2], E [54], AmbientTalk [22, 27], Newspeak [6, 7], functional objects in ABS [42], among others, avoid data races by deep copying objects in messages (for some languages modulo far references, see *Complex and Inefficient*). This approach is relatively simple and the price for data race-freedom is copying overhead paid *on every message send.*

In addition to using more CPU and increasing the memory pressure, deep copying loses object identity, and requires traversal of the objects transferred, a $O(\#objects)$ operation, possibly requiring auxilliary data structures (adding overhead) to preserve internal aliasing of the copied structure.

Erlang requires two such traversals: the first calculates the size of all objects to enlarge the receiving process' heap and the second copies them across. Erlang's deep copying is key to keeping process' heaps disjoint, which simplifies concurrent garbage collection and reduce overall system latency. In languages that compile to Java, like ABS, ASP, ProActive, etc., the run-time is unable to see or leverage actor isolation.

A concurrent hashmap in Erlang has no choice: its keys and values must be immutable, and are therefore safe to share across multiple processes (but are copied anyway, see above). The internal hashmap data structures must be immutable too, which might be less efficient.[3]

Complex and Inefficient. To support safe sharing without losing object identity, E [54], AmbientTalk [22, 27], and Newspeak [6, 7] support far references. A far reference is a proxy that allows an object owned by an actor/process to directly reference another's innards, but all interaction with the proxy is lifted into an asynchronous message and sent back to the owner, to be executed there. Thus, despite the fact that many actors can point directly to an object, only its owning process will ever read or write the object. Thus, with proxies, there is a cost *per access* which can be expensive [58].

In E and AmbientTalk, proxied (*far*) references must be operated on asynchronously and only non-proxied (*local*) references allow synchronous access. Code that needs to be "proxy-agnostic" must use asynchronous access. In E and AmbientTalk, promises are also implicit proxied references.[4] This means that asynchronous sends (x←) may be delayed indefinitely if the promise is never fulfilled, meaning the x's value is never materialised so there is no recipient of the message.

Implementing a concurrent hashmap in AmbientTalk (E and Newspeak are similar) does not need any capability annotations to track how values stored in the map may be shared across threads. Promises remove the need for callbacks, but indirections make the implementation more complex or convoluted. As a blocking synchronisation on the result of a promise is not possible, we must use the promise chaining operator to access the (possible) value in the promise.

While syntactically simple, the inability to access resources directly makes reasoning about performance hard. The main implication of not allowing direct access is that we neither

[3] Erlang Term Storage provides a way to escape this design of Erlang, but at a cost of dropping to a much lower-level of programming and manual memory management.

[4] E calls them promises, AmbientTalk calls them futures. We will refer to them as promises [52].

know if far references are ever going to be fulfilled nor if the promise chaining combinator (on far references) makes the owning actor (of the promise) the bottleneck of the system.

2.2 Safe One-Size-Fits-All Concurrency

In race unsafe languages, abstractions are broken by the addition of concurrency constructs. For example, inconsistencies in an object's internal state *during a method's execution*, which are hidden in a purely sequential system, may be *observed* if the object can be accessed concurrently. This problem of object-oriented languages and their "unsafe" concurrency features was first studied by M. Papathomas [59]. Yet, there are plenty of abstractions and programming models that guarantee data-race freedom, such as implementations of the actor model [13, 24, 74]. In languages like *Akka* [74], the model is data race-free as long as all code in a system adheres to a set of guidelines [1].

These guidelines tie concurrency safety properties to their concurrency model, suffering a "one-size-fits-all" problem. For example, Akka can only guarantee data race-freedom when the program (follows the guidelines and) stays within the actor model. Spawning threads in an actor can easily break its concurrency safety, *e.g.,* data race-freedom.

Thus, to implement a concurrent hashmap, an Akka program might simply wrap the standard Java hashmap in an actor. As long as the hashmap itself is never leaked from the actor, and the actor does not create additional threads, all updates the hashmap will be sequential and therefore free from data races. Whether the keys and values are safe from data races is beyond the control of the hashmap, and is ultimately up to the diligence of the programmers (*e.g.,* to maintain uniqueness or immutability).

In E, AmbientTalk, and Newspeak, (as well as many languages that did not fit in Table 1 such as ABS, Pony, Proactive and others), the mechanisms that guarantee data race-freedom are inherently linked to the languages' chosen concurrency models. In this case, the near and far references span (affect) other concurrency models. Using the example from before, global objects can be safely protected by locks but this cannot be easily accommodated[5] and the run-time may throw an exception when multiple threads have access to a shared object [23].

2.3 Safety May Beget Unsafety

In safe languages based on capabilities or ownership types (*e.g.,* [8, 18, 19, 53]) it is sometimes necessary to side-step the type checker, to write low-level code that interacts with hardware, or when the type system is not "clever enough" to allow a correct behaviour. We exemplify these cases in turn.
Hardware Meets Software. Graphic cards constantly read from video memory and developers can write directly on the frame buffer that points to the video memory to update the image in the next refreshing cycle. Implicitly, this means that there is a data race between the graphics card and the main thread; such racing behaviour may show flickering of the image on screen. A double buffering technique removes this flickering, using an on-screen buffer that the graphics card reads, and one off-screen where developers write the next scene. A swap operation swaps the buffer pointers from off- to on-screen. This common – and racy – approach prevents the flickering.
Trust Me, I Know What I Am Doing. As all statically typed languages, capability- and ownership-based programming languages are engaged in a balancing act of expressivity and complexity. Simplifications made to the systems to reduce the programmer's overhead will invariably lead to exclusion of valid programs, simply because the system is not powerful enough to express its behaviour within its model.

We can illustrate this using the concurrent hashmap example in the context of the Encore programming language. Encore suffers from the problem mentioned in §2.1 where a hashmap must choose between supporting unique references (and therefore always moving values in an out), or not (allowing them to be in the hashmap at the same time as they are referenced elsewhere). This can be solved in Encore by dropping to the underlying language to which Encore compiles where no such checks are made, and implement, for example, parallel put and get methods that accepts linear values even though this is unsound in the type system, and transfers them (hopefully) correctly. Other languages (*e.g.,* Rust) may have an unsafe block, or provide reflective constructs that are unsafe from a capability perspective. Naturally, all such code gives rise to technical debt.

In order to circumvent shortcomings of the capability systems, programmers may resort to escape hatches that void the guarantees of data-race freedom. If data races happen inside an escape hatch, the behaviour is undefined [20, 43]. In Pony (and Encore), if an unsafe block introduces a data race the run-time (garbage collector) might eventually crash [20]. There is no safe *interoperability* between unsafe and safe code!

2.4 Summary

The data race-freedom guarantee of safe languages comes at a cost of complexity or inefficiency, or both. Furthermore, most or all languages' safety is tied to a specific concurrency model, and may not compose with others. Finally, data races are sometimes desired, or a systems' notion of safe is too safe to express correct code. Escape hatches overcome these problems, but at a cost of losing data race-freedom.

In the next section, we describe our simple model that is efficient, concurrency-model agnostic, and provides safe interoperability between unsafe and safe code.

[5] This can be partially mitigated by adding futures, which imposes again other concurrent semantics to maintain the safety properties [23]

3 An Overview of The Dala Model

Objects in Dala are associated with capabilities that describe how they can interact with other objects. The association happens at creation time and is fixed for life. We refer to *immutable*, *isolated*, and *thread local* as the "safe capabilities". Programs which only operate on a safe heap are guaranteed to be data-race free. When it is not important to distinguish between an object and its capability, we will say *e.g.,* "a local object" to mean an object with a local capability or a "safe object" etc.

The following code creates an immutable object with a field f with value v and a unary method m with body t: `object { use imm; def f = v; method m(x){t} }`. Fig. 1 shows the interaction of capabilities: arrows show all legal references from an object with one capability to another, modulo reflexivity.

The Dala capabilities form a hierarchy; imm < iso < local < unsafe. Objects can only refer to other objects with the same or lesser capabilities: an immutable object can only refer to other immutable objects; an isolate object can refer to other isos or imms, and a local object can refer to imms, isos, and other local objects. The model treats objects outside Dala as having a fourth unrestricted *unsafe* capability for uniformity. Using unsafe objects in a Dala program makes it susceptible to data races. Because it is the top capability, objects in the safe heap (imms, isos, locals) cannot refer to objects in the unsafe heap, while unsafe objects can refer to anything (see Fig. 2). The implication of this model is that once an object is safe, its entire reachable object graph is safe as well. *Thus: data races can only occur in unsafe objects.* By including unsafe objects in the model, we can describe the semantics of a program partially annotated with Dala capabilities.

Avoiding data-races by ensuring that two threads do not concurrently execute a code block that accesses the field f of some object o at the same time focuses on *code*. If there is another place in the program that also accesses o's f field which could be run at the same time, a data race could still happen. Making the *object* safe means that all *code* that interacts with o must follow the rules that make o safe from data races. If o is local, it cannot be shared across threads, so all accesses to its f field will come from the same thread. If o is immutable, all accesses to f will be read accesses which are benign. If o is isolated, two accesses to its f field by different threads require an explicit transfer from the first thread to the second thread (possibly via additional "stop-overs").

Dala capabilities are self-protecting in the sense that safety stems from a capability's own internal restrictions, not from restrictions elsewhere in the system. Writing to a field of an imm throws an error; so does aliasing an iso[6], or accessing a local object from outside of the thread it is local to. Imagine that objects' fields are private, and that field accesses implicitly go through a getter/setter indirection – in this case *all the checks necessary for an object to maintain its invariants are in its own internal code.* This is an important part of the design and key to adding unsafe capabilities in cases where these cannot be expected to be cooperative in avoiding data races. Fig. 3 shows how capabilities restrict certain effects in the system. Due to the absence of static types, these restrictions are enforced at run-time, meaning the cost of data-race freedom is (roughly) *per access.*

Dala's isolated objects are the key to efficient transfer of mutable state between different threads. (Immutable objects can be shared directly without causing problems, and thread local objects are permanently confined within their owning thread.) Isolates have only one unique incoming reference, and some extra care must be taken to preserve this uniqueness. Dala incorporates an explicit *consume* operation that destructively reads [40] the contents of a variable, and prevents those contents from being used again, similar to C++'s "move semantics". The contents of any mutable variable may be consumed but variables containing isolates *must* be consumed—otherwise the attempt to read the isolate will fail. Experiences by Gordon et al. [35] suggest explicit consume is to be favoured over implicit

The consume operator succeeds on all variables except the special `self` variable. This makes an isolated self variable effectively *borrowed* [4], meaning its value is tied to the current stack frame. Because the value of self cannot escape, expressions such as `x.m(y)` when `x` is an isolated object do not need to consume `x` and are guaranteed not to introduce any alias to `x`, other than through `self` on subsequent stack frames. With this borrowing-like behaviour, it is possible to traverse isolated structures without consuming them (a well-known problem when dealing with unique or linear values), but notably only using *internal* methods: `x.m` is allowed to borrow whereas `m(x)` is not. Since overcoming this limitation is well-known, we refrain from discussing this any further.

Dala guarantees that data races can only happen in places with unsafe objects, never in objects with safe capabilities (Theorem 5.5: *Data Race Freedom*). Developers can easily add capabilities to migrate from a "racy" program to a data race-free program with the certainty that this migration is semantics preserving, modulo permission and cast errors (Theorem 5.6: *Dynamic Gradual Guarantee*). These two properties are key in Dala and we will formally state them in §5.3.

3.1 Simple Case Study: A Concurrent Hash Map

To illustrate the simplicity of the Dala capabilities, consider the implementation of a simple concurrent hashmap in Fig. 4.

[6]This property can be implemented by ensuring that all code external to an iso accesses it via a proxy object with movement semantics, *e.g.,* by overloading the = operator. A simple implementation is possibly using linear proxies and allows unsafe objects to race on iso stack variables (as they already can on fields in Dala).

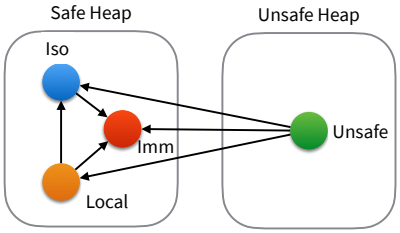

Figure 1. Dala Heap.

Object	Field contents Imm	Iso	Local	Unsafe
Immutable	•	×	×	×
Isolated	•	•	×	×
Local	•	•	•	×
Unsafe	•	•	•	•

Figure 2. Structural restrictions.

Object	Effects Read	Write	Alias	Transfer
Immutable	•	×	•	•
Isolated	•	•	×	•
Local	•	•	•	×
Unsafe	•	•	•	•

Figure 3. Capabilities and effects.

Assume that the hashmap is created inside a lightweight process, eventually calling the tail-recursive `run()` method with the channel `msgs`. The `run()` method reacts to input sent on the channel. It dispatches on the `op` field of messages received, and additionally expects the fields `key` and `val` to be present depending on operation.

A hashmap has five moving pieces: the hashmap object itself, the array of buckets, the entries in the buckets, and the keys and values of the entries. We examine the possible capabilities of these in turn to ensure thread-safety of the hashmap implementation.

For keys and values, there are two possibilities: iso and imm. In the former case, the keys and values can be transferred between the hashmap and its clients. In the latter case, they can be shared but also never change. It makes sense for keys to be immutable so that any client thread can know of their existence and ask for their associated value. We capture this in the code on Line 7 by calling the built-in `freeze()` operator that is the identity function on immutables, or creates an immutable copy otherwise (like in Ruby).

For values, iso and imm make sense under different circumstances. A sensible hashmap implementation should store only a single reference to its values, so the same code base should be reusable in both scenarios. The code in Fig. 4 support values which are both iso and imm. The key lines are 9, 11, 21, 34, and 40 which always *move* a value. Our field assignment makes use of "swap semantics" where the old value of the field is returned on an update. As a result, a get operation will move the value associated with a key out of the hashmap and remove the corresponding entry (Lines 20–21). (But see §4.3.1!)

From a thread-safety perspective, linked entries that constitute a bucket could be either immutable, iso, or local. Immutable entries would complicate the code when entries are unlinked (Line 20). Iso entries would slightly complicate searching through linked entries while maintaining uniqueness (lines 16–18 and 29–32). Thus, our entries are local (Line 37).

A similar thought process applies to the array of buckets. However, as our entries are local, we have no choice but to make the array local as well. For simplicity, the implementation of the array is elided above but we include Line 48 to explicitly show this choice. (As suggested by lines 15–16 and 27–29, our implementation assumes an empty bucket has a dummy entry for simplicity.)

Consequently, the hashmap itself must be local (Line 2). This ties the hashmap to the thread or lightweight process where it was created which does not seem unreasonable.

3.1.1 Gradual Transition to Dala Capabilities. Notice the bottom-up thinking when assigning the capabilities in the previous section. In a top-down approach, we might have decided to make the hashmap an iso to support its movement across threads or processes. This would have excluded local from the possible capabilities for the buckets array and its entries. When retrofitting existing code to use Dala capabilities, the bottom-up approach is superior to the top-down approach as it will incur the smallest possible change to the program. Assume the code in Fig. 4 was written as now, but without capabilities in mind: including no **`use`** declarations on Lines 2, 37 and 48, no consumes on Lines 34 and 40, and no `freeze()` on Line 39, etc. In this case, changing the keys to be immutable is simply adding the `freeze()` call on Line 39. All other objects in the hashmap can remain as-is. Similar, making values isos only needs the two **`consume`** operations, and changing `msg.val` to `msg.val = null` on lines 9 and 11.

Because of the structural constraints imposed on objects with a capability, top-down migration is likely to require bigger changes. For example, adding the **`use local`** on Line 2 will require adding a capability to the bucket array on Line 47–49. This will propagate to the entries and their keys and values.

A good regression test suite might be useful to help drive adding annotations regardless of the approach.

3.1.2 Constructing and Using the Hashmap. While orthogonal to the Dala capabilities, Fig. 5 illustrates how a program might construct a hashmap (lines 1–8) by spawning a new process with an associated channel that creates the hashmap and connects it to the channel.

When calling the constructor (Line 9), the caller gets a channel that can be used to send messages to the hashmap. It is easy to construct a proxy for a map that captures the channel used to communicate with the map, and uses a dedicated channel to get the reply (lines 11–24). The current `map_proxy` creates a new channel per interaction with the concurrent hashmap. This allows it to be immutable. If the

```
1  map = object { // hashmap
2    use local
3
4    method run(msgs) {
5      msg ←msgs
6      if (msg.op == "done") return "done"
7      k = msg.key.freeze(); c = msg.reply
8      if (msg.op == "get") c ←get(k)
9      if (msg.op == "put") c ←put(k, msg.val = null)
10     if (msg.op == "update") {
11       get(k); c ←put(k, msg.val = null); }
12     run(msgs)
13   }
14   method get(key) {
15     link = buckets.get(key.hash() % buckets.size)
16     while (link.next && link.next.key != key) {
17       link = link.next
18     }
19     if (link.next != null) {
20       target = link.next = link.next.next // unlink
21       return target.val = null // return result
22     } else {
23       return "Failure: No such key"
24     }
25   }
26   method put(key, val) {
27     link = buckets
28       .get(key.hash() % buckets.size)
29     while (link.next &&
30           link.next.key != key) {
31       link = link.next
32     }
33     if (link.next != null) {
34       return link.val = consume val
35     } else {
36       link.next = object { // link in bucket
37         use local
38
39         key = key
40         val = consume val
41         next = null
42       }
43       return "Success"
44     }
45   }
46
47   buckets = object { // array implementation
48     use local ...
49   }
50 }
```

Figure 4. Simple concurrent hashmap using the Dala capabilities. Note that `x.f = new` returns the old value of `x.f` and is used to move isos in and out of the heap.

creation of `map_ch` was moved outside of the object, a local proxy per client would make more sense. This change would be captured by changing Line 12 to **`use local`**. (Line 18 is just defensive programming.)

With the code in Fig. 5 in place, `map_proxy.put(k, v)` will asynchronously communicate with the concurrent hashmap even though it looks like a synchronous operation.

The message object on Lines 16–21 is the first example of creating an iso. Because isos can only hold other isos or immutables, `key` must be immutable. If it was local, it would err due to the structural constraints. If if was an iso, it would err due to the lack of a **`consume`**. As a consequence of the **`use`** `iso` on Line 17, there is thus a guarantee that any call to `map_proxy.put(k, v)` that would share mutable state across threads would throw an error.

3.1.3 Dala Properties and Concurrency Models.

Recall that the Dala capabilities are a set of rules for constructing objects (structural constraints) that guarantee that – with the exception of explicitly unsafe objects – programs are safe from data-races. Unless a reference to an unsafe object is passed in as an argument, a method inside a safe object cannot see an unsafe object. Furthermore, while unsafe objects can store references to safe objects, they cannot violate their properties. Assume for example that an unsafe object u is shared across multiple threads and that the safe object s is stored in a field f of u: $u.f = s$.

1. When s is an imm, it cannot be subject to data-races because it cannot be updated. If all field accesses are required to go via setters, this can be implemented by having all setters throw an error on use.
2. When s is an iso, it cannot be subject to data-races because iso's can only be dereferenced on the stack (*i.e.,* we allow $f.g$ but not $x.f.g$ when f contains an iso). Thus, any thread wanting to do $u.f.g$ must first transfer the contents of f to its local stack where it is unreachable from all other threads. This means that multiple threads can race on the f field of the unsafe object, which is not a race on s. The simplest implementation of iso's ensure that variables containing iso's are destructively read. (This however requires that unsafe objects cooperate in preserving the properties of isos.)
3. When s is a local, it cannot be subject to data-races because it can only be dereferenced by its creating thread. All other attempts to dereference throw an error. Passing the local around freely and using its identity is allowed, but this is not a data-race. Forbidding dereferences can be implemented by recording the identity of the creating thread and checking it against the identity of the current thread at the beginning of each method-/getter/setter.

```
1  method new_hashmap() {
2    return spawn (msgs) {
3      map = object {
4        // Code from Fig 4
5      }
6      map.run(msgs)
7    } // returns channel
8  }
9  map = new_hashmap()
10
11 map_proxy = object {
12   use imm
13
14   method put(key, val) {
15     map_ch = ... // channel
16     map ←object {
17       use iso
18       key = key
19       val = consume val
20       reply = map_ch
21     }
22     return ←map_ch
23   }
24 }
```

Figure 5. Constructing and using the hashmap in Fig. 4

(In addition to the checks mentioned above, all setters must check that the structural constraints are satisfied, informally: $OK(o.f = o')$ if $capability(o') \leq capability(o)$. Getters must also throw an error if used to retrieve iso's.)

In our concurrent hashmap example, we used an Erlang-esque model with lightweight processes and channels for communication between processes. Dala capabilities are not inherently tied to a specific concurrency model. For example, in a language with support for actors/active objects, the hashmap in Fig. 4 might have been an actor, and `run()` might be replaced by different methods called asynchronously. In this case, the process to which the bucket array and its entries are local would be to the implicit thread of the active object.

4 How the Dala Model Addresses the Concurrent Problems in §2

We now revisit the problems in § 2 and show how Dala addresses these problems.

4.1 Balancing Safety, Complexity and Performance

First, programs with only safe objects are safe from data races. Allowing parts of a program to be unsafe is useful for several reasons: transition to guaranteed safety can happen incrementally without a full-blown rewrite; there may be elements in a program's surroundings that are unsafe but that we still may need to access.

With Dala, we set out to deliver both simplicity and efficiency.[7] Dala has only three capabilities (not counting unsafe). Our design avoids complexities like capability subtyping, promotion and recovery. As will be exemplified in §4.3.1, dynamic checking enables flexible programs without relying on concepts like capability subtyping. Our capabilities are also orthogonal to concepts like deep copying, and far references (summary Table 1) and inter-thread communication can be efficient as Dala avoids deep copying.

4.2 Safe One-Size-Fits-All Concurrency

Adding the Dala capabilities to an unsafe language does not change its existing semantics or the fact that data races can (still) happen (in unsafe objects). Capabilities prevent the introduction of data races in the safe heap, empowering the objects with safe semantics.

While our formal model in §5 uses channels, this choice is "unimportant" and was driven by the desire to reduce the complexity of the formal model. (An earlier draft of this paper used actors, but this introduced unnecessary complexity.) The Dala model can be applied to other concurrent models, *e.g.*, actor- or lock-based concurrency models, and combinations.

For example, we could allow far references to local objects. With this design, a client of the hashmap in Fig. 4 could call `get` and `put` etc. directly. This works well with our design with keys are values being isolated or immutable. If Dala capabilities are used in an actor-based system, the story would be similar, and local objects would constitute an actor's private (mutable) state, and isolated objects enable efficient transfer of arguments in message sends. If locks are available, they could be useful for operating on unsafe objects, which might potentially be shared across threads.

4.3 Safety Begetting Unsafety

Dala introduces safe interoperation between unsafe and data race-free fragments. From the Data Race-Free Theorem (Theorem 5.5) and Progress and Preservation (Theorems 5.1 and 5.2), unsafe objects may be involved in data races but do not produce undefined (untrapped) behaviour. *More importantly, unsafe objects cannot create errors that are observable in the safe heap.*

An implementation of a racy double-buffering can use unsafe capabilities without compromising the safety of any safe capabilities. However, with the current rules, unsafe capabilities cannot *read* local capabilities, which may lead to the programmer wanting to propagate the unsafe capability annotation through the system. A slightly weaker version of our system would allow unsafe capabilities to *read* fields of local capabilities freely. Since local objects cannot contain unsafe objects, any data race due to this weakening is not visible to the local object, which voids the need for unsafe propagation.

4.3.1 Supporting both Immutable and Isolated Values in a Hashmap.

Because of the inherent flexibility of dynamic checking, it is simple to add an additional operation

[7] See §7 for a discussion on both these claims.

Programs	P	::=	t
Fields	F	::=	$f = w$
Methods	M	::=	**method** $m(x)\ \{t\}$
Terms	t	::=	$w \mid$ **let** $x\ =\ e$ **in** t
Expressions	e	::=	$w \mid x.f \mid x.f = w \mid x.m(w)$
	$\mid$	::=	$x \leftarrow w \mid\ \leftarrow x \mid$ **spawn** $(x)\ \{t\}$
	$\mid$	::=	$\blacksquare_i\ \iota \mid K$ **copy** $x \mid K$ **obj** $\{\overline{F}\ \overline{M}\}$
	$\mid$	::=	$(K)\ w \mid v$
Variables	w	::=	$x \mid$ **consume** x
Values	v	::=	$\iota \mid \varnothing \mid \top$
Capabilities	K	::=	**imm** | **local** | **iso** | **unsafe**

Figure 6. Syntax of Dalarna. m, f, and x are meta-variables representing method, fields, and variable names; metavariable x includes **self**. $\varnothing$ is a run-time value representing an empty channel.

that behaves like `get`, but returns an *alias* to the value in the hashmap of Fig. 4. This operation would throw an error if used on isos, but work fine on immutable objects. We can even implement it by extending the existing `get` method:

```
method get(key, move) {
  // additonal parameter controls movement semantics
  // lines 15–19
  if (move) { /* lines 20–21 */ }
  else { return link.next.val; /* create alias */ }
  // lines 22–24
}
```

This overcomes the problem pointed out previously forcing developers to choose one particular semantics or duplicate code, and does not compromise soundness.

5 Formalising the Dala Capability and Concurrency Models

To formally study the Dala model and clearly state and prove its key properties (*Data Race-Freedom* and *Dynamic Gradual Guarantee*), we formalise Dala in a minimal concurrent object-based language which we call Dalarna, which we describe in this section. In Dalarna, objects have the usual fields and methods, and there are no classes and no inheritance. Threads are created by a **spawn** operation which also sets up a channel for communication. Channels support both reference semantics and value semantics for objects depending on their capability meaning Dalarna is a shared-memory model. However, with the exception of unsafe objects, two threads sharing a common object o cannot implicitly transfer objects between each other by reading and writing fields in o. Hence, with the exception of unsafe objects, objects are effectively running in a message passing model, which may use shared memory under the hood for efficiency, without compromising safety.

Not modelling classes or inheritance is a choice driven by the desire to keep the system minimal. We note that dynamically typed languages are less dependent on inheritance, because of the non-need to establish nominal subtyping relations. Our simplifications allow us to focus on the most important aspects of our work. Handling permissions and capabilities in the presence of various forms of inheritance is well-known (*e.g.,* [12, 14, 46, 70, 75]) in a statically typed world including problems that may arise in an untyped setting. Therefore, we do not believe that these simplifications accidentally suppress any fundamental limitations of our approach.

For simplicity, channels themselves cannot be transferred. There is nothing fundamental about this simplification but undoing it requires some design thinking which is not important for the paper at hand, such as, whether or not we allow multiple threads connected to a single channel to race on taking the next message, etc.

Figure 6 shows the syntax of Dalarna. As is common, overbars (*e.g.,* $\overline{f}$) indicate possibly empty sequences (*e.g.,* $f_1, f_2, f_3, \ldots$). To simplify the presentation of the calculus programs are in A-normal form [33]: all subexpressions are named except for the **consume** expression. We further assume that programs use static single-assignment form [63], *i.e.,* the **let-in** term always introduces a new variable, field reads are assigned to variables before they are bound to other variables, etc, and that **self** is never aliased. None of these constraints are essential for the soundness of our approach.

A program (P) is a term (t). Terms are variables and let-bound expressions (*i.e.,* x and **let** $x\ =\ e$ **in** t). An unusual design choice borrowed from [19] is that assignment, $L = R$, binds the left-hand side L to the value of the right-hand side, R, and returns the *previous value* of L. (This is not uncommon when dealing with **iso** fields, and previous work in this area enforce it via an explicit swap operation [37–39]). Expressions are variables (w), destructive reads (**consume** x), field reads ($x.f$), field assignments ($x.f = w$), method calls ($x.m(w)$), a deep copy operation (K **copy** x) that returns a copy of an object graph with capability K, an object literal, a casting operation that asserts a capability ($(K)\ v$), and a spawn operation that creates a new thread. At run-time, the expression syntax also includes values. An object consists of fields ($\overline{f = x}$) and methods ($\overline{M}$) and are instantiated with a given capability K. For simplicity, methods have a single argument (*e.g.,* x) and more can be modelled using an object indirection using an unsafe object.

Spawning a new thread using **spawn** $(x)\ \{t\}$ introduces a new channel x both at the spawn-site, and inside the scope of the new thread whose initial term is t. The t is closed, *i.e.,* it cannot access variables declared from an outer scope. Channels are bidirectional unbuffered with blocking semantics on sending and receiving operations. The send operation $x \leftarrow w$ puts w on channel x, if necessary blocking until the

$$
\begin{array}{rcl}
E & ::= & \bullet \mid \texttt{let}\ x\ =\ E\ \texttt{in}\ t \mid x.f = E \mid E \leftarrow w \\
 & \mid & v \leftarrow E \mid \ \leftarrow E \mid K\ \texttt{obj}\ \{\overline{f = v}\ f' = E\ \overline{F}\ \overline{M}\} \\
 & \mid & x.m(E) \mid (K)\ E \\
H & ::= & \epsilon \mid H, \iota \mapsto K^i\ \texttt{obj}\ \{\overline{f = v}\ \overline{M}\} \mid H, x \mapsto v \\
 & \mid & H, \iota \mapsto \mathsf{chan}\ \{i, v\} \\
Cfg & ::= & H; \overline{T} \\
T & ::= & \epsilon \mid t^i \mid Err \\
Err & ::= & \mathrm{Err}_N \mid \mathrm{Err}_A \mid \mathrm{Err}_P \mid \mathrm{Err}_C
\end{array}
$$

Figure 7. Definitions of evaluation context, store and runtime configurations.

channel is "free" (*i.e.,* contains ∅). The sender then blocks ($\blacksquare_i\ \iota$) until the message i is received by the thread on the other end of the channel ι. The receive operation $\leftarrow y$ is similar to a send and blocks the current thread while the channel is "free". Values are locations (ι), the "absent value" $\top$ used to populate a consumed variable or field, and ∅, used at run time to indicate that a channel is empty.

For simplicity, we assume that programs do not attempt to consume **`self`**, which can be enforced through a simple syntactic check, and that all variable/method-parameter/channel names are distinct and none is called **`self`**. This is a common restriction in the literature, and key to avoiding breaking of abstraction [15].

5.1 Dynamic Semantics

We formalise the dynamic semantics of Dala as a small-step operational semantics with reduction-based, contextual rules for evaluation within threads (Fig. 7). The evaluation context E contains a hole $\bullet$ that denotes the location of the next reduction step [72]. We write the reduction step relation $H; \overline{T} \rightsquigarrow H'; \overline{T'}$ which takes a reduction step from heap H and a collection of threads in $\overline{T}$, to a new heap H' and a new thread state $\overline{T'}$. A store H is either empty (ϵ), or contains mappings from variables to values, and from locations to objects and channels (Fig. 7). The superscript i in K^i represents the object's thread owner and we used it to keep track of ownership of **`local`** objects (omitted from the rules when not relevant).

A configuration *Cfg* is a heap H and a collection of concurrently executing threads $\overline{T}$. A thread is either finished (ϵ), a term t^i (where i represents the thread owner id, omitted when not necessary), or a run-time error (*Err*). There are four kinds of run-time errors: consumption errors (Err_A, which occur when a program accesses a consumed value); permission errors (Err_P, which occur when a program violates the structural constraints imposed by its capabilities); cast errors (Err_C, which occur when a program has a different capability than the one casted to); and normal errors (Err_N), such as accessing a non-existing field, calling a non-existing method, etc. The execution of threads is concurrent and non-deterministic. The non-determinism comes from C-Eval and the commutativity equivalence rule.

$$
\frac{\text{(C-Eval)}\quad H; t \rightsquigarrow H'; \overline{T'}}{H; t\ \overline{T} \rightsquigarrow H'; \overline{T'}\ \overline{T}}
\qquad
\begin{array}{c}
H; v \equiv H;\ \epsilon \\
\overline{T}\ Err \equiv Err \\
\overline{T}\ \overline{T'} \equiv \overline{T'}\ \overline{T} \\
\overline{F}\ \ f = v \equiv f = v\ \ \overline{F} \\
\overline{M}\ M \equiv M\ \overline{M}
\end{array}
$$

The reduction of a program t begins in an initial configuration with an empty heap $\epsilon; t$ (Definition B.1, [32]). In the remainder of this section, we go through the reduction rules, ending with a discussion of the error trapping rules that dynamically trap actions which (might) lead to data races. For capabilities, the following relations hold: **`unsafe`** ≤ **`local`**, **`local`** ≤ **`iso`**, and **`iso`** ≤ **`imm`**. The ≤ relation is reflexive and transitive. $\mathsf{isImm}(H, \iota)$ (etc. for other capabilities) holds if $H(\iota) = K\ \texttt{obj}\ \{\_\ \_\}$ and K = **`imm`**.

The reduction of the **`let`** term updates the heap with a stack variable x pointing to the value v (R-Let). Reading a variable with a non-isolated object reduces to a location (R-Var); reading an isolated object involves moving semantics: moving the contents of a variable using **`consume`** reduces to a location, and leaves a $\top$ token in the variable which will cause an error if accessed before overwritten (R-Consume). Consuming fields is not allowed. Instead, one consumes a field when doing an assignment, *e.g.,* **`let`** $x\ =\ (y.f = z)$ **`in`** `...`, places on `x` the object pointed by `y.f` and places `z` in `y.f`. Reading a field ($x.f$) reduces to the value stored in the field (R-Field). Note that to read an isolated object's field one must update the field and place another object in its stead — directly reading an isolated field would create a new alias. The helper predicate $\mathsf{localOwner}(\ldots)$ checks that if the target object is **`local`**, then the current thread is its owner. (This prevents threads to access unowned local objects.)

Updating a field ($x.f$) with a value (v) reduces to returning the previously held value in the field and updating the field f to point to value v. There is a check that prevents mutating immutable objects, $\mathsf{OkRef}(H, K, v)$ ensures that the object v can be placed under an object with capability K, and the remaining helper predicates check that if the target object (x) is local, then its owner is the current thread and if the source (v) is local then its owner is the current thread.[8] (Other reduction rules repeat this local ownership check and for space reasons we omit mentions in the remaining rules.)

An object literal (R-New) checks that values of its fields do not violate the structural constraints imposed by its capability K, and if K is **`local`** then the current thread must own the local fields. R-New returns the (fresh) location of the object.

Calling a method on an object referenced by x and with argument v reduces to the body t of the method with **`self`**

[8] This design is good for JIT compilation since local objects have guarantees to not point to unowned local objects (Corollary 5.3), thus removing some unnecessary dynamic checks from possible implementations.

$$\frac{x \notin dom(H)}{H; \mathbf{let}\ x\ =\ v\ \mathbf{in}\ t \rightsquigarrow H, x \mapsto v; t}\ \text{(R-Let)}$$

$$\frac{H(x) = \iota \quad \neg\mathsf{isIso}(H, \iota)}{H; E[x] \rightsquigarrow H; E[\iota]}\ \text{(R-Var)}$$

$$\frac{H(x) = \iota \quad H' = H[x \mapsto \top]}{H; E[\mathbf{consume}\ x] \rightsquigarrow H'; E[\iota]}\ \text{(R-Consume)}$$

$$\frac{H(x) = \iota \quad H(\iota) = \_\ \mathbf{obj}\ \{\_\ f = v\ \overline{M}\} \quad \neg\mathsf{isIso}(H, v) \quad \mathsf{localOwner}(H, i, \iota)}{H; E[x.f]^i \rightsquigarrow H; E[v]^i}\ \text{(R-Field)}$$

$$\frac{H(x) = \iota \quad H(\iota) = K\ \mathbf{obj}\ \{\overline{f = v}\ f = v'\ \overline{M}\} \quad \neg\mathsf{isImm}(H, \iota) \quad \mathsf{OkRef}(H, K, v) \quad \mathsf{isLocal}(H, \iota) \Rightarrow (\mathsf{isOwner}(H, i, \iota) \wedge \mathsf{localOwner}(H, i, v))}{H; E[x.f = v]^i \rightsquigarrow H[\iota \mapsto K\ \mathbf{obj}\ \{\overline{f = v}\ f = v\ \overline{M}\}]; E[v']^i}\ \text{(R-FieldAssign)}$$

$$\frac{\forall f = v \in \overline{f = v}.\ \mathsf{OkRef}(H, K, v) \wedge (K = \mathbf{local} \wedge \mathsf{isLocal}(H, v)) \Rightarrow \mathsf{isOwner}(H, i, v) \quad \iota\ fresh \quad H' = H, \iota \mapsto K^i\ \mathbf{obj}\ \{\overline{f = v}\ \overline{M}\}}{H; E[K\ \mathbf{obj}\ \{\overline{f = v}\ \overline{M}\}]^i \rightsquigarrow H'; E[\iota]^i}\ \text{(R-New)}$$

$$\frac{x', y'\ fresh \quad H(x) = \iota \quad H(\iota) = \_\ \mathbf{obj}\ \{\_\ \overline{M}\ \mathbf{method}\ m(y)\ \{t\}\}}{H; E[x.m(v)] \rightsquigarrow H, x' \mapsto \iota, y' \mapsto v; E[t[\mathbf{self} = x'][y = y']]}\ \text{(R-Call)}$$

$$\frac{H(\iota) = K\ \mathbf{obj}\ \{\_\_\}}{H; E[(K)\ \iota] \rightsquigarrow H; E[\iota]}\ \text{(R-CastLoc)}$$

$$\frac{\iota, i, j\ fresh \quad x \notin dom(H) \quad H' = H, x \mapsto \iota, \iota \mapsto \mathsf{chan}\ \{i, \varnothing\}}{H; E[\mathbf{spawn}\ (x)\ \{t\}] \rightsquigarrow H'; E[\iota]\ t^j}\ \text{(R-Spawn)}$$

$$\frac{H(\iota) = \mathsf{chan}\ \{i, \iota'\} \quad H' = H[\iota \mapsto \mathsf{chan}\ \{i, \varnothing\}]}{H; E[\leftarrow \iota] \rightsquigarrow H'; E[\iota']}\ \text{(R-Recv)}$$

$$\frac{H(\iota) = \mathsf{chan}\ \{\_, \varnothing\} \quad i\ fresh \quad H' = H[\iota \mapsto \mathsf{chan}\ \{i, v\}]}{H; E[\iota \leftarrow v] \rightsquigarrow H'; E[\blacksquare_i\ \iota]}\ \text{(R-SendBlock)}$$

$$\frac{H(\iota) = \mathsf{chan}\ \{i', v\} \quad v = \varnothing \vee i \neq i'}{H; E[\blacksquare_i\ \iota] \rightsquigarrow H; E[\iota]}\ \text{(R-SendUnblock)}$$

$$\frac{\mathbf{iso} \neq K \quad H(x) = \iota' \quad \mathsf{localOwner}(H, i, \iota') \quad \mathsf{OkDup}(H, K, H(x)) = (H', \iota)}{H; E[K\ \mathbf{copy}\ x]^i \rightsquigarrow H'; E[\iota]^i}\ \text{(R-Copy)}$$

Figure 8. Runtime semantics

substituted for a fresh variable bound to the location of x and the singular argument substituted for a fresh variable bound to v (R-Call).

A new thread is introduced by a **spawn** operator which introduces a new channel connecting the spawned thread with its "parent" (R-Spawn). Rules (R-SendBlock), (R-SendUnblock) and (R-Recv) handle sending and receiving values on a channel. Sending on a channel ι blocks until the channel is empty, and subsequently blocks the sending thread until the value has been received on the other side. Reading on a channel blocks until there is a value that can be retrieved.

Casting an object (E-CastLoc) checks that the object has the specified capability, throwing a permission error, otherwise. The function R-Copy deep copies the object pointed by ι, returning a heap that contains the copy of the object graph with capability K and a fresh location that points to the object copied.[9] The helper functions used above are defined thus:

$$\frac{H(\iota) = K'\ \mathbf{obj}\ \{\_\_\} \quad K \leq K'}{\mathsf{OkRef}(H, K, \iota)}\ \text{(RefCheck)}$$

$$\frac{\mathsf{isLocal}(H, v) \Rightarrow \mathsf{isOwner}(H, i, v)}{\mathsf{localOwner}(H, i, v)}\ \text{(Helper-LocalOwner)}$$

For simplicity, we have gathered some rules that trap capability errors at run-time in Fig. 9. Common errors when accessing non-existent fields and methods throw a Err_N error (*e.g.,* E-NoSuchField). Accessing values which are absent due to a destructive read yields a Err_A (*e.g.,* E-Consume). Assigning an illegal value to a field is not allowed (*e.g.,* E-AliasIso and E-IsoField). Casts to the wrong capability reduce to Err_C. (Remaining rules in Appendix A in [32].)

[9]The helper function $\mathsf{OkDup}(H, K, v)$ is a standard deep-copying operation. [16, 50]

5.2 Well-Formedness

We define the environment (also used as store typing [60]) as $\Gamma ::= \epsilon \mid \Gamma, x : K \mid \Gamma, \iota : K$, where ϵ represents the empty environment, and $x : K$ and $\iota : K$ mean variable x and location ι have capability K. Well-formedness guarantee that the heap is well-formed *w.r.t.* object capability (and its fields) and that variables are not duplicated when introduced, but they do not statically forbid violation of object capabilities (which will throw a permission run-time error). Objects' thread-locality and proper isolation (for objects with local and isolated capabilities respectively) fall out of well-formedness and appear in [32].

5.3 Properties of Well-formed Programs

We highlight the properties satisfied by well-formed programs (proofs in [32]):

- Progress and Preservation (Theorems 5.1 and 5.2). This means that if a well-formed program is not finished (empty state), is not an error (normal, absent, permission, or cast error), or is not in a deadlock state (terminal configuration), then it can take a reduction step until it ends in a terminal configuration state and the result of each reduction step is well-formed.
- Data-Race Freedom (Theorem 5.5). Programs without unsafe objects are data-race free by construction.
- Dynamic Gradual Guarantee (Theorem 5.6). Adaptation of the gradual guarantee [67, 68] stating capabilities do not affect the run-time semantics, modulo casts and capability errors. Essentially, if an unsafe program is well-formed and takes a reduction step, the same program with capability annotations either reduces to

$$\frac{H(x.f) = \bot}{H; E[x.f] \rightsquigarrow H; \mathrm{Err}_N} \text{(E-NoSuchField)} \qquad \frac{H(x) = \top}{H; E[\textbf{consume}\ x] \rightsquigarrow H; \mathrm{Err}_A} \text{(E-Consume)}$$

$$\frac{H(x) = \iota \quad \mathsf{isIso}(H, \iota)}{H; E[x] \rightsquigarrow H; \mathrm{Err}_P} \text{(E-AliasIso)} \qquad \frac{H(x) = \iota' \quad H(x.f) = \iota \quad \mathsf{localOwner}(H, i, \iota') \quad \mathsf{isIso}(H, \iota)}{H; E[x.f]^i \rightsquigarrow H; \mathrm{Err}_P} \text{(E-IsoField)}$$

$$\frac{H(\iota) = K'\ \textbf{obj}\ \{\_\_\} \quad K' \neq K}{H; E[(K)\ \iota] \rightsquigarrow H; \mathrm{Err}_C} \text{(E-CastError)}$$

Figure 9. Expression rules producing errors. To reduce clutter, we write $H(x.f) = v$ when $H(x) = \iota \wedge H(\iota) = \_\ \textbf{obj}\ \{F\ \_\}$ and $f = v \in F$. (Remaining rules in Appendix A in [32].)

the same run-time configuration (modulo safe erasure, Definition B.11 in [32]) or throws an error due to a cast or capability violation.

Programs start in an initial well-formed configuration $\epsilon; t$ (Definition B.1 in [32], and reduce to new configurations. Progress (Theorem 5.1) guarantees that a well-formed configuration reduces to a new configuration, or it is terminal (Definition B2 in [32]). From Preservation (Theorem 5.2), a reduction step always leads to a well-formed configuration. Terminal configurations are either finished programs, errors (with $\overline{T}\ \mathit{Err} \equiv \mathit{Err}$ from equivalence rules on Page 10), or a deadlock configuration (Definition B.3 in [32]). A deadlock configuration happens when all threads are either waiting on a receive or on a send operation.

Theorem 5.1 (Progress)**.** *A well-formed configuration* $\Gamma \vdash H; \overline{T}$ *is either a terminal configuration or* $H; \overline{T} \rightsquigarrow H'; \overline{T'}$.

Theorem 5.2 (Preservation)**.** *If* $\Gamma \vdash H; t\ \overline{T}$ *is a well-formed configuration, and* $H; t\ \overline{T} \rightsquigarrow H'; \overline{T'}\ \overline{T}$ *then, there exists a* Γ' *s.t.* $\Gamma' \supseteq \Gamma$ *and* $\Gamma' \vdash H'; \overline{T'}\ \overline{T}$

Corollary 5.3 (Thread-Affinity of Thread-Local Fields)**.** *Implied by Preservation, a thread local object with owner i cannot contain a thread local object with owner j, where* $i \neq j$*. The only way a local object can reference another local object of a different owner is via field assignment (R-FieldAssign). But R-FieldAssign checks that target and source share owners. Thus, thread local objects can only reference thread local objects of the same owner.*

Definition 5.4 (Data Race)**.** Informally, a data race is defined as two threads accessing (write-write or read-write) the same field without any interleaving synchronisation. In our setting, this translates to two accesses to the same field of an object o in threads i and j without an interleaving explicit transfer of o from i to j. Thus, a data race in Dalarna requires the ability of a mutable object to be referenced from two threads at the same time (formal definition in [32]).

Theorem 5.5 states that Dalarna is data race-free modulo unsafe objects. Objects that use safe capabilities cannot introduce a data race; unsafe objects may introduce data races.

Theorem 5.5 (Dalarna is Data-Race Free Modulo Unsafe Objects)**.** *All data races directly or indirectly involve an unsafe object. A data race is defined as a read/write or write/write access to an object by different threads without interleaving synchronisation (In our formalism, this means an interleaving transfer of an object to another thread; formal proof in [32]).*

We now proceed to sketch the proof of data-race freedom for safe objects by showing that it is not possible for a mutable safe object to be aliased from two different threads at the same time. We refer to l, i, j, r_1 and r_2 from Definition 5.4 for clarity.

Let us examine the implications of l having any of the three safe capabilities (and ignore unsafe objects for now).

1. l *is immutable.* By E-BadFieldAssign, attempts to write fields of an immutable object will err. Thus, if l is immutable, then R will contain an error, which it did not by assumption.
2. l *is local.* By R-Field and R-FieldAssign, attempts to write a field of a local from outside of its creating thread will err. Thus, if l is local, then R will contain an error (because $i \neq j$), which it did not by assumption.
3. l *is isolated.* Fields of isolated objects can be read or written freely. Thus, we must show that l can be accessible in two threads at the same time. In the initial heap, no objects exist that is shared across threads and our only way to share objects across threads is by sending them on a channel. An isolated l can be sent on a channel. However, this requires that l is consumed, meaning it will no longer be accessible by the sender. To avoid the consumption, we could store l in a field of an object, and then transfer the object. Such objects, would have to be immutable or local. However, by R-FieldAssign, immutable or local objects cannot contain isolated objects.
   Thus, we cannot create a situation where l is in both r_1 and r_2 without any interleaving send.

A program that uses unsafe objects may use these to store local and isolated objects. Thus, an unsafe object u aliased across threads could store a local or isolated l in $u.f$ (see 3. above). This would allow threads i and j to do $u.f.g$. If l is local, unless $i = j$, at least one of the accesses will err (or both if neither i nor j is the creating thread of l, see 2. above). If l is isolated, $u.f.g$ will err by E-IsoField. Thus, even in the presence of unsafe objects, safe objects will not participate in data races.

Progress and preservation guarantee the absence of untrapped errors and Theorem 5.5 shows that all data races can be blamed on unsafe objects. Next, we show that capability annotations do not affect the run-time semantics, modulo cast

or capability violations which trap operations that if allowed could lead to a data race. We show this in the Dynamic Gradual Guarantee theorem (Theorem 5.6, adapted from [68]). We unpack this theorem before stating it formally. Let P be a well-formed program, and S its "stripped equivalent", where all safe capabilities have been erased (and thus replaced with unsafe). Below, "reduces" denotes a single reduction step.

1. The well-formedness of S follows from the well-formedness of P, as an unsafe object can reference any object (Definition B.11 in [32]).
2. If P reduces to a non-error configuration, S reduces to the same configuration and the same heap (modulo heap capability erasure); in case P throws an error caused by absent values or normal errors, S will throw the same error. The two evaluations only diverge if P throws a permission error in a dynamic check – this will never happen in S because it does not have any safe capabilities. (See Lemma B.27 in [32] *e.g.,* E-BadFieldAssign or E-BadInstantiation among others.)
3. If S reduces to an error configuration, P will reduce to the same error configuration. If S reduces to a non-error configuration, P will either reduce to the same configuration (modulo capability stripping), or throw a capability or permission error.

Theorem 5.6 (Dynamic Gradual Guarantee)**.** *Let $H; t\ \overline{T_0}$ be a configuration and Γ a store type such that $\Gamma \vdash H; t\ \overline{T_0}$. Let ^{e} be a function that replaces safe capabilities with unsafe in heaps, terms, etc (Definition B.11 in [32]). Then:*

1. $\Gamma^e \vdash (H; t\ \overline{T_0})^e$.
2. a. *If* $H; t\ \overline{T_0} \rightsquigarrow H'; \overline{T_1}\ \overline{T_0}$ *and* $\overline{T_1} \neq Err$ *then,* $(H; t\ \overline{T_0})^e \rightsquigarrow (H'; \overline{T_1}\ \overline{T_0})^e$.
   b. *If* $H; t\ \overline{T_0} \rightsquigarrow H'; \overline{T_1}\ \overline{T_0}$ *and* $\overline{T_1} = Err_A \vee Err_N$ *then,* $(H; t\ \overline{T})^e \rightsquigarrow H''; \overline{T_1}\ \overline{T_0}$
3. a. *If* $(H; t\ \overline{T_0})^e \rightsquigarrow (H'; Err\ \overline{T_0})^e$ *then,* $H; t\ \overline{T_0} \rightsquigarrow H'; Err\ \overline{T_0}$
   b. *If* $(H; t\ \overline{T_0})^e \rightsquigarrow (H'; \overline{T}\ \overline{T_0})^e$ *and* $\overline{T} \neq Err$ *then,* $H; t\ \overline{T_0} \rightsquigarrow H'; \overline{T'}\ \overline{T_0}$ *and* $\overline{T'} = Err_P \vee Err_C \vee \overline{T}$.

The Dynamic Gradual Guarantee (Theorem 5.6) uses a single step reduction to guarantee that the capabilities are semantics preserving, modulo permission and cast errors. We extend the Dynamic Gradual Guarantee to account for multi-step reductions, starting from an initial configuration until reaching a terminal configuration, *i.e.,* $\epsilon; P \rightsquigarrow^* H; C$. To remove non-determinism of program reductions, we define the trace of a program as a list of pairs that contain the reduction step and the thread id on which the reduction happens. We extend the reduction relation to account for the trace, named the replay reduction relation, which is the standard reduction relation except that it deterministically applies the expected reduction step on the expected thread id ([32], Definitions B.5 to B.7 and Theorem B.28). The basic idea is to reduce a safe program to a terminal configuration, which produces a trace. We use this trace to replay the reductions on the capability stripped (unsafe) program (and *vice versa*), showing two programs reduce to the same terminal configuration modulo cast errors and permission errors. Since it obscures some cases where the identical step is taken, we show the single-step theorem in the paper which highlights these cases.

6 Related Work

The Dala capability model (immutables < isolates < thread-locals) carefully selects a number of well-known concepts from the literature [16, 19, 40, 48, 55, 73]. Dala's key contribution here is the careful combination: what we have left out is at least as important as the features we have included. Given that an actor is essentially a thread plus thread-local storage [25], our distinctions are similar to many object-actor hybrid systems [22, 27, 36, 44, 45, 65], although, crucially, we follow Singularity [47] by incorporating isolates for fast transfers. Similarly, there are many more flexible models for distinguishing between read-only and read-write objects: we adopt "deep immutability" for its clear conceptual model Glacier [21]. While there are certainly more complicated models of permissions and capabilities for data-race freedom (*e.g.,* [12, 16, 19, 75] and many others), we consider our chosen set of concepts a "sweet spot" in the balance between expressivity and complexity.

6.1 Capabilities and Ownership

Dala is also heavily influenced by static capability-based programming languages [8, 12, 18, 19, 35]. Capability-based programming languages require all programs to be fully annotated with capabilities, and these annotations guarantee data-race freedom, with erasure semantics. In contrast, our approach begins with a dynamic language that allows data race and data race-free programs to interact, and we maintain data race-freedom for programs with safe capabilities.

Sergey and Clarke [66] add gradual ownership to a static language, introducing notions of parametric ownership and inserting casts when needed; they prove soundness and common ownership invariants. Our work has similarities in that isolated objects can be seen as owners-as-dominators, and our local objects have threads as owners. Dala differs in that it is a dynamic, concurrent language, and we prove common invariants and data race-freedom for safe objects.

HJp [70] enforces safe sharing of objects using a permission-based gradual typing, which inserts run-time checks when necessary. Objects are either in shared-read permission, which allows reading from multiple threads but no mutation, or read-write permission, allows any mutation and aliasing but no sharing. It also introduces storable permissions which allows a permission to refer to a tree of objects. Our approach uses capabilities at run-time; `imm` capabilities are similar to

HJp shared-read and read-write permissions are similar to `local`. In addition, Dala also has the concept of `iso` which can move across threads but do not allow any aliasing.

Roberts *et al.* [61] (and recently [56]) showed that run-time gradual type checks could have minimal or no performance impact on a suitable virtual machine, despite what is naively much extra checking for partially-typed programs. We chose to extend their Moth virtual machine for our dynamically-checked implementation to take advantage of their work.

6.2 Capabilities In The Wild

Castegren's *et al* work [12] seems to be the first one where reference capabilities are orthogonal to the concurrency model *i.e.,* the reference capabilities seem applicable to multiple concurrency models. Their capabilities have been formalised using fork-join style but their implementation uses active objects [8]. The implementation of Gordon's *et al* reference immutability work closely follows the formal semantics; because their reference permissions apply transitively to fields of the object, it is not clear whether the model is general enough to be applicable to different concurrency models, *e.g.,* actor or tuplespace model [24, 34]. Boyland et al's work [5] can encode 8 object capabilities to express different invariants and they argue that these can be used in concurrent programs with no run-time cost, when programs are fully typed. We believe the capability system is expressive enough to work on different concurrency models, but it is not clear whether their capabilities enforce data race freedom. In languages such as E [54], AmbientTalk [22], and Newspeak [6] references (far and near) represent object capabilities [28] and use a *vat*-based concurrency model. In contrast, our work is simple and uses 3 capabilities, allows interoperation between safe and "racy" programs and (as far as we know) it is the first one to use reference capabilities in a dynamic language where the capabilities are orthogonal to the concurrency model.

6.3 Race Detectors

Although our capability checks guarantee data-race freedom, they are different to the checks that a data-race detector might employ [64]. These checks are also in some sense "eager" or may cause false positives. For example, a program that effectively transfers a mutable object o between two threads will execute without errors if o is isolated, but not if it is local and the non-owning thread dereferences it. This is a somewhat pragmatic choice, but guided by our desire to make our capabilities a tool for programmers to capture their *intent*. Thus, we expect that a local object is explicitly demarcated local (at creation time) and not isolated for a reason. Thus, Dala helps programmers state their intentions and check that the programs they write conform to said intentions. This is different from a data race detector which may only fire if a data race occurs (which may happen on some runs but not others of the same program).

7 Discussion and Future Work

Our claim of efficiency rests on absence of deep copying and turning local accesses into asynchronous operations. That said, our capabilities incur a cost on (most) accesses to objects – *e.g.,* on writes to fields, etc. To remove most of this cost will require self-optimising run-times [71] and techniques similar to Grace's transient checks [61] to reduce the number of checks needed to satisfy the capability invariants.

Adding gradual capabilities at the type level will allow most checks to be removed [56, 61], but more importantly help programmers document the behaviour of code. Notably this addition will not need escape hatches due to inflexible types as programmers can fall-back to dynamic checks which are equally safe.

In this paper, the Dala capabilities only constrain heap structures. Nothing prevents a stack variable in an immutable object to point to an unsafe object. To reason about data-race freedom of a method call on a safe object, we need to consider the methods arguments' capabilities. Extending the structural constraints to stack variables is an interesting point in the design space: if immutable objects can only "see" other immutable objects, method calls on immutables are guaranteed to be side-effect free modulo allocation and GC. For isos, side-effects are not possible, but preexisting objects may be updated in place. Finally, local objects would only observe local objects belonging to the same thread (which is probably very desirable), and permit side-effects visible in the current thread only. Such a design can reduce the number of checks (*e.g.,* all checks of thread-ownership happen only when calling a local method in an unsafe context).

The Dala design is both simple and simplistic. Additional expressivity might be gained for example by adding a notion of ownership, or borrowing. How to compare complexity of different capability systems is not clear. For example, let us briefly compare the number of rules and concepts in formalisms (on purpose in a footnote). Dala: 32 run-time rules (omitting helper predicates); 30 well-formed rules. Total: 62 rules. Encore's type system [12]: 73 rules for well-formed declaration and configurations, environment, type equivalence and expression typing; 40 reduction rules. Total: 113 rules. Pony [51]: 3 Table/Matrix with viewpoint-adaptation matrix, safe-to-write and capability constraints; 7 definitions for Restricted syntaxes of types and bounds; 16 reduction rules; 10 typing rules; 13 rules for reduction of types and bounds with a partial reification; 38 rules for inheritance, nominal, structural and bound inheritance, capability and reified subtyping, bound compliance, sub-bound compliance, method subtyping; 23 rules for safe-to-write, sendable types, reduction of types and translation of expressions Total: 110 rules.

8 Conclusion

A data race is a fundamental, low-level aspect of a program which is not tied to the intended semantics of a particular application. While many race conditions stem from data-races, data-race freedom does not mean absence of race conditions. Data-race freedom is however still important: removing them removes many race conditions and moreover makes a program's semantics independent on the idiosyncrasies of a particular (weak) memory model. In the case of programming languages like C and C++, data-races are examples undefined behaviour. The Dala capabilities guarantee absence of data races in safe objects by imposing restrictions on all code that interacts with these objects. Safe and unsafe objects can co-exist and the presence of the latter does not compromise the guarantees of the former.

Dala helps programmers structure their programs with capabilities including immutable, isolated, and thread-local. We support the Dala design with a formal model, clear and proven properties *w.r.t.* data-race freedom and semantics preservation when capabilities are added to a program.

We provide Daddala (Section 6 in [32]), an early proof-of-concept prototype implementation which is available as open source.[10] Based on this last experience, we believe that our model can provide opt-in data-race safety to programmers on top of existing languages, with relatively little implementation difficulty and overhead.

References

[1] [n.d.]. Akka Documentation. Actor Best Practices. https://doc.akka.io/docs/akka/current/general/actor-systems.html#actor-best-practices. Accessed August 26, 2020.

[2] Joe Armstrong, Robert Virding, and Mike Williams. 1993. *Concurrent programming in ERLANG.* Prentice Hall.

[3] Andrew P. Black, Kim B. Bruce, Michael Homer, and James Noble. 2012. Grace: the absence of (inessential) difficulty. In *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2012, part of SPLASH '12, Tucson, AZ, USA, October 21-26, 2012*, Gary T. Leavens and Jonathan Edwards (Eds.). ACM, 85–98. https://doi.org/10.1145/2384592.2384601

[4] John Boyland. 2001. Alias burying: Unique variables without destructive reads. *Softw. Pract. Exp.* 31, 6 (2001), 533–553. https://doi.org/10.1002/spe.370

[5] John Boyland, James Noble, and William Retert. 2001. Capabilities for Sharing: A Generalisation of Uniqueness and Read-Only. In *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2072)*, Jørgen Lindskov Knudsen (Ed.). Springer, 2–27. https://doi.org/10.1007/3-540-45337-7_2

[6] Gilad Bracha. 2017. *Newspeak programming language draft specification version 0.1.* Technical Report. Technical report, Ministry of Truth.

[7] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. 2010. Modules as Objects in Newspeak, See [29], 405–428. https://doi.org/10.1007/978-3-642-14107-2_20

[8] Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Einar Broch Johnsen, Ka I Pun, Silvia Lizeth Tapia Tarifa, Tobias Wrigstad, and Albert Mingkun Yang. 2015. Parallel Objects for Multicores: A Glimpse at the Parallel Language Encore. In *Formal Methods for Multicore Programming - 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015, Bertinoro, Italy, June 15-19, 2015, Advanced Lectures (Lecture Notes in Computer Science, Vol. 9104)*, Marco Bernardo and Einar Broch Johnsen (Eds.). Springer, 1–56. https://doi.org/10.1007/978-3-319-18941-3_1

[9] Denis Caromel, Christian Delbé, Alexandre Costanzo, and Mario Leyton. 2006. ProActive: an Integrated platform for programming and running applications on Grids and P2P systems. *Computational Methods in Science and Technology* 12 (01 2006). https://doi.org/10.12921/cmst.2006.12.01.69-77

[10] Denis Caromel, Ludovic Henrio, and Bernard P. Serpette. 2009. Asynchronous sequential processes. *Inf. Comput.* 207, 4 (2009), 459–495. https://doi.org/10.1016/j.ic.2008.12.004

[11] Elias Castegren, Joel Wallin, and Tobias Wrigstad. 2018. Bestow and atomic: Concurrent programming using isolation, delegation and grouping. *Journal of Logical and Algebraic Methods in Programming* 100 (2018), 130 – 151. https://doi.org/10.1016/j.jlamp.2018.06.007

[12] Elias Castegren and Tobias Wrigstad. 2016. Reference Capabilities for Concurrency Control. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy (LIPIcs, Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 5:1–5:26. https://doi.org/10.4230/LIPIcs.ECOOP.2016.5

[13] Dominik Charousset, Raphael Hiesgen, and Thomas C. Schmidt. 2014. CAF - the C++ Actor Framework for Scalable and Resource-Efficient Applications. In *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control, AGERE! 2014, Portland, OR, USA, October 20, 2014*, Elisa Gonzalez Boix, Philipp Haller, Alessandro Ricci, and Carlos Varela (Eds.). ACM, 15–28. https://doi.org/10.1145/2687357.2687363

[14] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. 2013. Ownership Types: A Survey. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). Lecture Notes in Computer Science, Vol. 7850. Springer, 15–58. https://doi.org/10.1007/978-3-642-36946-9_3

[15] Dave Clarke and Tobias Wrigstad. 2003. External Uniqueness Is Unique Enough. In *ECOOP 2003 - Object-Oriented Programming, 17th European Conference, Darmstadt, Germany, July 21-25, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2743)*, Luca Cardelli (Ed.). Springer, 176–200. https://doi.org/10.1007/978-3-540-45070-2_9

[16] Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Broch Johnsen. 2008. Minimal Ownership for Active Objects. In *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 5356)*, G. Ramalingam (Ed.). Springer, 139–154. https://doi.org/10.1007/978-3-540-89330-1_11

[17] David G. Clarke and Sophia Drossopoulou. 2002. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002, Seattle, Washington, USA, November 4-8, 2002*, Mamdouh Ibrahim and Satoshi Matsuoka (Eds.). ACM, 292–310. https://doi.org/10.1145/582419.582447

[18] Sylvan Clebsch and Sophia Drossopoulou. 2013. Fully concurrent garbage collection of actors on many-core machines. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). ACM, 553–570. https://doi.org/10.1145/2509136.2509557

[19] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE! 2015, Pittsburgh, PA, USA, October*

[10] https://github.com/gracelang/moth-SOMns/tree/daddala

26, 2015, Elisa Gonzalez Boix, Philipp Haller, Alessandro Ricci, and Carlos Varela (Eds.). ACM, 1–12. https://doi.org/10.1145/2824815.2824816

[20] Sylvan Clebsch, Juliana Franco, Sophia Drossopoulou, Albert Mingkun Yang, Tobias Wrigstad, and Jan Vitek. 2017. Orca: GC and type system co-design for actor languages. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 72:1–72:28. https://doi.org/10.1145/3133896

[21] Michael J. Coblenz, Whitney Nelson, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine. 2017. Glacier: transitive class immutability for Java. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. IEEE / ACM, 496–506. https://doi.org/10.1109/ICSE.2017.52

[22] Tom Van Cutsem, Elisa Gonzalez Boix, Christophe Scholliers, Andoni Lombide Carreton, Dries Harnie, Kevin Pinte, and Wolfgang De Meuter. 2014. AmbientTalk: programming responsive mobile peer-to-peer applications with actors. *Comput. Lang. Syst. Struct.* 40, 3-4 (2014), 112–136. https://doi.org/10.1016/j.cl.2014.05.002

[23] Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. 2009. Linguistic symbiosis between event loop actors and threads. *Comput. Lang. Syst. Struct.* 35, 1 (2009), 80–98. https://doi.org/10.1016/j.cl.2008.06.005

[24] Frank S. de Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. 2017. A Survey of Active Object Languages. *ACM Comput. Surv.* 50, 5 (2017), 76:1–76:39. https://doi.org/10.1145/3122848

[25] Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter. 2015. Partitioned Global Address Space Languages. *Comput. Surveys* 47, 4, Article 62 (June 2015), 27 pages.

[26] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. 2006. Ambient-Oriented Programming in AmbientTalk. In *ECOOP*. 230–254.

[27] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, and Wolfgang De Meuter. 2006. Ambient-Oriented Programming in AmbientTalk. In *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4067)*, Dave Thomas (Ed.). Springer, 230–254. https://doi.org/10.1007/11785477_16

[28] Jack B. Dennis and Earl C. Van Horn. 1966. Programming semantics for multiprogrammed computations. *Commun. ACM* 9, 3 (1966), 143–155. https://doi.org/10.1145/365230.365252

[29] Theo D'Hondt (Ed.). 2010. *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*. Lecture Notes in Computer Science, Vol. 6183. Springer. https://doi.org/10.1007/978-3-642-14107-2

[30] Werner Dietl, Sophia Drossopoulou, and Peter Müller. 2007. Generic Universe Types. In *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4609)*, Erik Ernst (Ed.). Springer, 28–53. https://doi.org/10.1007/978-3-540-73589-2_3

[31] Darach Ennis. 2005. My Little Pony. At CodeMesh 2015. https://cdn.rawgit.com/darach/my_little_pony/master/my-little-pony.html.

[32] Kiko Fernandez-Reyes, Isaac Oscar Gariano, James Noble, Erin Greenwood-Thessman, Michael Homer, and Tobias Wrigstad. 2021. Dala: A Simple Capability-Based Dynamic Language Design For Data Race-Freedom. arXiv:2109.07541 [cs.PL]

[33] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, Robert Cartwright (Ed.). ACM, 237–247. https://doi.org/10.1145/155090.155113

[34] David Gelernter. 1985. Generative Communication in Linda. *ACM Trans. Program. Lang. Syst.* 7, 1 (1985), 80–112. https://doi.org/10.1145/2363.2433

[35] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and reference immutability for safe parallelism. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, Gary T. Leavens and Matthew B. Dwyer (Eds.). ACM, 21–40. https://doi.org/10.1145/2384616.2384619

[36] Olivier Gruber and Fabienne Boyer. 2013. Ownership-Based Isolation for Concurrent Actors on Multi-core Machines. In *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7920)*, Giuseppe Castagna (Ed.). Springer, 281–301. https://doi.org/10.1007/978-3-642-39038-8_12

[37] Philipp Haller and Alexander Loiko. 2016. LaCasa: lightweight affinity and object capabilities in Scala. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 272–291. https://doi.org/10.1145/2983990.2984042

[38] Philipp Haller and Martin Odersky. 2010. Capabilities for Uniqueness and Borrowing, See [29], 354–378. https://doi.org/10.1007/978-3-642-14107-2_17

[39] Douglas E. Harms and Bruce W. Weide. 1991. Copying and Swapping: Influences on the Design of Reusable Software Components. *IEEE Trans. Softw. Eng.* 17, 5 (May 1991), 424–435. https://doi.org/10.1109/32.90445

[40] John Hogg. 1991. Islands: Aliasing Protection in Object-Oriented Languages. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'91), Sixth Annual Conference, Phoenix, Arizona, USA, October 6-11, 1991, Proceedings*, Andreas Paepcke (Ed.). ACM, 271–285. https://doi.org/10.1145/117954.117975

[41] Michael Homer, Timothy Jones, James Noble, Kim B Bruce, and Andrew P Black. 2014. Graceful dialects. In *ECOOP (LNCS, Vol. 8586)*, Richard Jones (Ed.). Springer, 131–156.

[42] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. 2012. ABS: A Core Language for Abstract Behavioral Specification. In *Formal Methods for Components and Objects*, Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 142–164.

[43] Steve Klabnik and Carol Nichols. 2019. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press.

[44] Joeri De Koster, Stefan Marr, Tom Van Cutsem, and Theo D'Hondt. 2016. Domains: Sharing state in the communicating event-loop actor model. *Comput. Lang. Syst. Struct.* 45 (2016), 132–160.

[45] Joeri De Koster, Stefan Marr, Theo D'Hondt, and Tom Van Cutsem. 2015. Domains: Safe sharing among actors. *Sci. Comput. Program.* 98 (2015), 140–158.

[46] Neel Krishnaswami and Jonathan Aldrich. 2005. Permission-based ownership: encapsulating state in higher-order typed languages. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, Mary Hall (Ed.). Chicago, IL, USA, 96–106.

[47] James Larus and Galen Hunt. 2010. The Singularity System. *Commun. ACM* 53, 8 (Aug. 2010), 72–79. https://doi.org/10.1145/1787234.1787253

[48] Douglas Lea. 1999. *Concurrent programming in Java. Second Edition: Design Principles and Patterns* (2nd ed.). Addison-Wesley Longman Publishing Co., Inc., USA.

[49] Amit A. Levy, Michael P. Andersen, Bradford Campbell, David E. Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. 2015. Ownership is theft: experiences building an embedded OS in Rust. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems, PLOS 2015, Monterey, California, USA, October 4, 2015*, Shan Lu (Ed.). ACM, 21–26. https://doi.org/10.1145/2818302.2818306

[50] Paley Li, Nicholas Cameron, and James Noble. 2012. Sheep cloning with ownership types. In *FOOL 2012: 19th International Workshop on Foundations of Object-Oriented Languages*. Citeseer, 59.

[51] Paul Liétar. 2017. *Formalizing Generics for Pony*. Master's thesis. Imperial College London.

[52] Barbara Liskov and Liuba Shrira. 1988. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, Richard L. Wexelblat (Ed.). ACM, 260–267. https://doi.org/10.1145/53990.54016

[53] Nicholas D. Matsakis and Felix S. Klock II. 2014. The Rust language. In *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, HILT 2014, Portland, Oregon, USA, October 18-21, 2014*, Michael Feldman and S. Tucker Taft (Eds.). ACM, 103–104. https://doi.org/10.1145/2663171.2663188

[54] Mark S. Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Ph.D. Dissertation. Baltimore, Maryland.

[55] Mark Samuel Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Ph.D. Dissertation. Johns Hopkins University, Baltimore, Maryland, USA.

[56] Cameron Moy, Phúc C Nguyen, Sam Tobin-Hochstadt, and David Van Horn. 2020. Corpse Reviver: Sound and Efficient Gradual Typing via Contract Verification. *arXiv preprint arXiv:2007.12630* (2020).

[57] James Noble, Andrew P Black, Kim B Bruce, Michael Homer, and Timothy Jones. 2017. Grace's Inheritance. *Journal of Object Technology* 16, 2 (2017).

[58] Nikolaos Papaspyrou and Konstantinos Sagonas. 2012. On Preserving Term Sharing in the Erlang Virtual Machine. In *Proceedings of the Eleventh ACM SIGPLAN Workshop on Erlang Workshop* (Copenhagen, Denmark) *(Erlang '12)*. Association for Computing Machinery, New York, NY, USA, 11–20. https://doi.org/10.1145/2364489.2364493

[59] Michael Papathomas. 1989. Concurrency issues in object-oriented programming languages. *Object Oriented Development* (1989), 207–245.

[60] Benjamin C. Pierce. 2002. *Types and programming languages*. MIT Press.

[61] Richard Roberts, Stefan Marr, Michael Homer, and James Noble. 2019. Transient Typechecks Are (Almost) Free. In *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom (LIPIcs, Vol. 134)*, Alastair F. Donaldson (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 5:1–5:28. https://doi.org/10.4230/LIPIcs.ECOOP.2019.5

[62] Venetia Laura Delano Robertson. 2013. Of ponies and men: My Little Pony: Friendship is Magic and the Brony fandom. *International Journal of Cultural Studies* (2013).

[63] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1988. Global Value Numbers and Redundant Computations. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, Jeanne Ferrante and P. Mager (Eds.). ACM Press, 12–27. https://doi.org/10.1145/73560.73562

[64] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comput. Syst.* 15, 4 (1997), 391–411. https://doi.org/10.1145/265924.265927

[65] Jan Schäfer and Arnd Poetzsch-Heffter. 2010. JCoBox: Generalizing Active Objects to Concurrent Components, See [29], 275–299. https://doi.org/10.1007/978-3-642-14107-2_13

[66] Ilya Sergey and Dave Clarke. 2012. Gradual Ownership Types. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7211)*, Helmut Seidl (Ed.). Springer, 579–599. https://doi.org/10.1007/978-3-642-28869-2_29

[67] Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *IN SCHEME AND FUNCTIONAL PROGRAMMING WORKSHOP*. 81–92.

[68] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA (LIPIcs, Vol. 32)*, Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 274–293. https://doi.org/10.4230/LIPIcs.SNAPL.2015.274

[69] Tom Van Cutsem and Mark S. Miller. 2013. Trustworthy Proxies: Virtualizing Objects with Invariants. In *Proceedings of the 27th European Conference on Object-Oriented Programming* (Montpellier, France) *(ECOOP'13)*. Springer-Verlag, Berlin, Heidelberg, 154–178. https://doi.org/10.1007/978-3-642-39038-8_7

[70] Edwin M. Westbrook, Jisheng Zhao, Zoran Budimlic, and Vivek Sarkar. 2012. Practical Permissions for Race-Free Parallelism. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7313)*, James Noble (Ed.). Springer, 614–639. https://doi.org/10.1007/978-3-642-31057-7_27

[71] Christian Wimmer and Thomas Würthinger. 2012. Truffle: a self-optimizing runtime system. In *Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '12, Tucson, AZ, USA, October 21-25, 2012*, Gary T. Leavens (Ed.). ACM, 13–14. https://doi.org/10.1145/2384716.2384723

[72] Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (1994), 38–94. https://doi.org/10.1006/inco.1994.1093

[73] Tobias Wrigstad, Filip Pizlo, Fadi Meawad, Lei Zhao, and Jan Vitek. 2009. Loci: Simple Thread-Locality for Java. In *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5653)*, Sophia Drossopoulou (Ed.). Springer, 445–469. https://doi.org/10.1007/978-3-642-03013-0_21

[74] Derek Wyatt. 2013. *Akka concurrency*. Artima Incorporation.

[75] Yang Zhao and John Boyland. 2008. A Fundamental Permission Interpretation for Ownership Types. In *TASE*. 65–72.