Exploring the Design Space for Runtime Enforcement of Dynamic Capabilities[*]

Andrew Fawcet
andrew.fawcet@vuw.ac.nz
Victoria University of Wellington
Wellington, New Zealand

James Noble
kjx@programming.ac.nz
Creative Research & Programming
Wellington, New Zealand

Michael Homer
michael.homer@vuw.ac.nz
Victoria University of Wellington
Wellington, New Zealand

Abstract

Dala is an "as simple as possible" concurrent object-oriented language designed to avoid data races. Dala objects come in three safe flavours: immutable, isolated, and local, plus a fourth unsafe flavour. The objects are organised into an hierarchy so that e.g. immutable objects can be accessed from anywhere but never mutated, while thread local objects can be mutated but cannot be accessed outside their containing thread.

Dala's flavours are intended to be enforced at runtime: unfortunately it is not clear when and how best to undertake that enforcement.

In this paper we present six axes of variation for the dynamic enforcement of Dala flavours: when each safe flavour is enforced, how isolated objects are moved, how new objects are assigned a flavour, and whether objects' flavours can change over time. These six axes embody 2,880 different combinations: we present five exemplary designs and discuss how they are placed on those those axes. Programming language designers can use our analysis to inform the design of dynamic capabilities in their languages, while the analysis may also help programmers understand what language designers have done.

CCS Concepts: • **Software and its engineering** → *Dynamic analysis*; *Data types and structures*; *Error handling and recovery*; *Software design tradeoffs.*

Keywords: capability systems, dynamic enforcement

[*]This work is supported in part by the Royal Society of New Zealand Te Apārangi Marsden Fund Te Pūtea Rangahau a Marsden grant CRP2101.

ACM Reference Format:
Andrew Fawcet, James Noble, and Michael Homer. 2025. Exploring the Design Space for Runtime Enforcement of Dynamic Capabilities. In *Proceedings of the 2025 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3759429.3762625

1 Introduction

Fernandez-Reyes *et al.* proposed Dala, a capability-based language design aimed at avoiding data races using dynamic enforcement of certain object access invariants [8]. These capabilities ensure safe object access and data-race freedom — meaning access that complies with capability constraints and avoids simultaneous access to shared data by concurrent threads. The Dala language [8] sets out a specific set of run-time enforcement mechanisms, different for each of its capabilities, and both a formal model and a prototype implementation. That prototype implementation already differed from the formal model in some respects for practical reasons, and the different approaches used for each capability also highlight the scope of variability possible. We set out to investigate the design space of gradual capability enforcement more broadly when applied in a practical system, and to understand the trade-offs that can be made which enable or prevent certain kinds of programs, or certain styles of programming. Our exploration has broader applications than just Dala, investigating a matrix of possible approaches to any kind of gradual enforcement, including typing, ownership, and information flow control. To assist with these explorations, we have implemented a prototype system supporting six axes of variation, enabling a wide range of design combinations. As the original Dala paper presents the concrete language design using a variation of Grace [3], but the "Daddala" implementation presented in its extended version [9] is not available, we have extended an existing implementation [13] of the Grace language to most closely align with the original presentation.

Many type systems or correctness properties can be implemented through static checkers, dynamic detection, or gradations of "gradual" approaches in between — but exactly *how* the dynamic enforcement happens for different properties has wider implications. There are trade-offs made in any design: every approach has some costs — conceptual, performance, syntactic — but who bears them? Which *parts* of the program bear them? Different models will facilitate different kinds of program, and different kinds of programming; different properties may suit different styles of enforcement within these models. We aim to extend exploration of the design space beyond typical approaches in the literature, and to reify abstract concepts in a working system for experimentation with real programs.

The contributions of this paper are:

- Six axes of variation for gradual capability enforcement, along with an illustration of how different selections along these axes impact the structure and behavior of programs.
- A prototype implementation capable of executing and enforcing variations on all of these axes.

The next section briefly describes the baseline Dala design that inspired the present exploration. Section 3 describes each axis of variation we explore, showing through code examples the practical implications of different points on each axis. Section 4 discusses the implementation of our system supporting all of the models. Section 5 considers these more holistically, showing coherent sets of choices resulting in different language semantics or ergonomics, and discusses other issues uncovered during this exploration. Section 6 positions our approach amongst related work, and Section 7 concludes.

2 Baseline Dala

The baseline Dala design given by Fernandez-Reyes et al. [8] includes four flavours, which can be assigned to an object at creation time.

All reachable code and data will be safe once within an object of any of the three "safe" flavours:

- **imm** - The object is immutable, can be aliased freely, and can be accessed on any thread.
- **iso** - The object can have only one reference at a time, but can be moved freely between threads.
- **local** - The object can be aliased, but can only be accessed on the thread it was created on.

The fourth flavour, **unsafe**, is the default:

- **unsafe** – This flavour imposes no restrictions. It is the default, meaning that unannotated programs will continue to work as-is.

The safe flavours impose restrictions on which other flavours of object can be referenced and how the objects can be used. Safe annotations can gradually be added to an existing program to improve safety in that area. The intention is that the unsafe part of the program can be gradually reduced in a "bottom-up" way by converting the deepest unsafe objects to safe ones.

Dala's flavours are similar to "capabilities" in the literature [5, 10], and previous work has formalised their treatment using Dafny [19]. The capabilities have two parallel effects. First, they impose restrictions on the use of an object, as described above: where and when it may be dereferenced, mutated, or bound to a name. Dereferencing gives access to the internal state of an object. Second, they impose a hierarchy on the object graph — that is, they restrict which flavours of object can refer to which other flavours. The hierarchical properties are quite simple, and not the focus of our work here:

- An `imm` object can only hold references to other `imm` objects, and so must be deeply immutable.
- An `iso` object can hold references to `imm` or `iso` objects, but not to `local` or unsafe objects.
- A `local` object can hold references to `iso` or `imm` objects, but not to unsafe objects.
- An unsafe object can hold reference to any other object and has no safety properties of its own.

These hierarchical limitations are distinct from the usage restrictions on the objects themselves.

The concrete language syntax of baseline Dala looks like this:

```
var account := object is iso {
    var balance is public := 0
}
def branch = object is imm {
    def branchName = "Main Street"
}
def teller = object is local {
    def location = branch
    var activeAccount
    method switchAccount(newAccount) {
        return (activeAccount := newAccount)
    }
}
teller.switchAccount(account := erased)
def unsafeObj = object {
    def leak = teller
}
def outerChannel = spawn { innerChannel →
    def o = innerChannel.receive
    ...
}
outerChannel.send(unsafeObj)
```

That is, objects are created using the **object** keyword, and may have a flavour specified using the **is** keyword. Both local variables and fields may be created as constant (**def**) or mutable bindings (**var**). Mutable fields may be made public to allow access from outside the object, also using **is**, and will otherwise be accessible only within the object. Assignments to mutable locations use the := operator, and the previous value (if any) is returned. If the variable held an `iso` object, the returned `iso` is now free to be stored elsewhere. erased is a special value that can be assigned to a variable containing an `iso` object when it is to be moved out. Threads can be spawned with a block of code, and opposite ends of a channel will be returned by `spawn` and given as argument to that block of code.

The Dala system intends for its restrictions to be enforced dynamically, with errors raised at the point that a violation concretely occurs, prior to any data race occurring. Fernandez-Reyes *et al.* set out a specific, plausible approach to this enforcement for each capability, but without particular justification for the combination of choices made.

In brief, the baseline Dala design proposes enforcement of each flavour using a different mechanism as follows:

- **imm** - An object with this capability must have no mutable fields, and the values put into its fields at construction time must also be of the immutable flavour. This enforcement is carried out entirely at the time of construction.
- **iso** - The creation of a second reference to an iso object is an error, and all kinds of object — including unsafe! — must participate in this enforcement when their fields are assigned. An existing reference can be explicitly consumed, using a dedicated operation, to allow it to be moved to a new location or thread. This enforcement is carried out "outside" the object itself, by the system or by other objects, at the time of assignment.
- **local** - A local object can be freely aliased, including across threads, but *dereferencing* it on a thread other than the creation thread is an error. This enforcement is carried out "within" the object, at the time of dereference.

All of the safe flavours enforce the hierarchical invariants on their fields when they are set.

These are very defensible choices, but we can see that there are different mechanisms being used within just this single design. This contrast raises important questions. For example, what if `iso` were enforced at the point of dereference, rather than when the reference is created? Or what if `local` objects were prevented from crossing threads altogether, rather than being checked at dereference time? What if `imm` enforcement was deferred to the point of actual violation too, and raised an error only when a mutation occurred? We seek to explore the design space of gradual, dynamic enforcement of these types of dynamic capabilities, and to understand the trade-offs made in both the original Dala proposal and the alternatives we consider. It is clear that there are many possible mechanisms, and combinations of mechanisms, that would allow for different programs to be written, different executions to be permitted, and different errors to be raised at different times.

The extended arXiv version of the Dala paper [9] notes that its implementation, Daddala, differs slightly from the formal model. For example, it performs syntactic rewriting of the reference-consumption operation. The paper also speculates about alternative implementation strategies to handle corner cases that arise when embedding the model in a practical language:

> While it is therefore [by static analysis] possible to eliminate these errors (and other implementation strategies, such as proxies for isolates, can similarly address the issue), our current prototype permits a programmer who goes far outside the model to shoot themselves in the foot.

These deviations are seen there as mere practical compromises, but we see them as openings for deeper consideration. The ability to go "outside" the model makes for a *different* model, with different restrictions, frictions, and guarantees. Proxied isolates would have distinct semantics again — and more variants in turn give different affordances, different promises, and different errors. The original Dala work does not consider these variations any deeper. In the next section we set out a range of possibilities and show how meaningfully different language designs can arise from such small practicalities when examined as more than just implementation details.

3 Design

To explore this design space, we set out six different axes of variation, each with several candidate mechanisms listed. For our exploration we selected a number of design points to give reasonable coverage within this space. The axes and their candidate mechanisms emerged from iterative discussion and informed judgment, providing a structured framework for comparison. The intention is that a particular design can be represented as a combination of selections from each of these axes. We have implemented each of these possibilities in a prototype.

There are three "when" axes, one for each of the core object flavours, and representing the trigger point where its restrictions are checked and enforced. Each has different practical options, but the three are independent of one another and can be combined in any way — just as the baseline Dala approach uses different mechanisms for each. The other axes are more diverse, representing options relating to the determination of an object's flavour and the semantics of moving `iso` objects.

The axes are as follows:

- **iso-when** - When is the iso capability enforced?
- **iso-move** - How is an iso object moved?
- **local-when** - When is the local capability enforced?
- **capability-where** - Where is an object's capability determined?

- **capability-change** - When and how can an object change flavour?
- **imm-when** - When is the imm capability enforced?

Each of the axes will be treated in more detail below, including its candidate mechanisms. Within each axis, we list the option chosen by Fernandez-Reyes *et al.* in Dala first.

3.1 When Is The `iso` Capability Enforced?

There are five options for enforcing the `iso` capability.

- **assignment**: It is an error to create a second reference to an iso object. This is the baseline Dala behaviour.
- **dereference**: An iso object may only be dereferenced when it has a single living reference, but additional references may be created temporarily.
- **thread**: It is an error to transmit an iso object across a thread boundary while multiple references exist.
- **dereference-thread**: Both dereferencing and thread transmission of aliased isos are errors.
- **never**: `iso` is not enforced at all.

Baseline Dala enforces `iso` when any variable or field is assigned a value, reporting an error whenever the assignment creates an alias to an `iso` object. That is, an error arises immediately on:

```
var x := object is iso { ... }
var y := x // iso violation, alias created
```

Dereference-time enforcement borrows the approach used for `local` in baseline Dala described in Section 2, in which the `local` capability is enforced at the time of dereferencing. Under this model, temporary aliases to an `iso` object are permissible, but all but one must be discarded before the object is used. While violating the strict terms of the name, permitting temporary aliases allows for patterns such as swapping two `iso` objects via a temporary variable. The above code would *not* produce an error with dereference-time checking, but as soon as either `x` or `y` is dereferenced, a violation is reported:

```
var x := object is iso { ... }
var y := x
y.someMethod(1, 2) // iso violation, dereference with alias
```

However, it would be possible to exchange two `iso` objects via a third temporary variable, without dereferencing them in the middle:

```
var a := object is iso { ... }
var b := object is iso { ... }
var tmp := a
a := b // OK
b := tmp // OK
a.run // OK
```

Thread-boundary checking takes a different approach. In some respects, it brings `iso` closer to `local`. It indicates an object that can be aliased freely within a thread, where data races cannot occur. However, the object can still be transferred to another thread and used there, once thread-local processing is complete. Again, the above cases would *not* be errors, but it would not be permitted to send the object across a thread boundary:

```
1 var x := object is iso { ... }
2 var y := x
3 y.set 123 // to update the internal value of the object
4 def channel = spawn { channel2 → ... }
5 channel.send(y) // error, aliased iso crossing threads
```

Under the dereference-thread case, both the send on line 5 and the `y.set` on line 3 would be errors.

The "never" option here and in the other axes represents an extreme of the gradual approach, and stands in both for cases where the capability in effect does not exist at all in this variant, and cases where other axes address (some of) the matters of this axis. We do not distinguish these two cases in this work as they do not impact the semantics of the matrix of designs we are exploring.

3.2 How Is an `iso` Object Moved?

There are four options for "moving" an `iso` object, that is, for transferring a reference from one variable or field to another. An `iso` object that is not currently stored anywhere may be put into a variable, passed as an argument, or returned from a function, while one that is stored somewhere can only be put into a new location according to one of these models.

- **destructive read**: An existing reference must be erased by an explicit operation, returning the object ready to be stored elsewhere. This is the baseline Dala behaviour.
- **move**: Any operation that would create a second reference to the object instead erases the existing reference automatically, so that future access through that name will fail.
- **newest**: Aliasing an iso implicitly erases the previous reference, but when the alias ceases to exist (i.e. goes out of scope, or is itself overwritten), the original reference is restored and can be accessed again.
- **immobile**: An iso object can never move once stored.

Baseline Dala uses the *destructive read* approach, with an explicit operation that both erases a stored reference and returns the object to which that reference originally pointed. This ensures that the original reference can no longer be used, preventing unintended aliasing or reuse. Our prototype implements this behaviour by overloading the assignment operator `:=` so that, when used as an expression, it returns the previous value, and thus a destructive read is accomplished by assigning a replacement value; this semantic was also used by the Daddala implementation discussed in the

```
1  var x := object is iso { def num = 1 }
2  var y := (x := erased) // explicit consume operation
3  var z := y
4  z := process(z)
5  print(y.num)
```

Figure 1. A trivial program that proceeds differently under the different iso-move options. Line 2 performs an explicit consume operation: x is `erased`, and the original object is assigned to y. After that, x has no usable content, and y holds the only reference to the `iso` object.

original work's extended version, but with a layer of syntactic sugar providing the illusion of a "consume" operator. This behaviour is common in other systems with unique references, but its drawbacks have also been noted [4].

Line 2 of Figure 1 shows the x variable being destructively read and its value moved into y. Under the *destructive read* model, line 3 would be an error, while under the *move* option, z :=y successfully and implicitly moves the iso object from y to z, automatically erasing y with the same sentinel value as used explicitly on the previous line.

While "move" semantics are common in many languages, the "newest" option is more unusual, and can be seen as a kind of dynamically-enforced borrowing. The "newest" option allows an isolated reference to be passed as an argument, automatically restored when the argument is no longer in scope. However, it would also allow the new reference to be stored in a field indefinitely, while still revivifying the original much later.

Under *newest*, the assignment on line 3 of Figure 1 leaves y in a suspended state. It is an invalid reference, but "remembers" what it pointed to. On line 4, z can be passed as an argument, resulting in a further newer reference that is given to process, and so both y and z are suspended while process runs. When z is assigned a new value after the function returns, y is reënlivened and can then be used on line 5.

With the *immobile* option, an `iso` object can never be stored in a mutable variable, only an unchangeable **def**, and none of Figure 1 would be possible.

3.3 When Is The `local` Capability Enforced?

There are four options for enforcing the `local` capability.

- **dereference**: It is an error to dereference a local object on another thread. This is the baseline Dala behaviour.
- **thread boundary**: It is an error to transmit a local object to a different thread.
- **dereference-thread**: Both dereferencing and thread transmission of local objects are errors.
- **never**: `local` is not enforced at all.

Baseline Dala enforces `local` at dereference time, making it an error to send a message to a local object on a thread other than the one it was created on. However, it *does* allow a reference to a local object to be sent to and stored in another thread, provided that it is never dereferenced there. If a local object is sent to another thread, stored, and sent back, both the retrieved and original reference can be freely used on the original thread.

In Figure 2, the counter object is sent to another thread via the channel `channel1` on line 9. Within the spawned thread, the object is recieved on the corresponding endpoint `channel2` on line 5, and then sent back to the original thread via `channel2.send(obj)` on line 6. The dereference-time enforcement model allows this, but would raise an error on line 7 when the obj variable is dereferenced on the other thread.

Thread-boundary checking prevents a local object from being sent, and in fully-safe code this ought to be sufficient (depending on some choices on other axes). Under the thread and dereference-thread models, line 9 would be an error immediately, and the local counter object would not be permitted to cross threads in the first place. With `local` *never* enforced, there would be no error reported, but the data race on lines 7 and 11 would occur and line 12 might print an unexpected value.

With unsafe objects available, it is possible that a reference to a local is accessible from another thread without ever being sent across a thread boundary. Figure 3 shows an instance of this, where an unsafe object holds a reference to the local object. Both dereference models would raise an error on line 9 when the local object is accessed: accessing the "`value`" field would trigger the dereference-time enforcement, which would detect that this access is from a different thread than the one where the local object was created. The thread-only model would not detect this, as the local object itself was never sent across a thread boundary. Without the existence of unsafe objects in the system, all three models would prevent a potential data race of this sort from arising. Whether this is undesirable, or the wages of the sin of using unsafe code, is a philosophical question. On the other hand, preventing a local object from being sent to another thread seems obvious, but there are plausible cases where it is safe and useful.

3.4 Where Is an Object's Capability Determined?

There are three options for determining an object's capability.

- **static**: The capability is determined at compile time by the source code, and cannot be changed. This is the baseline Dala behaviour.
- **construction**: The capability is determined at the time the object is constructed, potentially using run-time information. For example, a method could return a

```
 1 def counter = object is local {
 2     var value is public := 1
 3 }
 4 def channel1 = spawn { channel2 →
 5     var obj := channel2.receive
 6     channel2.send(obj)
 7     obj.value := obj.value + 2 // data race with line 11
 8 }
 9 channel1.send(counter)
10 def otherCounter = channel1.receive // same object
11 counter.value := counter.value + 3 // data race with line 7
12 print(otherCounter.value)
```

Figure 2. A local object sent to a newly-spawned thread in a trivial program illustrating a data race, utilising immutable bindings (`def`). Different enforcement models will detect and report the error at different times.

```
 1 def counter = object is local {
 2     var value is public := 1
 3 }
 4 def unsafeObj = object {
 5     def tally is public = counter
 6 }
 7 def channel1 = spawn { channel2 →
 8     def obj = unsafeObj.tally
 9     obj.value := obj.value + 2 // data race with line 11
10 }
11 counter.value := counter.value + 3 // data race w/ line 9
12 print(counter.value)
```

Figure 3. An unsafe object, which may be transmitted across or accessed on multiple threads, with a reference to a local object. Under different models, the data race may or may not be detected.

fresh object that is either `local` or `iso` depending on parameters passed to it from the call site.

- **variable-side**: Rather than objects being born with a specific flavour, variables and fields are annotated with capabilities. Storing an object into such a variable gives it the flavour indicated by that variable's annotation.

Baseline Dala annotates objects statically with their capabilities directly within the source code as part of the object constructor syntax. The top part of Figure 4 shows this more standard model, where the object constructor syntax includes an **is** annotation setting the flavour of this object.

In the construction-time model, the syntax is relaxed so that dynamic expressions, not just the three capability names literally, can be used to determine the flavour of the object.

```
// Object flavour determined statically
var w := object is iso { ...}
def x = object is local { ... }

// Object flavour determined by variable annotation
var y is iso := object { ... }
def z1 = object { ... }
def z2 is local = z1
```

Figure 4. Illustration of the distinct syntactic forms for static and variable-side capability determination. `w` and `x` are constructed with **is** iso determining their flavour at birth, as part of the *object constructor* syntax. The object in `z1` is `local` because it is stored into a *variable* with that annotation.

With this approach, a factory method could accept `local` or `iso` as a parameter, and the object constructor within it could be annotated with that parameter, allowing the flavour of the newly-created object to be determined at the call site.

The variable-side option is a drastic change from the baseline, in effect permitting objects to start as unsafe and crystallise into one of the safe capabilities according to how they are used. The flavour of an object is not intrinsic to it, but determined by the context in which it is stored — specifically, the capability annotation of the variable or field that holds it. The bottom part of Figure 4 shows this approach: the object syntax itself does not refer to a capability, but variable declarations do. `z1` and `z2` show an object that begins as unsafe but crystallises into `local` once stored into a variable with that annotation.

There are a number of nuanced interactions that arise with other axes here. For example, if an object already has multiple aliases and is assigned to an `iso`-annotated variable, under an "assignment"-time check this should be an immediate error, but with "dereference"-time checking there is no issue at this point (although there may be expensive bookkeeping to do). It is also conceivable that an object is held by both `local` and `imm` locations: depending on the combination of other axes, this could be an error, or it could be that the object must comply with both sets of rules. This approach is the most invasive, but offers a great deal of flexibility, particularly in combination with some of the capability-change options.

3.5 When and How Can an Object Change Flavour?

Four models describe how and when an object's capability may be changed:

- **fixed**: Once an object has a capability, it can never be changed. This is the baseline Dala behaviour.
- **dynamic**: There are operations to set the capability of an object at any time.
- **dynamic-iso**: An iso object can be changed to local or imm at run time, but no other flavour can be changed.

- **duration**: For as long as a reference to an object asserting a given flavour exists, the object conforms to those rules, but these relax if the last such reference is destroyed.

Baseline Dala fixes the capability of an object for its entire lifetime, and never allows it to change.

The dynamic-iso model permits the case of creating an initially-mutable object and, after some initialisation phase is complete, transforming it to one of the other flavours. As the reference to an `iso` object is unique, reclassification cannot violate capability assumptions elsewhere in the program. This model is particularly useful for initialization patterns or factory-based object creation, where an object can be safely transformed after a controlled setup phase and be immutable or local to a different thread thereafter. Both dynamic and dynamic-iso enable the code in Figure 5, where an `iso` object becomes `local` once transferred to the other thread.

Under the *dynamic* model, movement between any two flavours is permitted, which can be useful in cases where an object needs to adapt its capabilities based on runtime conditions or interactions with other objects. However, this flexibility comes at the cost of increased complexity and potential safety concerns, as the programmer must ensure that the object's capabilities are correctly managed throughout its lifecycle, particularly where the object may be shared across multiple contexts.

There are interactions with other axes under this model. For example, suppose an object has the `local` capability, and there are existing aliases on the same thread, but it is now converted to `iso`. Under the "assignment" model on the iso-when axis, where aliases to `iso` cannot be created, this must be an immediate error. On the other hand, under the "dereference" model on the same axis, there is no issue at this point: multiple references to an `iso` object exist, and dereferencing them will be an error until only one remains. The iso-move "newest" model would seem to require significant bookkeeping either at the time of the flavour change, or at all times for all objects that may change. Under iso-move "move", all other references to the object must be invalidated, in the manner they would have been if the object were moved out of them, meaning they must all be tracked. In both cases, this "action at a distance" may break expectations in other parts of the code, and the programmer is relied upon to ensure that the change can be handled. These and other interactions are trade-offs for the designer when selecting points on these axes, and this maximally-dynamic model imposes significant costs. Certain programs are possible under the dynamic model that no other model can express, but whether those programs are worth those costs is a question for the language designer.

The duration model pairs primarily with the variable-side capability determination model, allowing the object flavour to change fluidly as the set of variables holding references to

```
def channel1 := spawn { channel2 →
    var y := channel2.receive
    capability.set(y, local)
}
var x := object is iso { ... }
// ... setting up x ...
channel1.send(x := erased)
```

Figure 5. The flavour of an object being explicitly changed under the dynamic change models. The `iso` object is transferred to a different thread and then made `local` to that thread.

it change. Capabilities are inferred from the type of reference currently holding the object, and the object must comply with the associated rules as long as that reference is alive. Once an object is no longer held in an `iso` variable, for example, it may be aliased freely — but if then stored into a `local` variable or field, it must comply with the rules of that flavour, affecting all references to it. Figure 6 shows an example of this, where an object becomes `local` once stored into a variable with that annotation. Line 11 may present an error, depending on the position taken on other axes, because although x does not have a `local` annotation, the object it refers to is of the `local` flavour for as long as it is held in y. On line 17, y is no longer in scope and so the restrictions on x are relaxed.

Most of the cross-axis interactions discussed for the dynamic model also apply to the duration model, but the benefits and the structure of programming that it fosters are different. Certain combinations may be more naturally useful alongside it than others, but the kinds of programs that are intended to be supported will determine which for any given language design.

3.6 When Is The `imm` Capability Enforced?

There are three options for enforcing the `imm` capability.

- **construction**: The imm capability is enforced at the time of construction. This is the baseline Dala behaviour.
- **mutation**: An imm object may hold references to mutable state, but it is an error if state that has changed after construction is accessed.
- **never**: The imm capability is not enforced at all.

Baseline Dala enforces `imm` at construction time, requiring that all reachable state be deeply and fully immutable at the moment the object is instantiated. This ensures strong guarantees but restricts design flexibility.

Under the "mutation" option, immutability is enforced lazily: objects may reference mutable state, provided that the referenced state remains unchanged. If such state is later modified and subsequently accessed, an error is raised at

```
1  var x := object { ... }
2  // ... x is unsafe here ...
3  def channel1 = spawn { channel2 →
4      var obj := channel2.receive
5      obj := channel2.receive
6      obj.set(123)
7  }
8  method process(y is local) {
9      // Object is local while this method is running
10     // because it is held in a local–annotated variable.
11     channel1.send(x) // sending a local across threads
12     y.set(456)
13 }
14 process(x)
15 // At this point, x is unsafe again,
16 // unless other local references exist
17 channel1.send(x)
```

Figure 6. The variable-duration model permits the flavour of an object to change freely according to the places references to it are held. The flavour change applies to the object, however it is accessed, so line 11 is sending a `local` object across a thread boundary despite using the unannotated x reference. However, on line 17, y no longer exists and so the restrictions on x are relaxed.

runtime. For example, a mutable list could be created, values added to it, and the list then stored in a freshly-constructed `imm` object. The `imm` object can read from the list and use the list's non-mutating methods, but if the list is ever modified by other code it will become inaccessible from the `imm` object and an error will be raised if the object attempts to make use of it. This permits patterns involving "effectively final" values, offering greater flexibility at the cost of deferred validation and potentially subtle errors.

The distinction between construction-time and mutation-time enforcement is illustrated in Figure 7. Here, an `imm` object defines a field `a` referencing a mutable object, whose state is accessed in the method `foo`. With construction-time enforcement, this program fails immediately due to assigning a mutable object to a field of an `imm` object. Under mutation-time enforcement, the program is valid until the *second* foo call: when it accesses a.x it will observe the mutation that occurred on line 9 and report an error.

The "never" strategy disables enforcement entirely. The object still has the flavour and *other* checks may make use of it, but immutability guarantees are forfeited.

```
1  def mutableObject = object {
2      var x := 1
3  }
4  def immutableObject is imm {
5      def a = mutableObject
6      method foo { a.x }
7  }
8  print(immutableObject.foo)
9  mutableObject.x := 2
10 print(immutableObject.foo) // error, x was changed
```

Figure 7. A trivial program behaving differently under the different `imm` enforcement models. Construction-time enforcement will raise an error on line 5, while mutation-time enforcement will allow the program to run to line 10, where the visible mutation will be reported.

4 Implementation

Our prototype implementation extends an existing Java-based Grace interpreter with all of the axes of variation described above. The implementation is not a high-performance production system, implementing the semantics of the given language variant directly, and makes no attempt at efficiency of either computation or memory use. For example, it will maintain reference counts for all objects where those may be required, or wrap `iso` objects in proxy objects for some of the iso-move variants, notwithstanding that a production system could optimise much of this overhead away. Some variants or combinations may not have efficient implementations available at all, and the excess work our prototype performs to support the range of variants obscures the performance impact of individual design choices. We are concerned primarily with accurately reflecting the diverse semantics across the matrix of designs.

Performance questions are not the focus of the present work, and would be better addressed within a concrete context. While it seems intuitively likely that some of the models impose significant run-time overhead, we would be reluctant to speculate too strongly on the practical magnitude of this. It has been established by Roberts et al. Roberts et al. [20] that high-performance optimising virtual machines can often eliminate almost all costs of dynamic gradual type enforcement, despite the same widespread intuitions having suggested the costs were inescapable. We are open to the possibility that fusion with, for example, the garbage collection system may eliminate much of the overhead imposed, but also that some of the models may be inherently expensive with present technology. Our present work gives no substantive evidence either way, so we will not make any claims about it performance but commend exploring the practical impacts of different of our models to future work.

Design	iso-when	iso-move	local-when	cap.-where	cap.-change	imm-when
Dala	assignment	destructive	deref.	static	never	construction
Late enf.	dereference	conditionally aliasable	deref.	static	never	mutation
Var. duration	assignment	destructive	deref.	var.-side	duration	mutation
Thread enf.	thread	conditionally aliasable	thread	static	never	construction
Borrowing	dereference	newest	deref.	static	never	construction

Figure 8. The five designs being examined with these case studies mapped against the axes of variation.

Many programs can be executed under a wide variety of the designs, but some require syntactic or structural changes (for example, variable-side annotations or the insertion of explicit moves). In some cases, while a program can be executed under a variant, it may be "unidiomatic", for as much as that term makes sense in this context, by not using the features of the variant where it could. For example, a program that creates a new immutable object cloning data from an `iso` object will still work on a variant that allows changing the original object to be immutable, but it would be more idiomatic to make use of that capability-change feature.

The full implementation is available from https://zenodo.org/records/16939265 [7].

5 Discussion

While the six axes can be varied independently, coherent selections of different points represent distinct paths of design. To see the impact of the different design choices, we will focus on a small set of contrasting combinations on our axes, shown with respect to the axes in Figure 8:

- **Dala**: The original Dala design, with the same mechanisms as described in the original paper. A `local` object dereferenced on a different thread than it was created on will raise an error, while creating a second reference to an `iso` object immediately errors as well.
- **Late enforcement**: The latest, most dynamic enforcement option on each of the *when* axes that is able to detect an error. This design defers errors as late as possible, allowing the program to continue until it cannot go any further.
  The programmer is encouraged to experiment with a running system, finding out concretely where assumptions are violated. An `iso` object is able to be aliased, and as long as the object is not dereferenced the code will continue to run. While code that will inevitably result in an error will be permitted to run, a concrete instance of the error will always be produced, potentially aiding development. In this way this approach gives an analogous story to programming in dynamically-typed languages.
- **Variable duration**: Dynamically-changing flavours for as long as a reference to an object exists in a variable annotated with that flavour.
  For example, an object stored in a variable or field marked `local` is protected from access on another thread for as long as it is there, and as long as code relying on that restricted reference expects it to be — but those constraints are relaxed once that reference is no longer there. Similarly, an object stored in a variable marked `iso` is certainly unaliased, but can be moved into a `local` and changes flavour at that point.
  Which properties are being relied on is always locally manifest: code that needs a certain reference to be thread-local or unique can assert that in situ, but in contrast, code that is *not* expecting a reference to be restricted may be surprised by action at a distance. This design favours fluidity, and may be more amenable to gradual adoption of safe capabilities in existing code.
- **Thread enforcement**: `local` and `iso` are enforced when crossing a thread boundary only, focusing on these threshold points.
  Here in some respects the levels of restriction on `iso` and `local` are reversed: an `iso` is a safe object with an additional privilege of changing threads when the reference is unique, while both can be aliased within the same thread. An `iso` object without aliases is able to be moved to a different thread with a destructive read. In this case `iso` is likely not the best name, but the semantics are plausibly desirable and again may reflect existing, unannotated code more closely.
- **Borrowing**: References to `iso` are automatically borrowed and returned, while `local` objects are enforced at dereference time.
  This model uses the "newest" option for `iso` movement — where the most recent surviving reference is the only usable one — and so common patterns like passing a reference to an `iso` object as an argument that is used and discarded are permitted. If the passed reference is stored, the original reference is unusable and so that will be detected if unexpected, but revives once that stored reference is erased. It is analogous to the static borrowing in many ownership systems, but fully dynamic.

These five designs are not exhaustive, but they do represent a range of the design space such that every axis varies at least once. Code in any of the designs other than variable-duration will be syntactically valid in the others, but may have different error behaviour or be unidiomatic, while the variable-duration design inherently requires syntactic changes to code.

All of these designs present some advantages and some disadvantages, according both to the task at hand and the intended semantics of the program.

The borrowing model allows many straightforward operations to be straightforwardly expressed, with no local bookkeeping burden or need to care about the capability system, while creating opportunities for errors reported far from their cause.

The variable-duration model enables extremely dynamic assertions of properties that need to hold *right now*. It even allows data structures that deliberately cause changes of flavour, such as a set that makes its elements immutable. At the same time, it inherently and deliberately invokes spooky action at a distance by modifying objects based on localised statuses. The errors that arise far from their cause may be confusing, and it is unclear whether the benefits are worth this cost without empirical user testing.

Late enforcement forces the program to run as far as possible before producing a concrete instance of a failure. It allows temporarily-invalid states to exist as long as they are not part of the control flow. In doing so it saves the programmer from pointless busywork to satisfy the compiler, but allows execution beyond the point where errors were inevitable. Exposing concrete errors that have actually occurred may be helpful for development, but may also permit serious flaws to reach production deployment. Already, different languages make different choices on this front (most obviously, statically- and dynamically-typed languages), and the same approaches used to study these may be applied to this kind of error detection.

We do not suggest that our examination of these models is exhaustive, but hope to highlight the breadth of the design space and that very different trade-offs have potential benefits for different uses, without identifying a single "best" design. Any one of our candidate designs appears that it could be useful for some language with some intended use cases. Future work can explore specific designs in more depth from both software engineering and pedagogical perspectives, as well as exploring the performance implications.

We do note a limitation uncovered in our experimentation, but not directly related to the design space we are exploring. The issue relates to the "practical" embedding of the Dala model within the Grace language, and was not noted in the original Dala paper, but seems to be a limitation for real-world applications of this kind of dynamic capability system.

The Grace language, like many others, has objects and first-class code blocks that close over their enclosing lexical scope, and retain privileged visibility and access to their surroundings. An object literal or code block created within an object's method can access fields and methods of that object. If that enclosing object is `iso`, this closure is in effect creating a forbidden second reference to the object. Similarly, an immutable object could capture and make use of mutable state exposed by surrounding scopes. Grace's module system [11] has a "dialect" feature explicitly built around these sorts of injections into scope [12], and its inheritance can also bring references into a scope [18]. While the baseline Dala model has no issue with `local` in this way, due to dereference-time checking, the other object flavours present a problem.

Some of our models — primarily those with late enforcement — do permit these closures to be created while still catching any actual errors that occur. We consider that a possible point in favour of those approaches. Further static analysis to detect and reject closures that make use of references that violate the structural rules is another possibility, although this results in a significantly less dynamic model, and in some cases, such as code blocks used for control structures, it represents a quite severe limitation. Alternatively, we suggest that it may be that there is an incompatibility between this type of capability model and this type of language, and so such a model is more fitting for a somewhat more static style of programming language. For our implementation, we disregard the implicit references created by these closures entirely, as they do not raise practical issues in "normal" code not trying to exploit them. We leave resolving this issue, which we do not believe has been raised previously in the literature, to future work.

6 Related Work

This work builds directly on the Dala capability model [8], which introduced a minimal yet expressive set of capabilities—local, isolated, and immutable—for ensuring safe concurrent programming. The original Dala system emphasized a dynamic approach to capability checking, enabling interoperability between safe and potentially racy code, and made a pragmatic set of design choices to balance performance, safety, and simplicity. Our work extends this by exploring the application strategies for these capabilities, showing that their enforcement points—such as at thread boundaries or dereference—significantly affect both the usability and guarantees of the system.

6.1 Capability Models and Concurrency

A broad range of work has explored ownership and capability systems as tools for safe concurrency. Languages such as Pony [24], Rust [15], and systems like Verona [2] enforce capabilities statically to ensure race freedom. Our work follows Dala in retaining a dynamic enforcement strategy, but investigates whether alternative enforcement locations (e.g., on dereference rather than message send) can influence expressiveness and safety. This complements prior work on hybrid models like those of HJp [25] and gradual ownership [23], which similarly seek a middle ground between static guarantees and dynamic flexibility.

Our exploration of delayed versus eager enforcement resembles earlier debates in race detection and memory models [22], where tradeoffs between false positives, performance,and programmer intent have long been discussed. Unlike race detectors that flag violations post hoc, capability enforcement offers a means for programmers to declare intent—a distinction our experiments aim to make more precise through variation in enforcement timing.

6.2 Channels

Rogliano et al. explore runtime mechanisms for enforcing permission transfer semantics over channels in dynamically typed languages [21]. Their system categorizes permission transfer into four transfer modes, `copy`, `full-transfer`, `exclusive-write`, and `read-only`, and implements these via write barriers and partial-read barriers within a channel framework. While both their system and Dala address data race prevention through runtime enforcement, their focus remains on channel-mediated access control, rather than general-purpose object capability management. In contrast, Dala's design supports capability-based reasoning that is independent of the communication mechanism, and introduces capability flavours to govern aliasing and transfer. Both systems demonstrate the utility of runtime enforcement, but Dala offers a more flexible model that applies permission control across general object interactions, not just channel usage.

6.3 Handles

Another approach to reference control in dynamic systems is the use of handles, as proposed by Arnaud et al. [1]. Handles are behavior-propagating, first-class references that dynamically impose access semantics, such as read-only or revocable access, without relying on static types. Unlike Dala's capability model, which attaches semantic guarantees to object references using flavour-specific constraints (`local`, `iso`, `imm`), handles operate by interposing a proxy on each reference, dynamically enforcing restrictions only when accessed through the handle. This approach allows semantic behavior to propagate automatically through the reachable object graph, enabling control over aliasing and mutation at the level of individual references. While both systems share a focus on safe reference management in dynamic languages, Dala enforces constraints via explicit capability transitions and destructive updates, whereas handles embed semantics in reference wrappers that mediate behavior. Both models demonstrate how reference semantics can be enforced without static annotations, but differ in propagation mechanisms.

6.4 Variants of Isolation and Immutability

The notion of isolated references, central to our system, echoes earlier models from Singularity [14], where ownership and isolation were used to support zero-copy message passing. Similarly, our notion of deep immutability draws on work such as Glacier [6], which enforces recursive immutability for conceptual clarity. While our system inherits these semantics, we vary the conditions under which these guarantees are enforced—exploring, for instance, whether an immutable object's fields must be transitively frozen immediately or lazily upon first use.

Systems like E [16] and AmbientTalk [17] also combine object capabilities with concurrency models (e.g., vats, actors), but typically embed these concerns at the language design level. By contrast, our work modifies the runtime semantics of a capability system without altering its surface syntax, thereby illuminating how capability application can shift guarantees without requiring new annotations or constructs.

6.5 Applying Capabilities Dynamically

Dala's original system emphasized dynamic enforcement over static typing, similar to approaches taken in the Grace language and Moth [20]. Our implementation continues in this spirit, but introduces configurable enforcement modes that allow developers or systems researchers to experiment with new enforcement strategies. This has parallels to work in runtime systems like Truffle/Graal [26], which optimize dynamically enforced properties via speculative execution and inlining.

While prior work on capabilities has focused on expressivity and guarantees, we focus instead on where enforcement occurs and how such decisions affect race freedom, performance, and program modularity. To our knowledge, this is the first systematic study of enforcement placement within a reference capability system, and our results suggest a nuanced space of trade-offs that are underexplored in the literature.

7 Conclusion

We have shown a range of different approaches to enforcing a relatively simple dynamic capability system, building on the Dala data-race-freedom model proposed by Fernandez-Reyes et. al [8]. Through examples of the behaviour of different choices on six distinct axes of variation, we can see small changes to enforcement creating meaningful programmer-level impacts, and coherent selections of different points on these axes creating significantly different language designs. Our implementation of all six of these axes highlights the practicalities of the trade-offs made in these, and we encourage further exploration of *how* and *when* these kinds of type properties are dynamically enforced, beyond only considering their static semantics and treating practical implication choices as irrelevant details, but rather incorporating them as first-class concerns in the design of programming languages and systems.

References

[1] Jean-Baptiste Arnaud, Stéphane Ducasse, Marcus Denker, and Camille Teruel. 2015. Handles: Behavior-propagating first class references for dynamically-typed languages. *Science of Computer Programming* 98 (2015), 318–338. doi:10.1016/j.scico.2014.07.011

[2] Ellen Arvidsson, Elias Castegren, Sylvan Clebsch, Sophia Drossopoulou, James Noble, Matthew J. Parkinson, and Tobias Wrigstad. 2023. Reference Capabilities for Flexible Memory Management. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023), 1363–1393. doi:10.1145/3622846

[3] Andrew P. Black, Kim B. Bruce, Michael Homer, James Noble, Amy Ruskin, and Richard Yannow. 2013. Seeking Grace: A New Object-Oriented Language for Novices. In *ACM Technical Symposium on Computer Science Education*. doi:10.1145/2445196.2445240

[4] John Boyland. 2001. Alias burying: Unique variables without destructive reads. *Software: Practice and Experience* 31, 6 (2001), 533–553. doi:10.1002/spe.370 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.370

[5] John Boyland, James Noble, and William Retert. 2001. Capabilities for Sharing. In *ECOOP 2001 — Object-Oriented Programming*, Jørgen Lindskov Knudsen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 2–27. doi:0.1007/3-540-45337-7_2

[6] Michael Coblenz, Whitney Nelson, Jonathan Aldrich, Brad Myers, and Joshua Sunshine. 2017. Glacier: Transitive Class Immutability for Java. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 496–506. doi:10.1109/ICSE.2017.51

[7] Andrew Fawcet and Michael Homer. 2025. Dala Variants Implementation. doi:10.5281/zenodo.16939265

[8] 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 *ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. doi:10.1145/3486607.3486747

[9] 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. *CoRR* abs/2109.07541 (2021). arXiv:2109.07541 https://arxiv.org/abs/2109.07541

[10] Colin S Gordon. 2020. Designing with Static Capabilities and Effects: Use, Mention, and Invariants. *arXiv preprint arXiv:2005.11444* (2020).

[11] Michael Homer, Kim B. Bruce, James Noble, and Andrew P. Black. 2013. Modules As Gradually-typed Objects. In *Proceedings of the 7th Workshop on Dynamic Languages and Applications* (Montpellier, France) *(DYLA '13)*. ACM, New York, NY, USA, Article 1, 8 pages. doi:10.1145/2489798.2489799

[12] Michael Homer, Timothy Jones, James Noble, Kim B. Bruce, and Andrew P. Black. 2014. Graceful Dialects. In *ECOOP 2014 — Object-Oriented Programming*, Richard Jones (Ed.). Lecture Notes in Computer Science, Vol. 8586. Springer Berlin Heidelberg, 131–156. doi:10.1007/978-3-662-44202-9_6

[13] Michael Homer and James Noble. 2025. Fast & Easy ASTs for Flexible Embedded Interpreters. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '25)* (Singapore, Singapore). ACM, 8 pages. doi:10.1145/3759426.3760977

[14] James Larus and Galen Hunt. 2010. The Singularity System. *Commun. ACM* 53, 8 (2010), 72–79. doi:10.1145/1787234.1787255

[15] Nicholas D. Matsakis and Felix S. Klock. 2014. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology* (Portland, Oregon, USA) *(HILT '14)*. Association for Computing Machinery, New York, NY, USA, 103–104. doi:10.1145/2663171.2663188

[16] Mark Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Johns Hopkins University.

[17] Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. 2018. A CAPable Distributed Programming Model. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 88–98. doi:10.1145/3276954.3276957

[18] James Noble, Andrew P. Black, Kim B. Bruce, Michael Homer, and Timothy Jones. 2017. Grace's Inheritance. *The Journal of Object Technology* Volume 16, no. 2 (2017). doi:10.5381/jot.2017.16.2.a2

[19] James Noble, Julian Mackay, Tobias Wrigstad, Andrew Fawcet, and Michael Homer. 2024. Dafny vs. Dala: Experience with Mechanising Language Design. In *Workshop on Formal Techniques for Java-like Programs*. doi:10.1145/3678721.3686228

[20] Richard Roberts, Stefan Marr, Michael Homer, and James Noble. 2019. Transient Typechecks are (Almost) Free. In *European Conference on Object-Oriented Programming*. doi:10.4230/LIPIcs.ECOOP.2019.5

[21] Théo Rogliano, Guillermo Polito, Luc Fabresse, and Stéphane Ducasse. 2021. Analyzing permission transfer channels for dynamically typed languages. In *Proceedings of the 17th ACM SIGPLAN International Symposium on Dynamic Languages*. 23–34. doi:10.1145/3486602.3486769

[22] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems (TOCS)* 15, 4 (1997), 391–411. doi:10.1145/265924.265927

[23] Ilya Sergey and Dave Clarke. 2012. Gradual Ownership Types. In *European Symposium on Programming (ESOP 2012) (Lecture Notes in Computer Science, Vol. 7211)*. Springer, 579–599. doi:10.1007/978-3-642-28869-2_29

[24] George Steed and Sophia Drossopoulou. 2016. A Principled Design of Capabilities in Pony. https://www.ponylang.io/media/papers/a_prinicipled_design_of_capabilities_in_pony.pdf. White paper, accessed April 2025.

[25] Edwin Westbrook, Jisheng Zhao, Zoran Budimlić, 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)*. Springer, 614–639. doi:10.1007/978-3-642-31057-7_27

[26] Christian Wimmer and Thomas Würthinger. 2012. Truffle: A Self-Optimizing Runtime System. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*. 13–14. doi:10.1145/2384716.2384723

Received 2025-04-25; accepted 2025-08-11