Dafny vs. Dala: Experience with Mechanising Language Design

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

Julian Mackay
School of Engineering & Computer Science
Victoria University of Wellington
Wellington, New Zealand
julian.mackay@ecs.vuw.ac.nz

Tobias Wrigstad
Department of Information Technology
Uppsala University
Uppsala, Sweden
tobias.wrigstad@it.uu.se

Andrew Fawcet
School of Engineering & Computer Science
Victoria University of Wellington
Wellington, New Zealand
fawcetandrew@ecs.vuw.ac.nz

Michael Homer
School of Engineering & Computer Science
Victoria University of Wellington
Wellington, New Zealand
mwh@ecs.vuw.ac.nz

Abstract

Dala is a design for a concurrent dynamic object-oriented language. A key goal of Dala's design is to avoid data races, by ensuring threads do not share mutable state. In this paper we discuss our experience using the program verification tool Dafny to validate Dala's design. We explain how we modelled salient features of Dala in Dafny, and how Dafny did (or did not) assist our confidence in Dala's design.

CCS Concepts

• **Software and its engineering** → **Object oriented languages**; **Software verification**; *Semantics*.

Keywords

Dala, Dafny, Ownership, Uniqueness, Immutability

ACM Reference Format:
James Noble, Julian Mackay, Tobias Wrigstad, Andrew Fawcet, and Michael Homer. 2024. Dafny vs. Dala: Experience with Mechanising Language Design. In *Proceedings of the 26th ACM International Workshop on Formal Techniques for Java-like Programs (FTfJP '24), September 20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3678721.3686228

1 Introduction

Today's phones and laptops have ten or more processor cores, while datacentres have tens of millions. To use all these cores effectively, programs have to be concurrent, with multiple simultaneous threads of execution. Most of today's concurrent programs, however, are written in low-level languages which provide no correctness guarantees. To address this problem, we have been working on the design of Dala, a simple concurrent object-oriented programming language [26, 27], based on the Grace educational object-oriented programming language [4]. Dala is based on a novel model of *concurrent dynamic ownership* [28, 51] tailored to ensure that all valid programs are thread-safe by design (aka data race free, "fearlessly concurrent", "disentangled" etc). We have a design for Dala, prototype implementations based on various different Grace systems, and a LaTeX formal model which we claim ensures data-race freedom [25, 27].

In this short paper, we describe our attempts to formalise key features of the design of Dala using the Dafny verification-based programming language [42, 44, 48]. We have chosen to work with Dafny for the simple reason that Dafny was the verification tool with which we had the most experience [22, 50, 52]. Compared with some other tools [16, 18, 45], Dafny is not designed for validating type systems [19, 46, 47]: we hope this project will help to evaluate Dafny in such a task.

The next section introduces Dala and Dafny, and then we present some key features of modelling Dala with Dafny.

2 Background

2.1 Ownership and Structured Heaps

Many contemporary programming languages such as Rust [36], Pony [14], Encore [9], Obsidian [15], and Verona [12] have demonstrated the efficacy of *static ownership* [13, 53] to ensure concurrent programs are safe: Rust in particular has been adopted by Microsoft [34, 38].

By keeping track of each object's ownership, these languages can determine when an object may be used, when it may be changed, and the effects those changes can have on the rest of the program. While much simpler than full-scale program proof systems, these languages rely on complicated static (compile-time) rules and restrictions, with many different capability annotations and ownership parameterisations that many programmers often find hard to learn and use correctly [1, 5, 56]: they support writing correct and efficient programs, but they are still hard to understand [35, 58] for a number of reasons. First, their design must be conservative, banning not just all programs that are *actually* unsafe, but a large number of correct programs as well. To programmers, this manifests as a large number of *false positive* errors or warnings about problems that will never arise in practice. For example, Rust's version of

*Also with Australian National University.

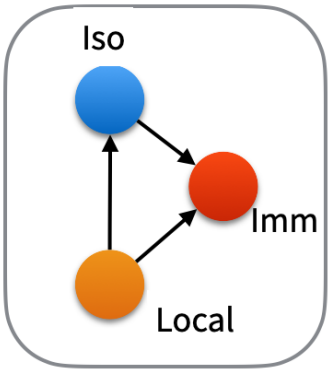
FTfJP '24, September 20, 2024, Vienna, Austria

ownership types [36] bans even such common idioms such as circular or doubly-linked lists. Second, programmers typically have to annotate their programs to give the ownership and capability checkers the information they need — so rather than declaring an input stream "`in`", programmers need to write complex expressions such as "`in : &mut InStream<'a>`" (where "`&mut`" indicates that "`in`" is a mutable reference, "`InStream`" indicates that "`in`" refers to an input stream, and "`<'a>`" is a lifetime (aka ownership) parameter indicating the originating scope of the input stream. Finally, these ownership annotations are required throughout the program, even if only a small part is actually concurrent, or is otherwise likely to cause critical errors — in Rust, an inflight entertainment system would have to be engineered to the same level of quality as a critical flight control system, even though the risks and requirements for each system are very different.

2.2 Dala: a simple concurrent language

Dala is a simple concurrent object-oriented programming language [26, 27] based on *concurrent dynamic ownership* [28, 51]. Dala is based on dividing objects in programs into one of three core *ownership capabilities*:

Imm**utable* values can be shared freely within and between concurrent threads but cannot be updated; ***Iso**lated* objects can be updated and transferred between threads but can only have one unique reference; and ***Local objects that can be accessed, updated, and shared only within a single thread.



In Dala, however, these capabilities do not stand alone: rather they form an ownership hierarchy (**Imm** < **Iso** < **Local**) that constrains interobject references: an immutable object can only refer to other immutable objects; an isolated object can refer to other isolated or immutable objects; and a local object can refer to immutable objects, isolated objects, and other local objects (see right). The hierarchy will be enforced by ownership checks integrated into the programming language: unintentional (buggy) or intentional (malicious) attempts to subvert the ownership model must be prevented by the language implementation (not the user). Enforcing the capability ownership hierarchy will ensure concurrent safety via data-race freedom, because threads can only communicate by reading immutable objects (which is always safe, because immutable objects never change) or by transferring isolated objects between threads (which is safe because isolated objects can only be accessed by one thread at any given instant).

2.3 Dafny

Dafny [48] is a verification-aware strongly typed programming language (with local type inference, objects, and algebraic data types) first developed at Microsoft Research (MSR) [49] and currently supported by the Amazon Automated Reasoning research group [2]. Dafny's toolchain includes translators for generating executable code in different target languages such as C#, Java, JavaScript, Go and Python [20]. But the distinguishing feature of Dafny is that it supports code verification via design by contract. For that, it follows the framework of Floyd-Hoare-style [32] program verification with preconditions, postconditions, loop invariants and other high-level formal proof synthesis features such as pure functions, predicates, lemmas, and automated proof by induction. Dafny employs explicit dynamic frames to determine which assertions may be invalidated by imperative updates: imperative methods must be annotated `writes S...` to gain permission to write to objects in the set `S` (or occasionally, `writes S`f...` to write only to field `f` of the objects in `S`).

To develop a verified program, developers write Dafny code along with the specifications and annotations to reason about the correctness of their code. While coding in Visual Studio, the Dafny static program verifier instantly verifies the functional correctness based on developers defined specifications and annotations. Correctness means the absence of any runtime errors with respect to the formal specifications, which means that the code does what the developers specify it to do. In order to confirm that the developer specifications holds, the Dafny program verifier first transforms the code into an intermediate verification representation (IVR) known as Boogie IVR [39] language that expresses the verification conditions into predicate calculus [43]. Next, an SMT solver tries to prove the verification conditions. In this phase, the Dafny verifier is backed by an advanced SMT solver known as the Z3 theorem prover [21]. The validity of these verification conditions implies the correctness of the code under consideration [43]. Sometimes, however, the Z3 solver cannot automatically reach the required proof even though such proof exists. In such cases, developer intervention is required to give more context in the specifications by writing auxiliary verification conditions in the form of functions, predicates, and lemmas.

In recent years, Dafny has had major successes: Microsoft used Dafny to formally verify security libraries and kernels [23, 37], distributed systems [30], and concurrent programs [31]; Intel is developing its hardware encryption library using Dafny [59]; ConsenSys successfully applied Dafny for their Ethereum Virtual Machine (EVM) verification [8]; Amazon implemented the Amazon Web Service (AWS) authorization and encryption logic in Dafny and deployed the Dafny-generated Java code into production [11, 17].

We can compare Dafny with tools more commonly used to verify programming language designs such as REDEX [24] or Coq [3, 18]. REDEX is a testing and exploration tool: designers can express the semantics of a whole language, and test and explore those semantics, but REDEX itself does not make a claim to formal validation. On the other hand, as a higher-order dependently-typed total functional language, designers can use Coq to both specify and prove entire languages — but constructing such a explicit proof can take a large amount of time, and an even larger amount of expertise. Superficially, Dafny's validation claims are much weaker than Coq's, as Dafny's implementation (verification condition generator, SMT solver) is not itself verified. While a proof in Coq is explicit in the syntax and must be manupiulated by the programmer, such a "proof" as Dafny

constructs is implicit: users write specifications and programs, but it's Dafny's job to validate them.

3 Modelling Dala with Dafny

3.1 Dala Objects

The core of our model of Dala is the Dafny class `Object` (note that while **`object`** is a reserved word in Dafny, `Object` is not — neither is `Class`)). Class `Object` models the essentials of a Dala object in the heap:

```
class Object {
  const kind : Kind;
  const fieldKinds : map<string,Kind>;
  var fields : map<string,Object>;
```

The key fields of an `Object` are the `kind`, the `fieldKinds`, and the object's `fields`.

The Dala model is fundamentally untyped, in that it is not set up to track e.g. the differences between a Point and a Rectangle class. Rather the `kind` field captures the object's ownership capability: Immutable (`Imm`), Isolated (`Iso`), or Local (`Mut`, originally mutable).

```
datatype Kind = Imm | Iso | Mut
```

We treat Dala's ownership capabilities more like meta-classes or meta-types rather that traditional types: they are close to what are often called "reference capabilities" [7, 10] but apply to each individual object rather than each reference to an object. In many languages capabilities manifest as additional keywords or attributes attached to variables that control how those variables may be used — i.e. how they relate to heap topologies and their evolution. In the Dafny code example above, the difference between **`const`** and **`var`** could potentially be considered different kinds of reference capability.

In Dala, then, an object's `kind` is the kind of object (ownership capability) that the object is. `fields` models object' fields, as a map between each field name (as a string) and the `Object` stored in that field, and the `fieldKinds` likewise gives the potential kind for each field.

Dafny **`map`**s are immutable data structures — only Dafny **`class`** instances are mutable, and then only **`var`** fields of classes; **`const`** fields are, well, constant. What this means is that while the values stored by Dala objects' fields can change (or rather, while our model permits them to change) the kinds of each field of each instance of that Dalia class are fixed.

The `Object` class contains a number of helper functions, such as `outgoing()`, which returns all outgoing references from an object, abstracting away field names; and `fieldNames()` which returns the names of fields, without their values.

Finally, the class contains a Dafny invariant, represented as a Dafny predicate conventionally called `Valid()`, which ensures the validity of each individual object. This predicate is typically defined as a conjunction of smaller invariants that must usually be adjusted to suit the particular heap being modelled. A minimal validity predicate would be something like this:

```
predicate Valid()
reads this`fields
  { AllFieldsAreDeclared() ∧ AllFieldsConsistentWithDclrn() }
```

saying that all fields (entries in the `fields`) must have a corresponding entry in the `fieldKinds`, and that the kind of object stored in a field must match the kind expected for that field. These are defined by auxiliary predicates:

```
predicate AllFieldsAreDeclared()
  reads this`fields
    { fields.Keys ⊆ fieldKinds.Keys }

predicate AllFieldsConsistentWithDclrn()
  requires AllFieldsAreDeclared()
  reads this`fields
    { ∀n <- fields :: fieldKinds[n] == fields[n].kind }
```

3.2 Dala Heaps

The other main structure in Dala is the heap itself, modelled by the eponymous `Heap` class. The Heap class looks trivial — a Dala heap is just a set of Dala objects:

```
class Heap {
  var objects : set<Object>
```

The full Heap class contains many auxiliary functions and invariants to capture the non-local structure of the heap. Thus there are corresponding class invariant validity predicates:

```
predicate Valid()
  reads this`objects, objects
    { ObjectsAreValid(objects) ∧ OutgoingRefsInThisHeap(obje
```

where `ObjectsAreValid(objects)` asserts each object's own `Valid()` predicate; and `OutgoingRefsInThisHeap` checks that the heap is closed, in the sense that there are no references to objects somehow "outside" the heap:

```
predicate ObjectsAreValid(os : set<Object>)
  reads os
    { (∀ o <- os :: o.Valid()) }

predicate OutgoingRefsInThisHeap(os : set<Object>)
  reads this`objects, objects, os
    { (∀ o <- os :: o.outgoing() ⊆ objects) }
```

The validity predicate will typically be invoked in method preconditions — Dafny does not assert class invariants automatically, rather programmers must follow conventions [55].

Unlike individual objects, Dala models often need to control the evolution of the heap, and so the class also declares an additional "two-state" predicate to capture the history constraint [40]. Unlike normal (aka "one-state") predicates, two-state predicates have access both to current values (unmarked) and values at the start of the containing method call (marked "**`old`**"). As such, two-state predicates can only be invoked from "two-state contexts", such as within an **`ensures`** clause or an assertion in a method body, where there are two states that can be compared.

```
twostate predicate Valid2()
  reads this`objects, objects, objects`fields
  ensures Valid2() ⟹ Valid()
    { Valid() ∧ HeapObjectsAreMonotonic() }
```

Here, for example, we can require that heap objects are monotonic — i.e that objects are never actually removed from the heap. (Note that heap models that e.g. wish to model explicit memory deallocation probably would not include this invariant).

```
twostate predicate HeapObjectsAreMonotonic()
  reads this`objects
    { old(objects) ⊆ objects }
```

Other utility functions are then provided as methods outside the class. For example: `edges(objects)` transforms a set of objects — often the `objects` field of a heap — into an adjacency set, i.e. a set of (*from*, *name*, *to*) triples as an alternative representation of the whole graph. Due to the idiosyncrasies of Dafny's verification, it turned out easier to generate the edge-list representation from the set-of-objects representation, rather than the other way around, or by maintaining both representations and continually assuring an overanxious Dafny, at pretty much every line of code, that the two were in sync.

3.3 Dala Ownership Hierarchy

We need to encode the ownership capability hierarchy as a Dafny predicate, that determines whether an object of kind `f` (from) can point to an object of kind `t` (to). (Note Dafny's alternative prefix syntax for repeated Boolean conjunctions.)

```
predicate DalaRefOK(f : Kind, t : Kind)
  {
    ∨ t.Imm? //anything can point to Imm
    ∨ f.Mut? //mut can point to anything
    ∨ (f.Iso? ∧ t.Iso?) //iso can point to Iso (and Imm, above)
  }
```

Finally we need to work this constraint throughout the heap model as a whole. We expand on the Object class invariant to ensure that all the field values — the outgoing references — confirm to the Dala model:

```
predicate Valid()
  reads this`fields
    {
    ∧ AllFieldsAreDeclared()
    ∧ AllFieldsConsistentWithDclrn()
    ∧ AllOutgoingRefsDala()
    }
```

based on an auxiliary predicate that quantifies over all the fields in the object, and requires that the object's `kind` is compatible with the kind of the contents of each field `fields[n].kind`:

```
predicate AllOutgoingRefsDala()
  reads this`fields
    { ∀n <- fields :: DalaRefOK(kind, fields[n].kind) }
```

3.4 Modelling Operations

Given we are taking a lightweight approach, we do not wish to produce a full operational semantics for Dala (or, likewise, require that programmers would have to be able to read a complete semantics for a language e.g. to find out what an "immutable" or a "unique" object means). Rather, we model the interface between the operational semantics or an interpreter as a series of Dafny methods on the `Heap` class. The code of these methods are trivial: the tricks come in writing the necessary pre- and post-conditions so that Dafny can verify that they maintain the invariants structuring the heap, i.e. the predicates installed as the one-state class invariant `Valid()` and two-state history constraint `Valid2()` defined within the Heap class.

For example, here is the code of the interface method for adding a new object within the heap:

```
method fAddObject(nu : Object)
  { objects := objects + {nu}; }
```

so far so good: add the `nu` object into the set of objects on the heap. Dafny abstracts methods as their pre-conditions (**`requires`**) and post-conditions (**`ensures`**) so the actual method header must include these specifications:

```
method fAddObject(nu : Object)
    requires Valid()
    requires nu.Valid()
    requires nu.size() == 0
    requires nu ∉objects
    modifies this`objects
    ensures Valid2()
    ensures unchanged(nu)
    ensures objects == old(objects) + {nu};
```

so that the method can rely on the class invariants of both heap and new object; check that the new object has no fields (to which objects would they refer?); that the new object isn't already in the heap; that it will modify the list of objects in the heap; and that when completed the history constraint will be maintained (which also maintains the class invariant); that the new object itself is not modified; and that the objects in the heap now consist of all the objects previously in the heap, plus the object just added![1]. We list these here not because the details are really important — they're not — but rather to give an idea of the amount of effort required to specify something that simple.

Finally, `fAddObject` requires another three lines within the method body to enable Dafny to verify the invariants:

```
  assert edges(objects) == edges(objects + {nu}) ==
old(edges(objects));
```

The issue here is that Dafny's axioms for the built-in collections (here sets) do not cover extensionality: this assertion provides a hint to the verifier that adding an object with no outgoing edges doesn't change the edges in the heap.

Table 1 below shows the core operations we modelled to capture basic heap semantics.

Table 1: Core API for basic Dala model.

`method` `fAddObject(nu : Object)` — add a new object to the heap.

`predicate` `fExists(o : Object, n : string)` — true if field f has a value (is not null)

`function` `fRead(o : Object, n : string) : (r : Object)` — read field value

`method` `fInitialise(o : Object, f : string, t : Object)` — initialise a null field

`method` `fNullify(o : Object, f : string)` — remove a field value

`method` `dynMove(o : Object, n : string, f : Object, m : string)` **`returns`** `(r : Status)` — Dala dynamically checked "move" `o.n <- f.m;`

`method` `dynCopy(o : Object, n : string, f : Object, m : string)` **`returns`** `(r : Status)` — Dala dynamically checked "copy" `o.n := f.m;`

[1] Whew!

Then, perhaps surprisingly, we need only a few additional annotations for Dafny to be able to verify that the operations of the language maintain the heap structures so that the ownership capability hierarchy is preserved. Much of the work is done by `fInitialise`, which writes a value into a waiting empty field — in fact, really by only three preconditions (or really by two actual preconditions and the invariant) of `fInitialise`:

```
method {:timeLimit 60}
  fInitialise(o : Object, f : string, t : Object)
    requires Valid()
    requires o.fieldKinds[f] == t.kind
    requires DalaRefOK(o.kind,t.kind)
    ...
  {
```

The first precondition is just the class invariant (that's the precondition that was already there). By extending the the class invariant to include `DalaRefOK`, Dafny automatically assumes the object capability hierarchy will be maintained upon entry to the method; and because it is also incorporated in the history constraint in the postcondition, the method itself must also maintain the hierarchy. The second precondition requires that the kind of the object about to be assigned to a field is the kind of the object expected by that field (Dala kinds are non-polymorphic and invariant). The third precondition requires that a references from the source object to the target will also maintain the object capability hierarchy. So we have that the existing references maintain the hierarchy in the pre-state; now the new reference also maintains the hierarchy; so all the references in the heap maintain the hierarchy in the post-state and initialise verifies.

3.5 Immutability

Many programming languages support one kind or another of immutable objects (also known as value objects, frozen objects, or just values) which cannot change. Immutability can be modelled straightforwardly in Dafny, and the predicates can be incorporated into any definition that requires immutability. We will use the `kinds` of objects defined above to distinguish between immutable vs mutable objects. Then we must give some semantics to the `Imm` kind of immutable objects. What makes an object immutable is that "naught changeth thee" [57], i.e. that the object is the same before any interaction as it is afterwards. We can encode this as a two-state predicate that can be conjoined into a Dala Heap's `Valid2()` predicate:

```
twostate predicate AllImmutablesAreImmutable()
  reads this`objects, objects, objects`fields
    {
      ∀o <- (objects * old(objects)) :: o.kind.Imm? ⟹
(o.fields == old(o.fields))
    }
```

which means for all objects existing both in the pre- and post-states, if an object's kind is immutable, then all its fields in the **old** pre-state must equal those in the current post-state. Note that since Dafny maps are themselves immutable values, we can compare entire maps to one another with simple equality.

3.6 Uniqueness

Another feature enjoyed by many of the coming generation of imperative languages is uniqueness: that there is at most one reference to any unique object [33]. This is also relatively straightforward to model in Dafny, and so supports Dala's unique objects — mostly known as Isolated Objects, or Isolates, thus kind `Iso`. We need to express the invariant that a unique Iso object has no more than one incoming reference. We can write an auxiliary function to extract all the (should-be) unique-kind objects — of note are three postconditions that were required to facilitate verification: that every object in the result is an Iso; that all the Isos in the input are in the output; and that if the input is empty, so is the output.

```
function justTheIsos(os : set<Object>)
                : (rs : set<Object>)
  reads os
  ensures ∀r <- rs :: r.kind.Iso?
  ensures ∀o <- os :: o.kind.Iso? ⟹ o in rs
  ensures (os == {}) ⟹ (rs == {})
    {
      set o <- os | o.kind.Iso?
    }
```

Using this function, and another function of around the same complexity that, given an object `i` and set of edges `edges`, returns the number of edges incident on the object — i.e. that object's reference count — we can define an invariant that all Isos have no more than one incoming reference:

```
predicate IsosAreUnique(os : set<Object>)
reads os
  {
    var edges := edges(os);
    var isos := justTheIsos(os);
    ∀i <- isos :: refCntEdges(i, edges) ≤ 1
  }
```

This is the easy part. The harder part is that every operation that might modify the Dala heap must be verified by Dafny, and maintaining the `IsosAreUnique` predicate requires several additional assertions and a lemma. We don't have the space to go over this in detail: but here is the imperative core of the `fInitialise` interface method that assigns a new object to an already-null field, and the necessary additional verification assertions:

```
1   method {:timeLimit 60} fInitialise(o : Object, f : string,
2     {
3       assert t.kind.Iso? ⟹ refCntEdges(t,edges) ==
    0;
4       o.fields := o.fields[f := t];
5       assert ObjectsAreValid({o});
6       assert edges(objects) == old(edges(objects)) + {Edge(o,f
7       assert (o ≠ t) ⟹ incomingEdges(t,{Edge(t,f,o)}) ≠
    {Edge(o,f,t)};
8       assert incomingEdges(t,{Edge(o,f,t)}) == {Edge(o,f,t)};
9       RefCountDistOverDisjointEdges(justTheIsos(edges), old(ed
10    }
```

Note in particular:
line 1, `{:timeLimit 60}` tells Dafny to spend more time to verify this method;
line 3 we reassure Dafny that if the new field value is supposed to be unique, then it currently has no incoming references from other fields. (If the object has just been read from a field, that field must already have been nullified, thus dropping the reference); This assertion acts as a hint to the verifier.
line 4 does all the work that needs to be done;
line 5 reminds Dafny that all objects should still be valid individually;
line 6 tells Dafny how the edges in the heap graph have changed as a result of line 4;

lines 7 & 8 explain consequences of the changes made in line 4 & described in 6;
finally, line 9 invokes a lemma expressing the graph-theoretic property that the reference counts to a set of objects distributes over two disjoint sets of edges involving those objects.

3.7 Move vs Copy Semantics

Since 2011, C++ has distinguished between what it calls *copy semantics* — essentially the situation from earlier versions of C++, where an assignment copies memory from rvalue to lvalue, leaving both rvalue and lvalue accessible afterwards, and *move semantics* where an assignment "moves" data from rvalue to lvalue, meaning that the rvalue is conceptually nullified by the move, and so is no longer accessible [6]. (In practice, the implementation of copy and move are identical: the only difference is in the continued accessibility of the rvalue afterwards [29]). Rust, famously, is built around the same distinction, and Dala also uses the distinction with Isos (copying a reference to an Iso means the Iso is no longer unique, thus breaking the heap structure invariant).

It is also straightforward to model these two different operations in our Dafny model. Here is the imperative core of the method that will `move` the contents of field `m` from object `f` into field `n` of object `o` — we retrieve the value from the source field, nullify the destination if necessary, nullify the source, and eventually write the value back into the destination field.

```
method move(o : Object, n : string,
            f : Object, m : string)
{
  var value := f.fields[m];
  if (fExists(o,n)) {fNullify(o,n);}
  fNullify(f,m);
  fInitialise(o,n,value);
```

Verification is straightforward unless we're moving an Iso: if we are, we need to reassure Dafny that reference counts distribute over disjoint sets of edges yet again, and other similar things.

Copy semantics, although in some sense it "feels" like a more complex operation —- and copying things is certainly more complex than moving things in the physical world [41] — has an even simpler definition and simpler proof. Still omitting the pre- and post-conditions, here's the entire body of `copy`, not just the imperative core:

```
method copy(o : Object, n : string,
            f : Object, m : string)
{
  var value := f.fields[m];
  if (fExists(o,n)) {fNullify(o,n);}
  fInitialise(o,n,value);
```

We have three of the same statements as for `move` — omitting the line that nullifies the source. There's also one time limit directive required. The key difference, it turns out, doesn't turn directly on the nullification itself, but rather on the fact that `move` can be used to move references to unique Iso objects, while `copy` cannot. Copy has a precondition that prevents it from being applied to Iso objects (by preventing it from being applied to fields whose `fieldKind` is Iso: we know any other kind of field does not contain an Iso), so the whole question can be sidestepped, whereas move must ensure the heap structure invariants around Iso objects are maintained.

4 Conclusion

Formal methods and tools are becoming more popular in software engineering practice, and accordingly more common in programming language design. We have described our experience in attempting to increase our assurance in the design of the Dala language, by modelling the key parts of Dala's design and then verifying that model in Dafny.

So far, this approach has been fruitful: we have been able to model a range of heap constraints in Dafny, and then express and verify that those constraints are maintained. The key factors supporting this outcome were the Dafny tool, which is now sufficiently mature to be used at this scale, and the necessary time and effort to model the heap structures Dafny (easy), express the invariants and operations permitted on those heaps (relatively easy), and then coax Dafny to admit — i.e. to satisfy itself, to prove — that the invariants are maintained (more difficult, but by no means insurmountable). We hope to continue with this work, both to integrate formal verification ever more tightly into programming language design, and to investigate how tools such as Dafny can best support this approach. In this sense, using Dafny has increased our confidence in Dala's design: the next steps are to extend the formal model to incorporate a gradual type system, and to implement a concurrent testbed to experiment with writing actual Dala programs.

The main disadvantage we found from using Dafny is that verification always takes longer than any estimate, although when working with Dafny, we often felt that just "one more assertion" would be enough to verify our entire model. We consider this comes from the auto-active / autonomic / opaque style of verification that is intrinsic to Dafny. Dafny verification acts as a intermittent positive reinforcement on a variable ratio schedule, like a slot machine, with the same addictive qualities [54].

Acknowledgments

Thanks to Rustan Leino and James Wilcox, to Lindsay Groves, longtime custodian of Formal Methods at VUW, to the many anonymous reviewers of slightly worse versions of this paper. This work is supported in part by the Royal Society of New Zealand Te Apārangi Marsden Fund Te Pūtea Rangahau a Marsden grants VUWU1815 and CRP2101, an Amazon Research Award, and Agoric Inc.

References

[1] Parastoo Abtahi and Griffin Dietz. 2020. Learning Rust: How Experienced Programmers Leverage Resources to Learn a New Programming Language. In *CHI Extended Abstracts*. 1–8.
[2] Amazon. 2023. Automated reasoning. https://www.amazon.science/research-areas/automated-reasoning. [Online], [Accessed: 2023-04-20].
[3] Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development*.
[4] Andrew P. Black, Kim B. Bruce, Michael Homer, and James Noble. 2012. Grace: the absence of (inessential) difficulty. In *Onward! '12: Proceedings 12th SIGPLAN Symp. on New Ideas in Programming and Reflections on Software*. ACM, New York, NY, 85–98.
[5] David Blaser. 2019. Simple Explanation of Complex Lifetime Errors in Rust. (2019). ETH Zürich.
[6] 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
[7] 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

[8] Franck Cassez, Joanne Fuller, Milad K. Ghale, David J. Pearce, and Horacio Mijail Anton Quiles. 2023. Formal and Executable Semantics of the Ethereum Virtual Machine in Dafny. In *Formal Methods - 25th International Symposium, FM 2023, Lübeck, Germany, March 6-10, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 14000)*, Marsha Chechik, Joost-Pieter Katoen, and Martin Leucker (Eds.). Springer, 571–583.

[9] Elias Castegren and Tobias Wrigstad. 2016. Reference Capabilities for Concurrency Control. In *ECOOP*.

[10] Elias Castegren and Tobias Wrigstad. 2016. *Reference Capabilities for Trait Based Reuse and Concurrency Control*. Technical Report 2016-007.

[11] Aleksandar Chakarov, Aleksandr Fedchin, Zvonimir Rakamaric, and Neha Rungta. 2022. Better Counterexamples for Dafny. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 404–411. https://doi.org/10.1007/978-3-030-99524-9_23

[12] David Chisnall, Matthew Parkinson, and Sylvan Clebsch. 2021. Project Verona. (2021). `www.microsoft.com/en-us/research/project/-project-verona`.

[13] David Clarke, John M. Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *OOPSLA*.

[14] Sylvan Clebsch et al. 2015. Deny capabilities for safe, fast actors. In *AGERE*. 1–12.

[15] Michael J. Coblenz, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine. 2020. Can advanced type systems be usable? An empirical study of ownership, assets, and typestate in Obsidian. *OOPSLA* (2020).

[16] CompCert. 2023. CompCert. https://github.com/AbsInt/CompCert. [Online], [Accessed: 2023-04-20].

[17] Byron Cook. 2018. Formal Reasoning About the Security of Amazon Web Services. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10981)*, Hana Chockler and Georg Weissenbacher (Eds.). Springer, 38–47. https://doi.org/10.1007/978-3-319-96145-3_3

[18] coq. 2023. The Coq Development Team. 2017. Coq, v.8.7. https://coq.inria.fr. [Online], [Accessed: 2023-04-20].

[19] Joseph W. Cutler, Michael Hicks, and Emina Torlak. 2024. Improving the Stability of Type Safety Proofs in Dafny. In *Dafny Workshop at POPL*.

[20] Dafny. 2023. dafny-lang. https://github.com/dafny-lang/dafny. [Online], [Accessed: 2023-04-20].

[21] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[22] Jan de Muijnck-Hughes and James Noble. 2024. Colouring Flags with Dafny & Idris. In *Dafny Workshop at POPL*.

[23] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2020. Simple High-Level Code For Cryptographic Arithmetic: With Proofs, Without Compromises. *ACM SIGOPS Oper. Syst. Rev.* 54, 1 (2020), 23–30. https://doi.org/10.1145/3421473.3421477

[24] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex* (1st ed.). The MIT Press.

[25] Kiko Fernandez-Reyes. 2021. *Abstractions to Control the Future*. Ph. D. Dissertation. Uppsala University.

[26] Kiko Fernandez-Reyes, James Noble, Tobias Wrigstad, et al. 2020. Dalarna: A Capability-Based Dynamic Language Design For Data Race Freedom. In *FTfJP*.

[27] Kiko Fernandez-Reyes, James Noble, Tobias Wrigstad, et al. 2021. Dala: A Simple Capability-Based Dynamic Language Design For Data Race-Freedom. In *Submitted*.

[28] Donald Gordon and James Noble. 2007. Dynamic Ownership in a Dynamic Language. In *DLS*.

[29] 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 Publisher: IEEE Press.

[30] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, Ethan L. Miller and Steven Hand (Eds.). ACM, 1–17. https://doi.org/10.1145/2815400.2815428

[31] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. 2014. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, Jason Flinn and Hank Levy (Eds.). USENIX Association, 165–181. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/hawblitzel

[32] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

[33] 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

[34] Vivian Hu. 2020. Rust Breaks into TIOBE Top 20 Most Popular Programming Languages. (June 2020). InfoQ.

[35] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2020. Safe Systems Programming in Rust: The Promise and the Challenge. *Communications of the ACM* (2020).

[36] Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language* (2nd ed.).

[37] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David A. Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: formal verification of an OS kernel.. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, Jeanna Neefe Matthews and Thomas E. Anderson (Eds.). ACM, 207–220. https://doi.org/10.1145/1629575.1629596

[38] Paul Krill. 2021. Microsoft forms Rust language team. (Feb. 2021). InfoWorld.

[39] Claire Le Goues, K Rustan M Leino, and Michał Moskal. 2011. The Boogie verification debugger. *SEFM, LNCS* 7041 (2011), 407–414.

[40] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. 2007. JML Reference Manual. (February 2007). Iowa State Univ. `www.jmlspecs.org`.

[41] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. 1993. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley.

[42] K Rustan M Leino. 2013. Developing verified programs with Dafny. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 1488–1490.

[43] K. Rustan M. Leino. 2017. Accessible Software Verification with Dafny. *IEEE Softw.* 34, 6 (2017), 94–97. https://doi.org/10.1109/MS.2017.4121212

[44] K. Rustan M. Leino. 2023. *Program Proofs*. MIT Press.

[45] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

[46] Mikael Mayer. 2023. How to use Dafny to prove type safety. In *Dafny Blog*. dafny.org/blog/2023/07/14/types-and-programming-languages

[47] Sean McLaughlin, Georges-Axel Jaloyan, Tongtong Xiang, and Florian Rabe. 2024. Enhancing Proof Stability. In *Dafny Workshop at POPL*.

[48] Microsoft. 2023. The Dafny Programming and Verification Language. https://dafny.org/. [Online], [Accessed: 2023-04-20].

[49] Microsoft. 2023. Microsoft Research. https://www.microsoft.com/en-us/research/. [Online], [Accessed: 2023-04-20].

[50] James Noble. 2024. Learn 'em Dafny. In *Dafny Workshop at POPL*.

[51] James Noble, David Clarke, and John Potter. 1999. Object Ownership for Dynamic Alias Protection. In *TOOLS*.

[52] James Noble, David Streader, Isaac Oscar Gariano, and Miniruwani Samarakoon. 2022. More Programming Than Programming: Teaching Formal Methods in a Software Engineering Programme. In *NASA Symposium on Formal Methods*.

[53] James Noble, Jan Vitek, and John Potter. 1998. Flexible Alias Protection. In *ECOOP*. 158–185.

[54] James Noble and Charles Weir. 2024. The Faultless Way of Programming: Principles, Patterns, Practices, and Peculiarities for Verification in Dafny. In *EuroPLoP*.

[55] Matthew Parkinson. 2007. Class Invariants: the end of the Road?. In *IWACO*.

[56] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding memory and thread safety practices and issues in real-world Rust programs. In *PLDI*. 763–779.

[57] Walter Chalmers Smith. 1867. *Immortal, Invisible*.

[58] Ryan James Spencer. 2020. Four Ways To Avoid The Wrath Of The Borrow Checker. (2020). justanotherdot.com.

[59] Zhenkun Yang, Wen Wang, Jeremy Casas, Pasquale Cocchini, and Jin Yang. 2023. Towards A Correct-by-Construction FHE Model. *IACR Cryptol. ePrint Arch.* (2023), 281.