Sound Invariant Checking Using Type Modifiers and Object Capabilities.

Authors omitted for double-blind review.

Unspecified Institution.

Abstract

In this paper we use pre existing language support for type modifiers and object capabilities to enable a system for sound runtime verification of invariants. Our system guarantees that class invariants hold for all objects involved in execution. Invariants are specified simply as methods whose execution is statically guaranteed to be deterministic and not access any externally mutable state. We automatically call such invariant methods only when objects are created or the state they refer to may have been mutated. Our design restricts the range of expressible invariants but improves upon the usability and performance of our system compared to prior work. In addition, we soundly support mutation, dynamic dispatch, exceptions, and non determinism, while requiring only a modest amount of annotation.

We present a case study showing that our system requires a lower annotation burden compared to Spec#, and performs orders of magnitude less runtime invariant checks compared to the widely used 'visible state semantics' protocols of D, Eiffel. We also formalise our approach and prove that such pre existing type modifier and object capability support is sufficient to ensure its soundness.

1 Introduction

Object oriented programming languages provide great flexibility through subtyping and dynamic dispatch: they allow code to be adapted and specialised to behave differently in different contexts. However this flexibility hampers code reasoning, since object behaviour is usually nearly completely unrestricted. This is further complicated with the support OO languages typically have for exceptions, memory mutation, and I/O.

Class invariants are an important concept when reasoning about software correctness. They can be presented as documentation, checked as part of static verification, or, as we do in this paper, monitored for violations using runtime verification. In our system, a class specifies its invariant by defining a boolean method called `invariant`. We say that an object's invariant holds when its `invariant` method would return `true`. We do this, like Dafny [44], to minimise special syntactic and type-system treatment of invariants, making them easier to understand for users. Whereas most other approaches treat invariants as a special annotation with its own syntax.

An *invariant protocol* [72] specifies when invariants need to be checked, and when they can be assumed; if such checks guarantee said assumptions, the protocol is sound. The two main sound invariant protocols present in literature are *visible state semantic* [51] and the *Boogie/Pack-Unpack methodology* [5]. The visible state semantics expect the invariants of receivers to hold before and after every public method call, and after constructors. Invariants are simply checked at all such points, thus this approach is obviously sound; however this can be incredibly inefficient, even in simple cases. In contrast, the pack/unpack methodology

marks all objects as either *packed* or *unpacked*, where a packed object is one whose invariant is expected to hold. In this approach, an object's invariant is checked only by the pack operation. In order for this to be sound, some form of aliasing and/or mutation control is necessary. For example, Spec#, which follows the pack/unpack methodology, uses a theorem prover, together with source code annotations. While Spec# can be used for full static verification, it conveniently allows invariant checks to be performed at runtime, whilst statically verifying aliasing, purity and other similar standard properties. This allows us to closely compare our approach with Spec#.

Instead of using automated theorem proving, it is becoming more popular to verify aliasing and immutability using a type system. For example, three languages: L42 [68, 67, 43, 35], Pony [22, 23], and the language of Gordon et. al. [38] use Type Modifiers (TMs) and Object Capabilities (OCs) to ensure safe and deterministic parallelism.[1] While studying those languages, we discovered an elegant way to enforce invariants.

We use the guarantees provided by these systems to ensure that that at all times, if an object is usable in execution, its invariant holds. What this means is that if you can do anything with an object, such as by using it as an argument/receiver of a method call, we know that the invariant of it, and all objects reachable from it, holds. In order to achieve this, we use TMs and OCs to restrict how the result of invariant methods may change, this is done by restricting I/O as well as what state the invariant can refer to and what can alias/mutate such state. We use these restrictions to reason as to when an object's invariant could have been violated, and when such object can next be used, we then inject a runtime check between these two points. See Section 3 for the exact details of our invariant protocol.

Example

Here we show an example illustrating our system in action. Suppose we have a `Cage` class which contains a `Hamster`; the `Cage` will move its `Hamster` along a path. We would like to ensure that the `Hamster` does not deviate from the path. We can express this as the invariant of `Cage`: the position of the `Cage`'s `Hamster` must be within the path (stored as a field of `Cage`).

```
class Point { Double x; Double y; Point(Double x, Double y) {..}
  @Override read method Bool equals(read Object that) {
    return that instanceof Point &&
      this.x == ((Point)that).x && this.y == ((Point)that).y; }
}
class Hamster {Point pos; //pos is imm by default
  Hamster(Point pos) {..}
}
class Cage {
  capsule Hamster h;
  List<Point> path; //path is imm by default
  Cage(capsule Hamster h, List<Point> path) {..}
  read method Bool invariant() {
    return this.path.contains(this.h.pos); }
  mut method Void move() {
    Int index = 1 + this.path.indexOf(this.h.pos));
```

---

[1] TMs are called *reference capabilities* in other works. We use the term TM here to not confuse them with object capabilities, another technique we also use in this paper.

```
89      this.moveTo(this.path.get(index % this.path.size())); }
90    mut method Void moveTo(Point p) { this.h.pos = p; }
91
92 }
```

Many verification approaches take advantage of the separation between primitive/value types and objects, since the former are immutable and do not support reference equality. However, our approach works in a pure OO setting without such a distinction. Hence we write all type names in **BoldTitleCase** to underline this. Note: to save space, here and in the rest of the paper we omit the bodies of constructors that simply initialise fields with the values of constructor parameters, but we show their signature in order to show any annotations.

We use the **read** annotation on `equals` to express that it does not modify either the receiver or the parameter. In **Cage** we use the **capsule** annotation to ensure that the **Hamster**'s *reachable object graph* (ROG) is fully under the control of the containing **Cage**. We annotated the `move` and `moveTo` methods with **mut**, since they modify their receivers ROG. The default annotation is always **imm**, thus **Cage**'s `path` field is a deeply immutable list of **Point**s. Our system performs runtime checks for the invariant at the end of **Cage**'s constructor, `moveTo` method, and after any update to one of its fields. The `moveTo` method is the only one that may (directly) break the **Cage**'s invariant. However, there is only a single occurrence of **this** and it is used to read the **h** field. We use the guarantees of TMs to ensure that no alias to **this** could be reachable from either **h** or the immutable **Point** parameter. Thus, the potentially broken **this** object is not visible while the **Hamster**'s position is updated. The invariant is checked at the end of the `moveTo` method, just before **this** would become visible again. This technique loosely corresponds to an implicit pack and unpack: we use **this** only to read the field value, then we work on its value while the invariant of **this** is not known to hold, finally we check the invariant before allowing the object to be used again.

Note: since only **Cage** has an invariant, only **Cage** has special restrictions, allowing the code for **Point** and **Hamster** to be unremarkable. This is not the case in Spec#: all code involved in verification needs to be designed with verification in mind [7].

Spec# Example

Here we show the previous example in Spec#, the system most similar to ours (see appendix B for a more detailed discussion about this solution):

```
122
123 // Note: assume everything is 'public'
124 class Point { double x; double y; Point(double x, double y) {..}
125    [Pure] bool Equal(double x, double y) {
126      return x == this.x && y == this.y; }
127 }
128 class Hamster{[Peer]Point pos;
129    Hamster([Captured]Point pos){..}
130 }
131 class Cage {
132    [Rep] Hamster h; [Rep, ElementsRep] List<Point> path;
133    Cage([Captured] Hamster h, [Captured] List<Point> path)
134      requires Owner.Same(Owner.ElementProxy(path), path); {
135        this.h = h; this.path = path; base(); }
136    invariant exists {int i in (0 : this.path.Count);
137      this.path[i].Equal(this.h.pos.x, this.h.pos.y) };
```

```
138    void Move() {
139      int i = 0;
140      while(i<path.Count && !path[i].Equal(h.pos.x,h.pos.y)){i++;}
141      expose(this) {this.h.pos = this.path[i%this.path.Count];}}
142
143  }
```

In both versions, we designed `Point` and `Hamster` in a general way, and not solely to be used by classes with an invariant, in particular `Point` is not an immutable class. However, doing this in Spec# proved difficult, in particular we were unable to override `Object.Equals`, or even define a usable `equals` method that takes a `Point`, as such we could not call either `List<Point>.Contains` or `List<Point>.IndexOf`.

Even with all of the above annotations, we still needed special care in creating `Cage`s:

```
150
151  List<Point> pl = new List<Point>{new Point(0,0),new Point(0,1)};
152  Owner.AssignSame(pl, Owner.ElementProxy(pl));
153  Cage c = new Cage(new Hamster(new Point(0, 0)), pl);
154
```

Whereas with our system we can simply write:

```
156
157  List<Point> pl = List.of(new Point(0, 0), new Point(0, 1));
158  Cage c = new Cage(new Hamster(new Point(0, 0)), pl);
159
```

In Spec# we had to add 10 different annotations, of 8 different kinds; some of which were quite involved. In comparison, our approach requires only 7 simple keywords, of 3 different kinds; however we needed to write a separate `moveTo` method, since we do not want to burden our language with extra constructs such as Spec#'s `expose`.

Summary

We have fully implemented our protocol in L42[2][3], we used this implementation to implement and test an interactive GUI involving a class with an invariant. On a test case with 5 objects with an invariant, our protocol performed only 77 invariant checks, whereas the visible state semantic invariant protocols of D and Eiffel perform 53 and 14 million checks (respectively). See Section 7 for an explanation of these result. We also compared with Spec#, whose invariant protocol performs the same number of checks as ours, however the annotation burden was almost 4 times higher than ours.

In this paper we argue that our protocol is not only more succinct than the pack/unpack approach, but is also easier and safer to use. Moreover, our approach deals with more scenarios than most prior work: we allow sound catching of invariant failures and also carefully handle non deterministic operations like I/O. Section 2 explains the *type modifier* and *object capability* support we use for this work. Section 3 explains the details of our invariant protocol, and section 4 formalises a language enforcing this protocol. Sections 5 and 6, respectively, explain and motivate how our protocol can handle invariants over immutable and encapsulated data. Section 7 presents our GUI case study and compares it against visible state semantics and Spec#. Sections 8 and 9 provide related work and conclusions.

Appendix A provides a proof that our invariant protocol is sound. Appendices B and C provide further case studies and comparisons against Spec#, D and Eiffel.

---

[2] A suitably anonymised, experimental version of L42, supporting the protocol described in this paper, together with the full code of our case studies, is available at `http://l42.is/EcoopArtifact.zip`.

[3] We also believe it would be easy to implement our protocol in Pony and Gordon et. al.'s language.

2 Type Modifiers and Object Capabilities

Reasoning about imperative object oriented (OO) programs is a non trivial task, made particularly difficult by mutation, aliasing, dynamic dispatch, I/O, and exceptions. There are many ways to perform such reasoning, here we use the type system to restrict, but not prevent such behaviour in order to be able to soundly enforce invariants with runtime verification (RV).

Type Modifiers (TMs)

TMs, as used in this paper, are a type system feature that allows reasoning about aliasing and mutation. Recently a new design for them has emerged that radically improves their usability; three different research languages are being independently developed relying on this new design: the language of Gordon et. al. [38], Pony [22, 23], and L42 [68, 67, 43, 35]. These projects are quite large: several million lines of code are written in Gordon et. al.'s language and are used by a large private Microsoft project; Pony and L42 have large libraries and are active open source projects. In particular the TMs of these languages are used to provide automatic and correct parallelism [38, 22, 23, 67].

While we focus on the specific TMs provided by L42, Pony, and Gordon et. al., type modifiers are a well known language mechanism [75, 11, 61, 22, 35, 38] that allow statically reasoning about mutability and aliasing properties of objects. With slightly different names and semantics, the four most common modifiers for references to objects are:

- Mutable (`mut`): the referenced object can be mutated, as in most imperative languages without modifiers. If all types are `mut`, there is no restriction on aliasing/mutation.
- Readonly (`read`): the referenced object cannot be mutated by such references, but there may be mutable aliases to such object, thus mutation can still be observed.
- Immutable (`imm`): the referenced object can never mutate. Like `read` references, one cannot mutate through an `imm` reference, however `imm` references also guarantee that the referenced object will not mutate through any other alias.
- Encapsulated (`capsule`): everything in the reachable object graph (ROG) of a capsule reference (including itself) is mutable only through that reference; however immutable references can be freely shared across capsule boundaries.

TMs are different to field or variable modifiers like Java's `final`: TMs apply to references, whereas `final` applies to fields themselves. Unlike a variable/field of a `read` type, a `final` variable/field cannot be reassigned, it always refers to the same object, however the variable/field can still be used to mutate the referenced object. On the other hand, an object cannot be mutated through a `read` reference, however a `read` variable can still be reassigned.[4]

Consider the following example usage of `mut`, `imm`, and `read`, where we can observe a change in `rp` caused by a mutation inside `mp`.

```
mut Point mp = new Point(1, 2);
mp.x = 3; // ok
imm Point ip = new Point(1, 2);
//ip.x = 3; // type error
read Point rp = mp; // ok, read is a common supertype of imm/mut
//rp.x = 3; // type error
mp.x = 5; // ok, now we can observe rp.x == 5
```

---

[4] In C, this is similar to the difference between `A* const` (like `final`) and `const A*` (like `read`), where `const A* const` is like `final read`.

```
227  ip = new Point(3, 5); // ok, ip is not final
228
```

There are several possible interpretations of the semantics of type modifiers. Here we assume the full/deep meaning [79, 66]:

- the objects in the ROG of an immutable object are immutable,
- a mutable field accessed from a **read** reference produces a **read** reference,
- no casting/promotion from **read** to **mut** is allowed.

There are many different existing techniques and type systems that handle the modifiers above [79, 21, 39, 38, 68]. The main progress in the last few years is with the flexibility of such type systems: where the programmer should use **imm** when representing immutable data and **mut** nearly everywhere else. The system will be able to transparently promote/recover [38, 22, 68] the type modifiers, adapting them to their use context. To see a glimpse of this flexibility, consider the following example:

```
240
241      mut Circle mc = new Circle(new Point(0, 0), 7);
242  capsule Circle cc = new Circle(new Point(0, 0), 7);
243      imm Circle ic = new Circle(new Point(0, 0), 7);
244
```

Here mc, cc, and ic are syntactically initialised with the same expression: **new Circle(..)**. The **new** expression returns a **mut**, so mc is obviously ok. Moreover, the expression does not use any **mut** local variables, thus the flexible TM system allows the **mut** result to be promoted to **capsule**, thus cc is ok. Additionally, a **capsule** can be implicitly converted to **imm**, thus ic is also ok. We want to emphasise that this is not a special feature of **new** expressions: any expression of a **mut** type that uses no free **mut** variables declared outside can be implicitly promoted to **capsule**/**imm**.[5] This is the main improvement on the flexibility of TMs in recent literature [67, 68, 38, 22, 23]. Former work [16, 14, 40, 69, 2], which eventually enabled the work of Gordon et. al., does not consider promotion and infers uniqueness/isolation/immutability only when starting from references that have been tracked with restrictive annotations along their whole lifetime. From a usability perspective, this improvement means that these TMs are opt-in: a programmer can write large sections of code mindlessly using **mut** types and be free to have rampant aliasing. Then, at a later stage, another programmer may still be able to encapsulate those data structures into an **imm** or **capsule** reference.

The **capsule** modifier (sometimes called isolated/**iso**) is possibly the one whose details differ the most in literature. Here we refer to the interpretation of [38], that introduced the concept of recovery/promotion. This concept is the basis for L42, Pony, and Gordon et. al.'s type systems [38, 67, 68, 67, 22, 23].

The capsule/isolated fields of Gordon et. al. and Pony rely on destructive reads [38, 22]: in order to read them, a new value (such as **null**) will be assigned to them. In contrast, L42 [67, 68] does not require such destructive reads, thus **capsule** fields can be accessed many times, and their content can be seen from outside; but only in controlled ways. Both Gordon et. al. and Pony restrict how **capsule** local variables can be used by changing the type they are seen as, however both allow the local variable to be 'consumed', allowing them to be used as normal capsule/isolated expressions, at the cost of being unable to use the variable again. L42 however uses a simpler approach where all accesses to **capsule** local variables consume them: they are expressed using linear/affine types [13], thus they can only be used once.

---

[5] This requires some restrictions on **read** fields not discussed in detail for lack of space.

Exceptions

In most languages exceptions may be thrown at any point; combined with mutation this complicates reasoning about the state of programs after exceptions are caught: if an exception was thrown whilst mutating an object, what state is that object in? Does its invariant hold? The concept of *strong exception safety* (SES) [1, 43] simplifies reasoning: if a **try**–**catch** block caught an exception, the state visible before execution of the **try** block is unchanged, and the exception object does not expose any object that was being mutated. L42 already enforces SES for unchecked exceptions.[6] L42 enforces SES using TMs in the following way:[7][8]

- Code inside a **try** block capturing unchecked exceptions is typed as if all **mut** variables declared outside of the block are **read**.

- Only **imm** objects may be thrown as unchecked exceptions.

This strategy does not restrict throwing exceptions, but only catching unchecked ones. SES allows us to soundly capture invariant failures as unchecked exceptions: the broken object is guaranteed to be garbage collectable when the exception is captured. For the purposes of soundly catching invariant failures, it would be sufficient to enforce SES only when capturing exceptions caused by such failures.

Object Capabilities (OCs)

OCs, which L42, Pony, and Gordon et. al.'s work have, are a widely used [54, 60, 42] programming style that allows associating resources with objects. When this style is respected, code that does not possess an alias to such an object cannot use its associated resource. Here, as in Gordon et. al.'s work, we use OCs to reason about determinism and I/O. To properly enforce this, the OC style needs to be respected while implementing the primitives of the standard library and when performing foreign function calls that could be non deterministic, such as operations that read from files or generate random numbers. Such operations would not be provided by static methods, but instead instance methods of classes whose instantiation is kept under control.

For example, in Java, **System.in** is a *capability object* that provides access to the standard input resource, however, as it is globally accessible it completely prevents reasoning about determinism.

In contrast, if Java were to respect the object capability style, the **main** method could take a **System** parameter, as in main(**mut** System s) {.. s.in.read() ..}. Calling methods on that **System** instance would be the only way to perform I/O; moreover, the only **System** instance would be the one created by the runtime system before calling **main**. This design has been explored by Joe-E [32]. OCs are typically not part of the type system nor do they require runtime checks or special support beyond that provided by a memory safe language. However, since L42 allows user code to perform foreign calls without going through a predefined standard library, its type system enforces the OC pattern over such calls:

- Foreign methods (which have not been whitelisted as deterministic) and methods whose names start with **#$** are *capability methods*.

- Constructors of classes declared as *capability classes* are also capability methods.

---

[6] This is needed to support safe parallelism. Pony takes a more drastic approach and does not support exceptions in the first place. We are not aware of how Gordon et. al. handles exceptions, however in order for it to have sound unobservable parallelism it must have some restrictions.

[7] Transactions are another way of enforcing strong exception safety, but they require specialized and costly run time support.

[8] A formal proof of why these restriction are sufficient is presented in the work of Lagorio [43].

313   ▬ Capability methods can only be called by other capability-methods or **mut**/**capsule**
314   methods of capability classes.
315   ▬ In L42 there is no **main** method, rather it has several main expressions; such expressions
316   can also call capability methods, thus they can instantiate capability objects and pass
317   them around to the rest of the program.

318   L42 expects capability methods to be used mostly internally by capability classes, whereas
319   user code would call normal methods on already existing capability objects.
320   For the purposes of invariant checking, we only care about the effects that methods could
321   have on the running program and heap. As such, *output* methods (such as a **print** method)
322   can be whitelisted as 'deterministic', provided they do not affect program execution, such as
323   by non deterministically throwing I/O errors.

Purity

325   TMs and OCs together statically guarantee that any method with only **read** or **imm** parameters
326   (including the receiver) is *pure*; we define pure as being deterministic and not mutating
327   existing memory. Such methods are pure because:
328   ▬ the ROG of the parameters (including **this**) is only accessible as **read** (or **imm**), thus it
329   cannot be mutated[9],
330   ▬ if a capability object is in the ROG of any of the arguments (including the receiver), then
331   it can only be accessed as **read**, preventing calling any non deterministic (capability)
332   methods,
333   ▬ no other preexisting objects are accessible (as L42 does not have global variables).[10]

3   Our Invariant Protocol

335   Our invariant protocol guarantees that the whole ROG of any object involved in execution
336   (formally, in a redex) is *valid*: if you can call methods on an object, calling **invariant** on it
337   is guaranteed to return **true** in a finite number of steps. However, calls to **invariant** that
338   are generated by our runtime monitoring (see below) can access the fields of a potentially
339   invalid **this**. This is necessary to allow for the **invariant** method to do its job: namely
340   distinguish between valid and invalid objects. However, as for any other method, calls to
341   **invariant** written explicitly by users are guaranteed to have a valid receiver.
342   For simplicity, in the following explanation and in our formalism we require receiv-
343   ers to always be specified explicitly, and require that the receivers of field accesses and
344   updates are always **this**; that is, all fields are instance private. We also do not allow
345   explicit constructor definitions, instead we assume constructors are of the standard form
346    $C(T_1 x_1, \ldots, T_n x_n)$ {**this**.f_1=x_1;…;**this**.f_n=x_n;}, where the fields of C are $T_1 f_1$; …; $T_n f_n$;.
347   This ensures that partially uninitialised (and likely invalid) objects are not passed around
348   or used. These restrictions only apply to our formalism; our code examples and the L42
349   implementation soundly relax these, see below for a discussion.

---

[9] This is even true in the concurrent environments of Pony and Gordon, since they ensure that no other
     thread/actor has access to a **mut**/**capsule** alias of **this**. Thus, since such methods do not write to
     memory accessible by another thread, nor read memory that could be mutated by another thread, they
     are atomic.

[10] If L42 did have static variables, getters and setters for them would be capability methods. Even
     allowing unrestricted access to **imm** static variables would prevent reasoning over determinism, due to
     the possibility of global variable updates; however constant/final globals of an **imm** type would not cause
     such problems.

Invariants

We require that all classes contain a `read method Bool invariant() {..}`, if no `invariant` method is present, a trivial one returning `true` will be assumed. As this method only takes a `read` parameter (the receiver), we can be sure that it is pure [11], as discussed in Section 2. The bodies of `invariant` methods are limited in their usage of `this`: `this` can only be used to access `imm` and `capsule` fields. This restriction ensures that an invalid `this` cannot be passed around. We prevent accessing `mut` fields since their ROG could be changed by unrelated code (see Section 5). Note that we do not require such fields to be `final`: when a field is updated, we simply check the invariant of the receiver of the update.

Capsule mutators

In order to allow complex mutations of objects with invariants we introduce the notion of *capsule mutator*. A *capsule mutator* can perform an arbitrarily complex mutation of the ROG of a capsule field. We use TMs to ensure that the object containing the capsule field is not usable whilst the fields ROG is mutated, and it's invariant is checked immediately afterwards.

Formally, *capsule mutators* are `mut` methods whose body accesses a `capsule` field mentioned in the invariant of the class containing the field. Capsule mutators must use `this` exactly once in their body, since fields are instance private, this will be to access the `capsule` field. Excluding the `mut` receiver, such methods cannot have any `mut` or `read` parameters, their return type must not be `mut`, and their `throws` clause must be empty.[12]

As capsule mutators use `this` only once, and have no `read` or `mut` parameters, `this` will not be accessible during execution. This is important, as it allows the invariant to be violated part way through the capsule mutator, but re established by the end. Preventing `mut` return types ensures that such methods cannot leak out a mutable alias to the `capsule` field, which could then be used to break the invariant. Note that these restrictions do not apply when the receiver of the field access is `capsule`, since we guarantee that the receiver is not in the ROG of any of its `capsule` fields, and hence it can never be seen afterwards.

Monitoring

The language runtime will insert automatic calls to `invariant`, if such a call returns `false`, an unchecked exception will be thrown. Such calls are inserted in the following points:

- After a constructor call, on the newly created object.
- After a field update, on the receiver.
- After a capsule mutator method returns, on the receiver of the method[13].

In Appendix A, we show that these checks, together with our aforementioned restrictions, are sufficient to ensure our guarantee that all objects involved in execution (except as part of an invariant check) are valid.

Relaxations

The above restrictions can be partially relaxed without breaking soundness, however this would not make the proof more interesting. In particular:

---

[11] If the invariant were not pure, it would be nearly impossible to ensure that it would return `true` at any point.

[12] To allow capsule mutators to leak checked exceptions, we would need check the invariant when such exceptions are leaked. However, this would make the runtime semantics of checked exceptions inconsistent with unchecked ones.

[13] The invariant is not checked if the call was terminated via an an unchecked exception, since strong exception safety guarantees the object will be unreachable anyway.

389   ▪   `invariant` methods can be allowed to call instance methods that in turn only use **this**
390   to read **imm** or **capsule**, or call other such instance methods. With this relaxation, the
391   semantics of `invariant` needs to be understood with the body of those methods inlined;
392   thus the semantics of the inlined code needs to be logically reinterpreted in the context of
393   `invariant`, where **this** may be invalid. In some sense, those inlined methods and field
394   accesses can be thought of as macro expanded, and hence are not dynamically dispatched.
395   Such inlining has been implemented in L42.
396   ▪   We could allow all fields to be public, however **capsule** fields, mentioned in the invariant
397   of their containing class, should not be accessible over a **mut** receiver other than **this**.
398   Even without this relaxation, however, getters and setters could be used to simulate
399   public fields.
400   ▪   Unrestricted readonly access to **capsule** fields can be allowed by automatically generated
401   getters of the form **read method read** C f() { **return this**.f; }. Such getters are
402   already a fundamental part of the L42 language.
403   ▪   Java style constructors could be allowed, provided that **this** is only used as the receiver
404   of field initialisations. L42 does not provide such constructors, but one can always write
405   a static factory method that behaves equivalently.
406   Both L42, and our formal language (see Section 4) do not have traditional subclassing, rather
407   all 'classes' are either interfaces (which only have abstract methods), or are final (which
408   cannot be subtyped). In a language with traditional subclassing, invariant methods would
409   implicitly start with a check that **super**.invariant() returns **true**. Note that invariant
410   checks would not be performed at the end of **super**(..) constructor calls, but only at the
411   end of **new** expressions, as happens in [30].

4   Formal Language Model

413   In order to model our system, we need to formalise an imperative object oriented language
414   with exceptions, object capabilities, and rich type system support for TMs and strong
415   exception safety. Formal models of the runtime semantics of such languages are simple, but
416   defining and proving, such a type system would require a paper of its own, and indeed many
417   such papers exist in literature [67, 68, 38, 22, 43]. Thus we are going to assume that we
418   already have an expressive and sound type system enforcing the properties we need, and
419   instead focus on invariant checking. We clearly list in Appendix A the assumptions we make
420   on such a type system, so that any language satisfying them, such as L42, can soundly
421   support our invariant protocol.
422   To keep our small step semantics as conventional as possible, we follow Pierce [64] and
423   Featherweight Java [41], and assume:
424   ▪   An implicit program/class table.
425   ▪   Memory, $\sigma : l \rightarrow C\{\overline{v}\}$, is a finite map from locations, l, to annotated tuples, $C\{\overline{v}\}$,
426   representing objects; where C is the class name and $\overline{v}$ are the field values. We use the
427   notation $\sigma[l.f = v]$ to update a field of an object, and $\sigma[l.f]$ to access one.
428   ▪   A main expression that is reduced in the context of such a memory and program.
429   ▪   A typing relation, $\Sigma; \Gamma \vdash e : T$, where the expression e can contain locations and free
430   variables. The types of locations are encoded in a memory environment, $\Sigma : l \rightarrow C$, while
431   the types of free variables are encoded in a variable environment, $\Gamma : x \rightarrow T$.
432   ▪   We use Σ^{σ} to trivially extract the corresponding Σ from a σ.
433   To encode object capabilities and I/O, we assume a special location c of class **Cap**. This
434   location would refer to an object whose fields model things like the content of files. In order

$$
\begin{array}{lll}
e & ::= & x \mid l \mid \texttt{true} \mid \texttt{false} \mid e.m(\overline{e}) \mid e.f \mid e.f = e \mid \texttt{new } C(\overline{e}) \mid \texttt{try \{}e_1\texttt{\} catch \{}e_2\texttt{\}} \quad \text{expression} \\
 & & \mid \texttt{M}(l\,;e_1\,;e_2) \mid \texttt{try}^\sigma\texttt{\{}e_1\texttt{\} catch \{}e_2\texttt{\}} \qquad\qquad\qquad\qquad \text{runtime expr.} \\
v & ::= & l \qquad\qquad \text{value} \\
\mathcal{E}_v & ::= & \square \mid \mathcal{E}_v.m(\overline{e}) \mid v.m(\overline{v}_1, \mathcal{E}_v, \overline{e}_2) \mid v.f = \mathcal{E}_v \qquad\qquad \text{evaluation context} \\
 & & \mid \texttt{new } C(\overline{v}_1, \mathcal{E}_v, \overline{e}_2) \mid \texttt{M}(l\,;\mathcal{E}_v\,;e) \mid \texttt{M}(l\,;v\,;\mathcal{E}_v) \mid \texttt{try}^\sigma\texttt{\{}\mathcal{E}_v\texttt{\} catch \{}e\texttt{\}} \\
\mathcal{E} & ::= & \square \mid \mathcal{E}.m(\overline{e}) \mid e.m(\overline{e}_1, \mathcal{E}, \overline{e}_2) \mid e.f = \mathcal{E} \mid \texttt{new } C(\overline{e}_1, \mathcal{E}, \overline{e}_2) \qquad \text{full context} \\
 & & \mid \texttt{M}(l\,;\mathcal{E}\,;e) \mid \texttt{M}(l\,;e\,;\mathcal{E}) \mid \texttt{try}^{\sigma?}\texttt{\{}\mathcal{E}\texttt{\} catch \{}e\texttt{\}} \mid \texttt{try}^{\sigma?}\texttt{\{}e\texttt{\} catch \{}\mathcal{E}\texttt{\}} \\
CD & ::= & \texttt{class } C \texttt{ implements } \overline{C}\texttt{\{}\overline{F}\,\overline{M}\texttt{\}} \mid \texttt{interface } C \texttt{ implements } \overline{C}\texttt{\{}\overline{M}\texttt{\}} \qquad \text{class declaration} \\
F & ::= & T\,f\texttt{;} \qquad\qquad \text{field} \\
M & ::= & \mu \texttt{ method } T\ \ m(\,T_1\,x_1, ..., T_n\,x_n\,)\ \ e? \qquad\qquad \text{method} \\
\mu & ::= & \texttt{mut} \mid \texttt{imm} \mid \texttt{capsule} \mid \texttt{read} \qquad\qquad \text{type modifier} \\
T & ::= & \mu\,C \qquad\qquad \text{type} \\
r_l & ::= & v.m(\overline{v}) \mid v.f \mid v_1.f = v_2 \mid \texttt{new } C(\overline{v}), \quad \text{where } l \in \{v, v_1, v_2, \overline{v}\} \qquad \text{redex containing } l \\
error & ::= & \mathcal{E}_v[\texttt{M}(l\,;v\,;\texttt{false})], \quad \text{where } \mathcal{E}_v \text{ not of form } \mathcal{E}'_v[\texttt{try}^{\sigma?}\texttt{\{}\mathcal{E}''_v\texttt{\} catch \{\_\}}] \qquad \text{validation error}
\end{array}
$$

■ **Figure 1** Grammar

to simplify our proof, we assume that:

- instances of `Cap` cannot be created with a `new` expression,
- all methods in the `Cap` class must require a `mut` receiver, and will mutate its ROG,
- `Cap` can only have `mut` fields, and
- `Cap`'s `invariant` method is defined to return `true`.

For simplicity, we do not formalise actual exception objects, rather we have *error*s, which correspond to expressions which are currently 'throwing' an exception; in this way there is no value associated with the *error*. Our L42 implementation instead models exceptions as throwing an `imm` value, formalising exceptions in this way would not cause any interesting variation of our proof.

Grammar

The detailed grammar is defined in Figure 1. Most of our expressions are standard. *Monitor expressions* are of the form $\texttt{M}(l\,;e_1\,;e_2)$, they are run time expressions and thus are not present in method bodies, rather they are generated by our reduction rules inside the main expression. Here, l refers to the object being monitored, e_1 is the expression which is being monitored, and e_2 denotes the evaluation of $l.\texttt{invariant()}$. If, at any point in execution, e_2 is `false`, then l's invariant failed to hold; such a monitor expression corresponds to the throwing of an unchecked exception.

In addition, our reduction rules will annotate `try` expressions with the original state of memory. This is used to model the guarantee of strong exception safety, that is, the annotated memory will not be mutated by executing the body of the `try`.

Well Formedness Criteria

We additionally restrict the grammar with the following well formedness criteria:

- `invariant` methods and capsule mutators satisfy the restrictions in Section 3.
- Field accesses and updates in methods are of the form $\texttt{this}.f$ or $\texttt{this}.f = e$, respectively.
- Field accesses and updates in the main expression are of the form $l.f$ or $l.f = e$, respectively.
- Locations that are preserved by `try` blocks are never monitored, that is, for $\texttt{try}^\sigma\texttt{\{}e\texttt{\} \_}$, if e is of the form $\mathcal{E}[\texttt{M}(l\,;\_\,;\_)]$, then $l \notin \sigma$.

Reduction rules

Our reduction rules are defined in Figure 2. They are pretty standard, except for our handling

(UPDATE)

$$\frac{}{\sigma|l.f \text{ = } v \to \sigma[l.f = v]|\texttt{M(}l\texttt{;}l\texttt{;}l.\texttt{invariant())}}$$

(NEW)

$$\frac{}{\sigma|\texttt{new }C(\overline{v}) \to \sigma, l \mapsto C\{\overline{v}\}|\texttt{M(}l\texttt{;}l\texttt{;}l.\texttt{invariant())}}$$

(MCALL)

$$\frac{\begin{array}{l} \sigma(l) = C\{\_\} \\ C.m = \mu\texttt{ method }T\ m\texttt{(}T_1\ x_1...T_n x_n\texttt{) } e \\ \text{if } \exists f \text{ such that } C.f = \texttt{capsule }\_, \mu = \texttt{mut}, \\ \quad f \text{ inside } C.m, \text{ and } f \text{ inside } C.\textit{invariant} \\ \quad \text{then } e' = \texttt{M(}l\texttt{;}e\texttt{;}l.\texttt{invariant())} \\ \text{otherwise } e' = e \end{array}}{\sigma|l.m\texttt{(}v_1,...,v_n\texttt{)} \to \sigma|e'[\texttt{this} = l, x_1 = v_1, ..., x_n = v_n]}$$

(MONITOR EXIT)

$$\frac{}{\sigma|\texttt{M(}l\texttt{;}v\texttt{;true)} \to \sigma|v}$$

(CTXV)

$$\frac{\sigma_0|e_0 \to \sigma_1|e_1}{\sigma_0|\mathcal{E}_v[e_0] \to \sigma_1|\mathcal{E}_v[e_1]}$$

(TRY ENTER)

$$\frac{}{\sigma|\texttt{try \{}e_1\texttt{\} catch \{}e_2\texttt{\}} \to \sigma|\texttt{try}^\sigma\texttt{\{}e_1\texttt{\} catch \{}e_2\texttt{\}}}$$

(TRY OK)

$$\frac{}{\sigma, \sigma'|\texttt{try}^\sigma\texttt{\{}v\texttt{\} catch \{\_\}} \to \sigma, \sigma'|v}$$

(TRY ERROR)

$$\frac{}{\sigma, \_|\texttt{try}^\sigma\texttt{\{}error\texttt{\} catch \{}e\texttt{\}} \to \sigma|e}$$

(ACCESS)

$$\frac{}{\sigma|l.f \to \sigma|\sigma[l.f]}$$

■ **Figure 2** Reduction rules

of monitor expressions. We define the relation *inside* as follows:

$f\ inside\ C.m$ iff $C.m = \_\texttt{ method}\_\ \mathcal{E}[\texttt{this.}f]$

Monitor expressions are added after all field updates, **new** expressions, and calls to capsule mutators. Monitor expressions are only a proof device, they need not be implemented directly as presented. For example, in L42 we implement them by statically injecting calls to **invariant** at the end of setters, factory methods and capsule mutators; this works as L42 does not have primitive expression forms for field updates and constructors, rather they are uniformly represented as method calls.

Our CTXV rule evaluates monitor expressions, $\texttt{M(}l\texttt{;}e_1\texttt{;}e_2\texttt{)}$, by first evaluating e_1 and then e_2. If e_2 evaluates to **true**, then the monitor succeeded, and will yield the result of e_1. If however e_2 evaluated to **false**, then the monitor failure will be caught by our TRY ERROR rule, as will any other uncaught monitor failure in e_1 or e_2.

Statement of Soundness

We define a deterministic reduction to mean that exactly one reduction is possible:

$\sigma_0|e_0 \Rightarrow \sigma_1|e_1$ iff $\{\sigma_1|e_1\} = \{\sigma|e, \text{ where } \sigma_0|e_0 \to \sigma|e\}$

An object is *valid* iff calling its **invariant** method would deterministically produce **true** in a finite number of steps, i.e. it does not evaluate to **false**, fail to terminate, or produce an *error*. We also require evaluating **invariant** to preserve existing memory (σ), however new objects (σ') can be created and freely mutated.

$valid(\sigma, l)$ iff $\sigma|l.\texttt{invariant()} \Rightarrow^+ \sigma, \sigma'|\texttt{true}.$

To allow the invariant method to be called on an invalid object, and access fields on such object, we define the set of trusted execution steps as the the call to **invariant** itself, and any field accesses inside its evaluation. Note that this only applies to single small step reductions, and not the entire evaluation of **invariant**.

$trusted(\mathcal{E}_v, r_l)$ iff

either $r_l = l.\texttt{invariant()}$ and $\mathcal{E}_v = \mathcal{E}'_v[\texttt{M}(l;v;\square)]$,

or $r_l = l.\texttt{f}$ and $\mathcal{E}_v = \mathcal{E}'_v[\texttt{M}(l;v;\mathcal{E}''_v)]$.

Finally, we define what it means to soundly enforce our invariant protocol: every object referenced by any untrusted redex is valid.

▶ **Theorem 1** (Soundness). *if* $c : \texttt{Cap}; \emptyset \vdash e : T$ *and* $c \mapsto \texttt{Cap}\{\_\}|e \to^+ \sigma|\mathcal{E}_v[r_l]$, *then either* $valid(\sigma, l)$ *or* $trusted(\mathcal{E}_v, r_l)$.

5 Invariants Over Immutable State

In this section we consider validation over fields of `imm` types. In the next section we detail our technique for **capsule** fields.

In the following code **Person** has a single immutable (non final) field `name`:

```
class Person {
  read method Bool invariant() { return !name.isEmpty(); }
  private String name;//the default modifier imm is applied here
  read method String name() { return this.name; }
  mut method String name(String name) { this.name = name; }
  Person(String name) { this.name = name; }
}
```

Person only has immutable fields and its constructor only uses **this** to initialise them. Note that the `name` field is not final, thus **Person** objects can change state during their lifetime. This means that the ROGs of all **Person**s fields are immutable, but **Person**s themselves may be mutable. We can easily enforce **Person**'s invariant by generating checks on the result of **this**.invariant(): immediately after each field update, and at the end of the constructor.

```
class Person { .. // Same as before
  mut method String name(String name) {
    this.name = name; // check after field update
    if (!this.invariant()) { throw new Error(...); }}
  Person(String name) {
    this.name = name; // check at end of constructor
    if (!this.invariant()) { throw new Error(...); }}
}
```

Such checks will be generated/injected, and not directly written by the programmer. If we were to relax (as in Rust), or even eliminate (as in Java), the support for TMs or OCs, the enforcement of our invariant protocol for the **Person** class would become harder, or even impossible.

Unrestricted use of non determinism

Allowing the `invariant` method to (indirectly) perform a non deterministic operation, such as by creating new capability objects, could break our guarantee that (manually) calling it always returns **true**. For example consider this simple and contrived (mis)use of person:

```
class EvilString extends String {
  @Override read method Bool isEmpty() {
    // Create a new capability object out of thin air
    return new Random().bool(); }
```

```
537  } ..
538  method mut Person createPersons(String name) {
539    // we can not be sure that name is not an EvilString
540    mut Person schrodinger = new Person(name); // exception here?
541    assert schrodinger.invariant(); // will this fail?
542  ..}
543
```

Despite the code for `Person.invariant` intuitively looking correct and deterministic, the above call to it is not. Obviously this breaks any reasoning and would make our protocol unsound. In particular, note how in the presence of dynamic class loading, we have no way of knowing what the type of `name` could be. Since our system allows non determinism only through capability objects, and restricts their creation, the above example would be prevented.

Allowing Internal Mutation Through Back Doors

Suppose we relax our rules by allowing interior mutability as in Rust and Javari, where sneaky mutation of the ROG of an 'immutable' object is allowed. Those back doors are usually motivated by performance reasons, however in [38] they briefly discuss how a few trusted language primitives can be used to perform caching and other needed optimisations, without the need for back doors.

Our example shows that such back doors can be used to break determinism of `invariant` methods, by allowing the invariant to store and read information about previous calls. In the following example we use `MagicCounter` as a back door to remotely break the invariant of `person` without any interaction with the `person` object itself:

```
561  class MagicCounter {
562    method Int increment(){
563      //Magic mutation through an imm receiver, equivalent to i++
564  }}
565  class NastyS extends String {..
566    MagicCounter evil = new MagicCounter(0);
567    @Override read method Bool isEmpty() {
568      return this.evil.increment() != 2; }
569  } ..
570  NastyS name = new NastyS("bob"); //TMs believe name's ROG is imm
571  Person person = new Person(name); // person is valid, counter=1
572  name.increment(); // counter == 2, person is now broken
573  person.invariant(); // returns false!, counter == 3
574  person.invariant(); // returns true, counter == 4
575
```

Strong Exception Safety

The ability to catch and recover from invariant failures is extremely useful as it allows programs to take corrective action. Since we represent invariant failures by throwing unchecked exceptions, programs can recover from them with a conventional **try**–**catch**. Due to the guarantees of strong exception safety, any object that has been mutated during a **try** block is now unreachable (as happens in alias burying [13]). In addition, since unchecked exceptions are immutable, they can not contain a **read** reference to any object (such as the **this** reference seen by `invariant` methods). These two properties ensure that an object whose invariant fails will be unreachable after the invariant failure has been captured. If instead we were to not enforce strong exception safety, an invalid object could be made

reachable:

```
mut Person bob = new Person("bob");
// Catch and ignore invariant failure:
try { bob.name(""); } catch (Error t) { } // ill typed in L42
assert bob.invariant(); // bob is invalid!
```

As you can see, recovering from an invariant failure in this way is unsound and would break our protocol.

6 Invariants over encapsulated state

Consider managing the shipment of items, where there is a maximum combined weight:

```
class ShippingList {
  capsule Items items;
  read method Bool invariant() {
    return this.items.weight() <= 300; }
  ShippingList(capsule Items items) {
    this.items = items;
    if (!this.invariant()) {throw Error(...);}}// injected check
  mut method Void addItem(Item item) {
    this.items.add(item);
    if (!this.invariant()) {throw Error(...);}}// injected check
}
```

To handle this class we just inject calls to `invariant` at the end of the constructor and the `addItem` method. This is safe since the `items` field is declared **capsule**. Relaxing our system to allow a **mut** modifier for the `items` field and the corresponding constructor parameter breaks the code: the cargo we received in the constructor may already be compromised:

```
mut Items items = ...;
mut ShippingList l = new ShippingList(items); // l is valid
items.addItem(new HeavyItem()); // l is now invalid!
```

As you can see it would be possible for external code with no knowledge of the `ShippingList` to mutate its items.[14]

Our restrictions on capsule mutators ensure that capsule fields are essentially an exclusive mutable reference. Removing these restrictions would break our invariant protocol. If we were to allow `x.items` to be seen as **mut**, where x is not **this**, then even if the `ShippingList` has full control at initialisation time, such control may be lost later, and code unaware of the `ShippingList` could break it:

```
mut ShippingList l = new ShippingList(new Items()); // l is ok
mut Items evilAlias = l.items // here l loses control
evilAlias.addItem(new HeavyItem()); // now l is invalid!
```

---

[14] Conventional ownership solves these problems by requiring a deep clone of all the data the constructor takes as input, as well as all exposed data (possibly through getters). In order to write correct library code in mainstream languages like Java and C++, defensive cloning [12] is needed. For performance reasons, this is hardly done in practice and is a continuous source of bugs and unexpected behaviour [12].

If we allowed a **mut** return type the following would be accepted:

```
mut method mut Items expose(C c) {return c.foo(this.items);}
```

Depending on dynamic dispatch, `c.foo()` may just be the identity function, thus we would get in the same situation as the former example.

Allowing **this** to be used more than once can also cause problems:

```
mut method imm Void multiThis(C c) {
  read Foo f = c.foo(this);
  this.items.add(new HeavyItem());
  f.hi(); } // Can 'this' be observed here?
```

If the former code were accepted, **this** may be reachable from **f**, thus `f.hi()` may observe an invalid object.

In order to ensure that a second reference to **this** is not reachable through the parameters, we only accept **imm** and **capsule** parameters. If we were however to accept a **read** parameter, as in the example below, we would be in the same situation as before, where **f** may contain a reference to **this**:

```
mut method imm Void addHeavy(read Foo f) {
  this.items.add(new HeavyItem())
  f.hi() } // Can 'this' be observed here?
...
mut ShippingList l = new ShippingList();
read Foo f = new Foo(l);
l.addHeavy(f); // We pass another reference to 'l' through f
```

7   GUI Case study

Here we show that we are able to verify classes with circular mutable object graphs, that interact with the real world using I/O. Our case study involves a GUI with containers (**SafeMovable**s) and **Button**s; the **SafeMovable** class has an invariant to ensure that its children are completely contained within it and do not overlap. The **Button**s move their **SafeMovable** when pressed. We have a **Widget** interface which provides methods to get **Widget**s' size and position as well as children (a list of **Widget**s). Both **SafeMovable**s and **Button**s implement **Widget**. Crucially, since the children of **SafeMovable** is a list of **Widget**s it can contain other **SafeMovable**s, and all queries to their size and position are dynamically dispatched, such queries are also used in **SafeMovable**'s invariant. Here we show a simplified version[15], where **SafeMovable** has just one **Button**, and certain sizes and positions are fixed. Note that **Widgets** is a class representing a mutable list of **mut Widget**s.

```
class SafeMovable implements Widget { capsule Box box;
  @Override read method Int left() { return this.box.l; }
  @Override read method Int top() { return this.box.t; }
  @Override read method Int width() { return 300; }
  @Override read method Int height() { return 300; }
```

---

[15] The full version, written in L42, which uses a different syntax, is available in our artifact at
http://l42.is/EcoopArtifact.zip

```
677   @Override read method read Widgets children() {
678     return this.box.c; }
679   @Override  mut method Void dispatch(Event e) {
680     for (Widget w:this.box.c) { w.dispatch(e); }}
681   read method Bool invariant() {..}
682   SafeMovable(capsule Widgets cs) { this.box = makeBox(c); }
683   static method capsule Box makeBox(capsule Widgets c) {
684     mut Box b = new Box(5, 5, cs);
685     b.c.add(new Button(0, 0, 10, 10, new MoveAction(b)));
686     return b; } //mut b is soundly promoted to capsule
687 }
688 class Box { Int l; Int t; mut Widgets c;
689   Box(Int l, Int t, mut Widgets c) {..}
690 }
691 class MoveAction implements Action { mut Box outer;
692   MoveAction(mut Box outer) { this.outer = outer; }
693   mut method Void process(Event event) { this.outer.l += 1; }
694 }
695 ..
696 //main expression; #$ is a capability method making a Gui object
697 Gui.#$().display(new SafeMovable(..));
698
```

As you can see, **Box**es encapsulate the state of the **SafeMovable**s that can change over time: `left`, `top`, and `children`. Also note how the ROG of **Box** is circular: since the **MoveAction**s inside **Button**s need a reference to the containing **Box** in order to move it. Even though the children of **SafeMovable**s are fully encapsulated, we can still easily dispatch events to them using `dispatch`. Once a **Button** receives an **Event** with a matching ID, it will call its **Action**'s `process` method.

Our example shows that the restrictions of TMs and OCs are flexible enough to encode interactive GUI programs, where widgets may circularly reference other widgets. In order to perform this case study we had to first implement a simple GUI Library in L42. This library uses object capabilities to draw the widgets on screen, as well as fetch and dispatch the events. Importantly, neither our application, nor the underlying GUI library require back doors into either our type modifier or capability system to function, demonstrating the practical usability of our restrictions.

The Invariant

SafeMovable is the only class in our GUI that has an invariant, our system automatically checks it in two places: the end of its constructor and the end of its `dispatch` method (is a capsule mutator). There are no other checks inserted since we never do a field update on a **SafeMovable**. The code for the invariant is just a couple of simple nested loops:

```
718 read method Bool invariant() {
719   for(Widget w1 : this.box.c) {
720     if(!this.inside(w1)) { return false; }
721     for(Widget w2 : this.box.c) {
722       if(w1!=w2 && SafeMovable.overlap(w1, w2)){return false;}}}
723   return true;}
724
```

Here `SafeMovable.overlap` is a static method that simply checks that the bounds of the widgets don't overlap. The call to `this.inside(w1)` similarly checks that the widget is not outside the bounds of `this`; this instance method call is allowed as `inside` only uses `this` to access its fields.
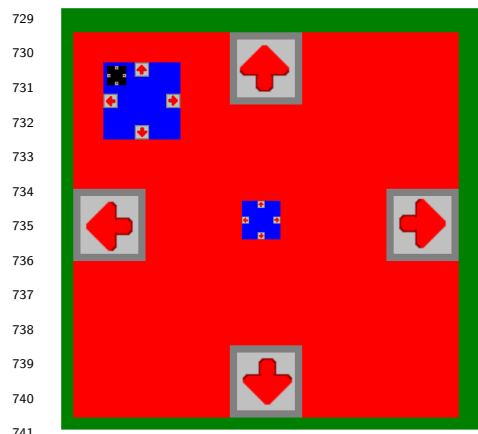
Our Experiment

As shown in the figure to the left, counting both `SafeMovable`s and `Button`s, our main method creates 21 widgets: a top level (green) `SafeMovable` without buttons, containing 4 (red, blue, and black) `SafeMovable`s with 4 (gray) buttons each. When a button is pressed it moves the containing `SafeMovable` a small amount in the corresponding direction. This set up is not overly complicated, the maximum nesting level of `Widget`s is 5. Our main method automatically presses each of the 16 buttons once. In L42, using the approach of this paper, this resulted in 77 calls to `SafeMovable`'s invariant.

Comparison With Visible State Semantics

As an experiment, we set our implementation to generate invariant checks following the visible state semantics approaches of D and Eiffel [3, 24], where the invariant of the receiver is instead checked at the start and end of *every* public (in D) and qualified[16] (in Eiffel) method calls. In our `SafeMovable` class, all methods are public, and all calls are qualified, thus this difference is irrelevant. Neither protocol performs invariant checks on field accesses or updates, however due to the 'uniform access principle', Eiffel allows fields to directly implement methods, allowing the `width` and `height` *fields* to directly implement `Widget`s `width` and `height` *methods*. On the other hand in D, one would have to write getter *methods*, which would invoke invariant checks. When we ran our test case following the D approach, the `invariant` method was called $52,734,053$ times, whereas the Eiffel approach 'only' called it $14,816,207$ times;in comparison our invariant protocol only performed 77 calls. The number of checks is exponential in the depth of the GUI: the invariant of a `SafeMovable` will call the `width`, `height`, `left`, and `top` methods of its children, which may themselves be `SafeMovable`s, and hence such calls may invoke further invariant checks. Note that `width` and `height` are simply getters for fields, whereas the other two are non trivial *methods*.

Spec# Comparison

We also encoded our example in Spec#[17], which like L42, statically verifies aliasing/ownership properties, as well as the admissibility of invariants. The backend of the L42 GUI library is written in Java, we did not port it to Spec#, rather we just simulate the backend, and don't actually display a GUI in Spec#.

We ran our code through the Spec# verifier (powered by Boogie [4]), which only gave us 2 warnings[18]: that the invariant of `SafeMovable` was not known to hold at the end of its constructor and `dispatch` method. Like our system however, Spec# checks the invariant at those two points at runtime. Thus the code is equivalently verified in both Spec# and L42;

---

[16] That is, the receiver is not `this`.

[17] We compiled Spec# using the latest available source (from 19/9/2014). The verifier available online at `rise4fun.com/SpecSharp` behalves differently.

[18] We used `assume` statements, equivalent to Java's `assert`, to dynamically check array bounds. This aligns the code with L42, which also performs such checks at runtime.

in particular it performed exactly the same number (77) of runtime invariant checks.[19]

We found it quite difficult to encode the GUI in Spec#, due to its unintuitive and rigid ownership discipline. In particular we needed to use many more annotations, which were larger and had greater variety. In the following table we summarise the annotation burden, for the *program* that defines and displays the **SafeMovable**s and our GUI; as well as the *library* which defines **Button**s, **Widget**, and event handling.[20]:

	Spec# program	Spec# library	L42 program	L42 library
Total number of annotations	40	19	19	18
Tokens (except `.,;(){}[]` and whitespace)	106	34	18	18
Characters (with minimal whitespace)	619	207	74	60

To encode the GUI example in L42, the only annotations we needed were the 3 type modifiers: **mut**, **read**, and **capsule**. Our Spec# code requires things such as, purity, immutability, ownership, method pre/post conditions and method modification annotations. In addition, it requires the use of 4 different ownership functions including explicit ownership assignments. In total we used 18 different kinds of annotations in Spec#. Together these annotations can get quite long, such as the following precondition on **SafeMovable**'s constructor:

```
requires Owner.Same(Owner.ElementProxy(children), children);
```

The Spec# code also required us to deviate from the style of code we showed in our simplified version: we could not write a usable `children` method in **Widget** that returns a list of children, instead we had to write `children_count()` and `children(int i)` methods; we also needed to create a trivial class with a `[Pure]` constructor (since **Object**'s one is not marked as such). In contrast, the only strange thing we had to in L42 was creating **Box**es by using an additional variable in a nested scope. This is needed to delineate scopes for promotions. Based on these results, we believe our system is significantly simpler and easier to use.

The Box Pattern

Our design, using an inner **Box** object, is a common pattern in static verification: where one encapsulates all relevant mutating state into an encapsulated sub object which is not exposed to users.

Both our L42 and Spec# code required us to use the box pattern for our **SafeMovable**, due to the circular object graph caused by the **Action**s of **Button**s needing to change their enclosing **SafeMovable**'s position.

The Transform Pattern

Suppose we want to scale a **Widget**, we could add **mut** setters for `width`, `height`, `left`, and `top` in the **Widget** interface. However, if we also wish to scale its children we have a problem, since **Widget.children** returns a **read** **Widgets**, which does not allow mutation. We could of course add a **mut** method `zoom` to the **Widget** interface, however this does not scale if more operations are desired. If instead **Widget.children** returned a **mut** **Widgets**, it would be difficult for **Widget** implementations, such as **SafeMovable**, to keep control of their ROGs.

---

[19] We also encoded our GUI in Microsoft Code Contracts [27], whose unsound heuristic also calls the invariant 77 times; however Code Contract does not enforce the encapsulation of `children`, thus their approach would not be sound in our context.

[20] We only count constructs Spec# adds over C# as annotations, we also do not count annotations related to array bounds or null checks.

A simple and practical solution would be to define a `transform` method in `Widget`, and a `Transformer` interface like so:[21]

```
interface Transformer<T> { method Void apply(mut T elem); }
interface Widget { ..
  mut method Void top(Int that); // setters for immutable data
  mut method read Void transform(Transformer<Widgets> t);
} // transformers for possibly encapsulated data
class SafeMovable { ..
  mut method Void transform(Transformer<Widgets> t) {
    return t.apply(this.box.c); }} // Well typed capsule mutator
```

The `transform` method offers an expressive power similar to **mut** getters, but prevents `Widgets` from leaking out. With a `Transformer`, a `zoom` function could be simply:

```
static method Void zoom(mut Widget w) {
  w.transform(ws -> { for (wi : ws) { zoom(wi,scale); }});
  w.width(w.width() / 2); ..; w.top(w.top() / 2); }
```

8   Related work

Type Modifiers

We rely on a combination of modifiers that are supported by at least 3 languages/lines of research: L42 [68, 67, 43, 35], Pony [22, 23], and Gordon et. al. [38]; each of these works is accompanied by proofs about the properties of those modifiers. Since such proofs have already been done, in this work we just assume the required properties. Those approaches all support deep/strong interpretation, without back doors.

TM approaches like Javari [75, 15] and Rust [49] are unsuitable since they introduce back doors which are not easily verifiable as being used properly. Many approaches just try to preserve purity (as for example [63]), but here we also need aliasing control. Ownership [20, 79, 25] is another popular form of aliasing control that can be used as a building block for static verification [57, 7]. Capsule/isolated local variables are affine in that they can be used only once, however this linearity is a property of variables, not expressions or fields. Linear/affine types extend this idea further, however they usually do not consider the ROGs of such types, or work in an OO setting [56, 29].

Object Capabilities

In literature, OCs are used to provide a wide range of guarantees, and many variations are present. Object capabilities [55], in conjunction with type modifiers, are able to enforce purity of code in a modular way, without requiring the use of monads. L42 and Gordon use OCs simply to reason about I/O and non determinism. This approach is best exemplified by Joe-E [32], which is a self contained and minimalistic language using OCs over a subset of Java in order to reason about determinism. However, in order for Joe-E to be a subset of Java, they leverage on a simplified model of immutability: immutable classes must be final with only final fields that refer to immutable classes. In Joe-E, every method that only takes instances of immutable classes is pure. Thus their model would not allow the verification of purity for invariant methods of mutable objects. In contrast our model has a more fine

---

[21] A more general transformer could return a generic **read R**.

grained representation of mutability: it is *reference based* instead of *class based*. In our work, every method taking only `read` or `imm` *references* is pure, regardless of their class type.

Class invariant protocols

Class invariants are a fundamental part of the design by contract methodology. Invariant protocols differ wildly and can be unsound or complicated, particular due to re entrancy and aliasing [45, 26, 53].

While invariant protocols all seem to check and assume the invariant of an object after its construction, they handle invariants differently across object lifetimes; popular sound approaches include:

- The invariants of objects in a *steady* state are known to hold: that is when execution is not inside any of the objects public methods [36]. Invariants need to be constantly maintained between calls to public methods [77].

- The invariant of the receiver before a public method call and at the end of every public method body needs to be ensured. The invariant of the receiver at the beginning of a public method body and after a public method call can be assumed [17, 26]. Some approaches ensure the invariant of the receiver of the *calling* method, rather than the *called* method [58]. JML [34] relaxes these requirements for helper methods, whose semantic is the same as if they were inlined.

- The same as above, but only for the bodies of 'selectively exported' (i.e. non instance private) methods, and only for 'qualified' (i.e. not `this`) calls [53].

- The invariant of an object is assumed only when a contract requires the object be 'packed'. It is checked after an explicit 'pack' operation, and objects can later be 'unpacked' [5].

- Or, as in this work, the invariant of any object which could be *involved* in execution is assumed to hold. It is checked after every modification of the object or its encapsulated ROG.

These different protocols can be deceivingly similar, and some approaches like JML suggest verifying a simpler approach (that method calls preserve the invariant of the *receiver*) but assume a stronger one (the invariant of *every* object, except `this`, holds).

Runtime Verification Tools

Many languages and tools support some form of runtime invariant checking (e.g. Eiffel [52], D [3], and JML [17]). By looking to a survey by Voigt et al. [76] and the extensive MOP project [50], it seems that most runtime verification tools (RV) empower users to implement the kind of monitoring they see fit for their specific problem at hand. This means that users are responsible for deciding, designing, and encoding both the logical properties and the instrumentation criteria [50]. In the context of class invariants, this means the user defines the invariant protocol and the soundness of such protocol is not checked by the tool.

In practice, this means that the logic, instrumentation, and implementation end up connected: a specific instrumentation strategy is only good to test certain logic properties in certain applications. No guarantee is given that the implemented instrumentation strategy is able to support the required logic in the monitored application. Some of these tools are designed to support class invariants: for example InvTS [37] lets you write Python conditions that are verified on a set of Python objects, but the programmer needs to be able to predict which objects are in need of being checked and to use a simpler domain specific language to target them. Hence if a programmer makes a mistake while using this domain specific language, invariant checking will not be triggered. Some tools are intentionally unsound and just perform invariant checking following some heuristic that is expected to catch most failures: such as jmlrac [17] and Microsoft Code Contracts [28].

Many works attempt to move out of the 'RV tool' philosophy to ensure RV monitors work as expected, as for example the study of contracts as refinements of types [31]. However, such work is only interested in pre and post conditions, not class invariants.

Our invariant protocol is much stronger then visible state semantics, and keeps the invariant under tight control. Gopinathan et. al.'s. [36] approach keeps a similar level of control: relying on powerful aspect oriented support, they detect any field update in the whole ROG of any object, and check all the invariants that such update may have violated. We agree with their criticism of visible state semantics, where methods still have to assume that any object may be broken; in such case calling any public method would trigger an error, but while the object is just passed around (and for example stored in collections), the broken state will not be detected; Gopinathan et. al. says "*there are many instances where o's invariant is violated by the programmer inadvertently changing the state of p when o is in a steady state. Typically, o and p are objects exposed by the API, and the programmer (who is the user of the API), unaware of the dependency between o and p, calls a method of p in such a way that o's invariant is violated. The fact that the violation occurred is detected much later, when a method of o is called again, and it is difficult to determine exactly where such violations occur.*"

However, their approach addresses neither exceptions nor non determinism caused by I/O, so their work is unsound if those aspects are taken into consideration.

Their approach is very computationally intensive, but we think it is powerful enough that it could even be used to roll back the very field update that caused the invariant to fail, making the object valid again. We considered a roll back approach for our work, however rolling back a single field update is likely to be completely unexpected, rather we should roll back more meaningful operations, similarly to what happens with transactional memory, and so is likely to be very hard to support efficiently. Using TMs to enforce strong exception safety is a much simpler alternative, providing the same level of safety, albeit being more restrictive (namely that if the operation did succeed it is still effectively rolled back).

Chaperones and impersonators [71] lifts the techniques of gradual typing [73, 74, 78] to work on general purpose predicates, where values can be wrapped to ensure an invariant holds. This technique is very powerful and can be used to enforce pre and post conditions by wrapping function arguments and return values. This technique however does not monitor the effects of aliasing, as such they may notice if a contract has broken, but not when or why. In addition, due to the difficulty of performing static analysis in weakly typed languages, they need to inject runtime checking code around every user facing operation. Aspect oriented systems like Jose [30], similarly wrap invariant checks around method bodies.

Security and Scalability

Our approach allows verifying an object's invariant independently of the actual invariants of other objects. This is in contrast with the main strategy of static verification: to verify a method, the system assumes the contracts of other methods, and the content of those contracts is the starting point for their proof. Thus, static verification proceeds like a mathematical proof: a program is valid if it is all correct, but a single error invalidates all claims. This makes it hard to perform verification on large programs, or when independently maintained third party libraries are involved. This is less problematic with a type system, since its properties are more coarse grained, simpler and easier to check. Static verification has more flexible and fine grained annotations and often relies on a fragile theorem prover as a backend.

To soundly verify code embedded in an untrusted environment, as in gradual typing [74, 78], it is possible to consider a verified core and a runtime verified boundary. You can see our

approach as an extremely modularized version of such system: every class is its own verified core, and the rest of the code could have Byzantine behaviour. Our formal proofs show that every class that compiles/type checks is soundly handled by our protocol, independently of the code that uses such class or any other surrounding code.

Our approach works both in a library setting and with the open world assumption. Consider for example the work of Parkinson [62]: in his short paper he verified a property of the `Subject/Observer` pattern. However, the proof relies on (any override of) the `Subject.register(Observer)` method respecting its contract. Such assumption is unrealistic in a real world system with dynamic class loading, and could trivially be broken by a user defined `EvilSubject`.

Static Verification

Spec# [8] is a language built on top of C#, it adds various annotations such as method contracts and class invariants. It primarily follows the Boogie methodology [59] where (implicit) annotations are used to specify and modify the owner of objects and whether their invariants are required to hold. Invariants can be *ownership* based [5], where an invariant only depends on objects it owns; or *visibility* based [6, 46], where an invariant may depend on objects it doesn't own, provided that the class of such objects know about this dependence. Unlike our approach, Spec# does not restrict the aliases that may exist for an object, rather it restricts object mutation: an object cannot be modified if the invariant of its owner is required to hold. This is more flexible than our approach as it also allows only part of an object's ROG to be owned/encapsulated. However as we showed in Section 7, it can become much more difficult to work with and requires significant annotation since merely having an alias to an object is insufficient to modify it or call methods on it. Spec# also works with existing .NET libraries by annotating them with contracts, however such annotations are not verified. Spec#, like us, does perform runtime checks for invariants and throws unchecked exceptions on failure. However Spec# does not allow soundly recovering from an invariant failure, since catching unchecked exceptions in Spec# is intentionally unsound. [48]

Another system is AutoProof [65], a static verifier for Eiffel that also follows the Boogie methodology, but extends it with *semantic collaboration* where objects keep track of their invariants' dependencies using ghost state. Dafny [44] is a new language where all code is statically verified, it supports invariants by injecting pre and post conditions following visible state semantics; however it requires objects to be newly allocated (or cloned) before another object's invariant may depend on it. Dafny is also generally highly restrictive with its rules for mutation, and object construction, it also does not provide any means of performing non deterministic I/O.

Specification languages

Using a specification language based on the mathematical metalanguage and different from the program language's semantics may seem attractive, since it can express uncomputable concepts, has no mutation or non determinism, and is often easier to formally reason about.

However, a study [18] discovered that developers expect specification languages to follow the semantics of the underling language, including short circuit semantics and arithmetic exceptions; thus for example `1/0 || 2>1` should not hold, while `2>1 || 1/0` should, thanks to short circuiting. This study was influential enough to convince JML to change its interpretation of logical expressions accordingly [19]. Dafny [44] uses a hybrid approach: it has mostly the same language for both specification and execution. Specification ('ghost') contexts can use uncomputable constructs such as universal quantification over infinite sets. Whereas runtime contexts allow mutation, object allocation and print statements. The semantics of shared constructs (such as short circuiting logic operators) is the same in both

contexts.

Most runtime verification systems, such as ours, use a metacircular approach: specifications are simply code in the underlying language. Since specifications are checked at runtime, they are unable to verify uncomputable contracts. Ensuring determinism in a non functional language is challenging. Spec# recognizes the need for purity/determinism when method calls are allowed in contracts [9] '*There are three main current approaches: a) forbid the use of functions in specifications, b) allow only provably pure functions, or c) allow programmers free use of functions. The first approach is not scalable, the second overly restrictive and the third unsound.*'.

They recognize that many tools unsoundly use option (c), such as AsmL [10]. Spec# aims to follow (b) but only considers non determinism caused by memory mutation, and allows other non deterministic operations, such as I/O and random number generation. For example, the following method verifies:

```
[Pure] bool uncertain() {return new Random().Next() % 2 == 0;}
```

And so `assert uncertain() == uncertain();` also verifies, but randomly fails with an exception at runtime. As you can see failing to handle non determinism jeopardises reasoning.

A simpler and more restrictive solution to these problems is to prevent 'pure' functions from reading or writing to any non final fields, or calling any impure functions. This is the approach used by [33], one advantage of their approach is that invariants (which must be 'pure') can read from a chain of final fields, even when they are contained in otherwise mutable objects. However their approach completely prevents invariants from mutating newly allocated objects, thus greatly restricting how computations can be performed.

9   Conclusions and Future Work

Our approach follows the principles of *offensive programming* [70], where no attempt to fix or recover an invalid object is performed and failures (unchecked exceptions) are raised close to their cause: at the end of constructors creating invalid objects and immediately after field updates and instance methods that invalidate their receivers.

Our work builds on a specific form of TMs and OCs, whose popularity is growing, and we expect future languages to support some variation of these. Crucially, any language already designed with such TMs and OCs can also support our invariant protocol with minimal added complexity.

We demonstrated the applicability and simplicity of our approach with a GUI example. Our invariant protocol performs several orders of magnitude less checks than visible state semantics, and requires much less annotation than Spec#, (the system with the most comparable goals). In Section 4 we formalised our invariant protocol and in Appendix A we prove it sound. To stay parametric over the various existing type systems which provably enforce the properties we require for our proof (and much more), we do not formalise any specific type system.

One interesting avenue for future work would be to use invariants to encode pre and post conditions, as done by [33]: where pre and post conditions are encoded as the invariants of the parameter and return types (respectively). Without good syntax sugar, such an approach could be quite verbose, however it would ensure that a methods precondition holds during the entire execution of a method, and not just the beginning. In addition this could be more efficient than traditional runtime checking when the same argument is used in the invocations

of methods with the same pre condition, as happens often in practice for recursive methods: where many parameters are simply parsed unmodified in recursive calls.

The language we presented here restricts the forms of `invariant` and capsule mutator methods; such strong restrictions allow for sound and efficient injection of invariant checks. These restrictions do not get in the way of writing invariants over immutable data, but the box pattern is required for verifying complex mutable data structures. We believe this pattern, although verbose, is simple and understandable. While it may be possible for a more complex and fragile type system to reduce the need for the pattern whilst still ensuring our desired semantics, we prioritize simplicity and generality.

In order to obtain safety, simplicity, and efficiency we traded some expressive power: the `invariant` method can only refer to immutable and encapsulated state. This means that while we can easily verify that a doubly linked list of immutable elements is correctly linked up, we can not do the same for a doubly linked lists of mutable elements. Our approach does not prevent correctly implementing such data structures, but the `invariant` method would be unable to access the list's nodes, since they would contain `mut` references to shared objects. In order to verify such data structures we could add a special kind of field which cannot be (transitively) accessed by invariants; such fields could freely refer to any object. We are however unsure if such complexity would be justified.

For an implementation of our work to be sound, catching exceptions like stack overflows or out of memory cannot be allowed in `invariant` methods, since they are not deterministically thrown. Currently L42 never allows catching them, however we could also write a (native) capability method (which can't be used inside an invariant) that enables catching them. Another option worth exploring would be to make such exceptions deterministic, perhaps by giving invariants fixed stack and heap sizes.

Other directions that could be investigated to improve our work include the addition of syntax sugar to ease the burden of the box and the transform patterns; type modifier inference, and support for flexible ownership types.

References

1   David Abrahams. *Exception-Safety in Generic Components*, pages 69–79. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000. `doi:10.1007/3-540-39953-4_6`.

2   Alexander Aiken, Jeffrey S. Foster, John Kodumal, and Tachio Terauchi. Checking and inferring local non-aliasing. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, pages 129–140, 2003. `doi:10.1145/781131.781146`.

3   Andrei Alexandrescu. *The D Programming Language*. Addison-Wesley Professional, 1st edition, 2010.

4   Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, pages 364–387, 2005. `doi:10.1007/11804192\_17`.

5   Michael Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004. `doi:10.5381/jot.2004.3.6.a2`.

6   Michael Barnett and David A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *Mathematics of Program Construction, 7th International Conference, MPC 2004, Stirling, Scotland, UK, July 12-14, 2004, Proceedings*, pages 54–84, 2004. `doi:10.1007/978-3-540-27764-4\_5`.

7   Mike Barnett, Manuel Fähndrich, K Rustan M Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and verification: the spec# experience. *Communications of the ACM*, 54(6):81–91, 2011. `doi:10.1145/1953122.1953145`.

8   Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system: An overview. In *Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, CASSIS'04, pages 49–69, Berlin, Heidelberg, 2005. Springer-Verlag. `doi:10.1007/978-3-540-30569-9_3`.

9   Mike Barnett, David A Naumann, Wolfram Schulte, and Qi Sun. 99.44% pure: Useful abstractions in specifications. In *ECOOP workshop on Formal Techniques for Java-like Programs (FTfJP)*, 2004. `doi:10.1.1.72.3429`.

10   Mike Barnett and Wolfram Schulte. Runtime verification of .net contracts. *Journal of Systems and Software*, 65(3):199–208, 2003. `doi:10.1016/S0164-1212(02)00041-9`.

11   Adrian Birka and Michael D. Ernst. A practical type system and language for reference immutability. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2004)*, pages 35–49, 2004. `doi:10.1145/1035292.1028980`.

12   Joshua Bloch. *Effective Java (2Nd Edition) (The Java Series)*. Prentice Hall PTR, 2 edition, 2008.

13   John Boyland. Alias burying: Unique variables without destructive reads. *Software: Practice and Experience*, 31(6):533–553, 2001. `doi:10.1002/spe.370`.

14   John Boyland. Checking interference with fractional permissions. In *International Static Analysis Symposium*, pages 55–72. Springer, 2003.

15   John Boyland. Why we should not add readonly to Java (yet). *Journal of Object Technology*, 5(5):5–29, 2006. `doi:10.5381/jot.2006.5.5.a1`.

16   John Boyland. Semantics of fractional permissions with nesting. *ACM Transactions on Programming Languages and Systems*, 32(6), 2010. `doi:10.1145/1749608.1749611`.

17   Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of jml tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, Jun 2005. `doi:10.1007/s10009-004-0167-4`.

18   Patrice Chalin. Are the logical foundations of verifying compiler prototypes matching user expectations? *Formal Aspects of Computing*, 19(2):139–158, 2007. `doi:10.1007/s00165-006-0016-1`.

19   Patrice Chalin and Frédéric Rioux. Jml runtime assertion checking: Improved error reporting and efficiency using strong validity. *FM 2008: Formal Methods*, pages 246–261, 2008. `doi:10.1007/978-3-540-68237-0_18`.

20   Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. Ownership types: A survey. In Dave Clarke, James Noble, and Tobias Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *Lecture Notes in Computer Science*, pages 15–58. Springer, 2013. `doi:10.1007/978-3-642-36946-9_3`.

21   David Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *ECOOP'03 - Object-Oriented Programming*, volume 2473 of *Lecture Notes in Computer Science*, pages 176–200. Springer, 2003. `doi:10.1007/978-3-540-45070-2_9`.

22   Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, pages 1–12. ACM, 2015. `doi:10.1145/2824815.2824816`.

23   Sylvan Clebsch, Juliana Franco, Sophia Drossopoulou, Albert Mingkun Yang, Tobias Wrigstad, and Jan Vitek. Orca: Gc and type system co-design for actor languages. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):72, 2017. `doi:10.1145/3133896`.

24 D Language Foundation. D Programming Language Specification, 2018. URL: `https://dlang.org/dlangspec.pdf`.

25 Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic universe types. In *ECOOP'07 - Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, pages 28–53. Springer, 2007. `doi:10.1007/978-3-540-73589-2_3`.

26 Sophia Drossopoulou, Adrian Francalanza, Peter Müller, and Alexander J Summers. A unified framework for verification techniques for object invariants. In *European Conference on Object-Oriented Programming*, pages 412–437. Springer, 2008. `doi:10.1007/978-3-540-70592-5_18`.

27 Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. Embedded contract languages. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010*, pages 2103–2110, 2010. `doi:10.1145/1774088.1774531`.

28 Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. Embedded contract languages. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 2103–2110. ACM, 2010. `doi:10.1145/1774088.1774531`.

29 Manuel Fahndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *ACM SIGPLAN Notices*, volume 37, pages 13–24. ACM, 2002. `doi:10.1145/543552.512532`.

30 Yishai A Feldman, Ohad Barzilay, and Shmuel Tyszberowicz. Jose: Aspects for design by contract. In *Software Engineering and Formal Methods, 2006. SEFM 2006. Fourth IEEE International Conference on*, pages 80–89. IEEE, 2006. `doi:10.1109/SEFM.2006.26`.

31 Robert Bruce Findler and Matthias Felleisen. Contract soundness for object-oriented languages. In *ACM SIGPLAN Notices*, volume 36, pages 1–15. ACM, 2001. `doi:10.1145/504311.504283`.

32 Matthew Finifter, Adrian Mettler, Naveen Sastry, and David Wagner. Verifiable functional purity in java. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 161–174. ACM, 2008. `doi:10.1145/1455770.1455793`.

33 Cormac Flanagan. Hybrid types, invariants, and refinements for imperative objects. In *In International Workshop on Foundations and Developments of Object-Oriented Languages*, 2006.

34 Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter M¨uller, Joseph Kiniry, Patrice Chalin, Daniel M. Zimmerman, Werner Dietl. JML Reference Manual, 2013. URL: `http://www.eecs.ucf.edu/~leavens/JML//refman/jmlrefman.pdf`.

35 Paola Giannini, Marco Servetto, and Elena Zucca. Types for immutability and aliasing control. In *ICTCS'16 - Italian Conf. on Theoretical Computer Science*, volume 1720 of *CEUR Workshop Proceedings*, pages 62–74. CEUR-WS.org, 2016. URL: `http://ceur-ws.org/Vol-1720/full5.pdf`.

36 Madhu Gopinathan and Sriram K. Rajamani. Runtime verification. chapter Runtime Monitoring of Object Invariants with Guarantee, pages 158–172. Springer-Verlag, Berlin, Heidelberg, 2008. `doi:10.1007/978-3-540-89247-2_10`.

37 Michael Gorbovitski, Tom Rothamel, Yanhong A. Liu, and Scott D. Stoller. Efficient runtime invariant checking: A framework and case study. In *Proceedings of the 6th International Workshop on Dynamic Analysis (WODA 2008)*. ACM Press, 2008. `doi:10.1145/1401827.1401837`.

38 Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and reference immutability for safe parallelism. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2012)*, pages 21–40. ACM Press, 2012. `doi:10.1145/2384616.2384619`.

1185 **39** Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In Theo
1186   D'Hondt, editor, *ECOOP'10 - Object-Oriented Programming*, volume 6183 of *Lecture Notes*
1187   *in Computer Science*, pages 354–378. Springer, 2010. `doi:10.1007/978-3-642-14107-2_`
1188   `17`.

1189 **40** John Hogg. Islands: Aliasing protection in object-oriented languages. In *ACM Symp. on*
1190   *Object-Oriented Programming: Systems, Languages and Applications 1991*, pages 271–285.
1191   ACM Press, 1991.

1192 **41** Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal
1193   core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*,
1194   23(3):396–450, 2001.

1195 **42** Paul Ashley Karger. *Improving security and performance for capability systems*. PhD thesis,
1196   Citeseer, 1988.

1197 **43** Giovanni Lagorio and Marco Servetto. Strong exception-safety for checked and unchecked
1198   exceptions. *Journal of Object Technology*, 10:1:1–20, 2011. `doi:10.5381/jot.2011.10.1.`
1199   `a1`.

1200 **44** K. Rustan M. Leino. Developing verified programs with dafny. In *Proceedings of the 2012*
1201   *ACM Conference on High Integrity Language Technology, HILT '12, December 2-6, 2012,*
1202   *Boston, Massachusetts, USA*, pages 9–10, 2012. `doi:10.1145/2402676.2402682`.

1203 **45** K Rustan M Leino and Peter Müller. Object invariants in dynamic contexts. In *European*
1204   *Conference on Object-Oriented Programming*, pages 491–515. Springer, 2004. `doi:10.1007/`
1205   `978-3-540-24851-4_22`.

1206 **46** K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In *ECOOP*
1207   *2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-*
1208   *18, 2004, Proceedings*, pages 491–516, 2004. `doi:10.1007/978-3-540-24851-4\_22`.

1209 **47** K. Rustan M. Leino, Peter Müller, and Angela Wallenburg. Flexible immutability with
1210   frozen objects. In *Verified Software: Theories, Tools, Experiments, Second International*
1211   *Conference, VSTTE 2008, Toronto, Canada, October 6-9, 2008. Proceedings*, pages 192–
1212   208, 2008. `doi:10.1007/978-3-540-87873-5\_17`.

1213 **48** K. Rustan M. Leino and Wolfram Schulte. Exception safety for c#. *Proceedings of*
1214   *the Second International Conference on Software Engineering and Formal Methods, 2004.*
1215   *SEFM 2004.*, pages 218–227, 2004.

1216 **49** Nicholas D Matsakis and Felix S Klock II. The rust language. In *ACM SIGAda Ada Letters*,
1217   volume 34, pages 103–104. ACM, 2014. `doi:10.1145/2663171.2663188`.

1218 **50** Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Roşu. An
1219   overview of the mop runtime verification framework. *International Journal on Software*
1220   *Tools for Technology Transfer*, 14(3):249–289, 2012. `doi:10.1007/s10009-011-0198-6`.

1221 **51** Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle
1222   River, NJ, USA, 1st edition, 1988.

1223 **52** Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA,
1224   1992.

1225 **53** Bertrand Meyer. Class invariants: Concepts, problems, solutions. *arXiv preprint*
1226   *arXiv:1608.07637*, 2016.

1227 **54** Mark S Miller, Ka-Ping Yee, Jonathan Shapiro, et al. Capability myths demolished. Tech-
1228   nical report, Technical Report SRL2003-02, Johns Hopkins University Systems Research
1229   Laboratory, 2003. http://www. erights. org/elib/capability/duals, 2003.

1230 **55** Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control*
1231   *and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland,
1232   USA, May 2006.

56 Greg Morrisett, Amal Ahmed, and Matthew Fluet. L 3: a linear language with locations. In *International Conference on Typed Lambda Calculi and Applications*, pages 293–307. Springer, 2005. `doi:10.1007/11417170_2`.

57 Peter Müller. *Modular specification and verification of object-oriented programs*. Springer-Verlag, 2002. `doi:10.1007/3-540-45651-1`.

58 Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Sci. Comput. Program.*, 62(3):253–286, 2006. URL: `https://doi.org/10.1016/j.scico.2006.03.001`, `doi:10.1016/j.scico.2006.03.001`.

59 David A. Naumann and Michael Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state. *Theor. Comput. Sci.*, 365(1-2):143–168, 2006. URL: `https://doi.org/10.1016/j.tcs.2006.07.035`, `doi:10.1016/j.tcs.2006.07.035`.

60 James Noble, Sophia Drossopoulou, Mark S Miller, Toby Murray, and Alex Potanin. Abstract data types in object-capability systems. 2016.

61 Johan Östlund, Tobias Wrigstad, Dave Clarke, and Beatrice Åkerblom. Ownership, uniqueness, and immutability. In Richard F. Paige and Bertrand Meyer, editors, *International Conference on Objects, Components, Models and Patterns*, volume 11 of *Lecture Notes in Computer Science*, pages 178–197. Springer, 2008. `doi:10.1007/978-3-540-69824-1_11`.

62 Matthew Parkinson. Class invariants: The end of the road? *Aliasing, Confinement and Ownership in Object-oriented Programming (IWACO)*, page 9, 2007.

63 David Pearce. Jpure: a modular purity system for java. In *Compiler construction*, pages 104–123. Springer, 2011. `doi:10.1007/978-3-642-19861-8_7`.

64 Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

65 Nadia Polikarpova, Julian Tschannen, Carlo A. Furia, and Bertrand Meyer. Flexible invariants through semantic collaboration. In *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, pages 514–530, 2014. `doi:10.1007/978-3-319-06410-9\_35`.

66 Alex Potanin, Johan Östlund, Yoav Zibin, and Michael D. Ernst. *Immutability*, pages 233–269. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. `doi:10.1007/978-3-642-36946-9_9`.

67 Marco Servetto, David J. Pearce, Lindsay Groves, and Alex Potanin. Balloon types for safe parallelisation over arbitrary object graphs. In *WODET 2014 - Workshop on Determinism and Correctness in Parallel Programming*, 2013. `doi:doi=10.1.1.353.2449`.

68 Marco Servetto and Elena Zucca. Aliasing control in an imperative pure calculus. In Xinyu Feng and Sungwoo Park, editors, *Programming Languages and Systems - 13th Asian Symposium (APLAS)*, volume 9458 of *Lecture Notes in Computer Science*, pages 208–228. Springer, 2015. `doi:10.1007/978-3-319-26529-2_12`.

69 Frederick Smith, David Walker, and J. Gregory Morrisett. Alias types. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*, ESOP '00, pages 366–381, London, UK, UK, 2000. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=645394.651903`.

70 R. Stephens. *Beginning Software Engineering*. Wiley, 2015.

71 T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and impersonators: run-time support for reasonable interposition. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 943–962, 2012. `doi:10.1145/2384616.2384685`.

72 Alexander J. Summers, Sophia Drossopoulou, and Peter Müller. The need for flexible object invariants. In *International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming*, IWACO '09, pages 6:1–6:9, New York, NY, USA, 2009. ACM. `doi:10.1145/1562154.1562160`.

73   Asumu Takikawa, Daniel Feltey, Earl Dean, Matthew Flatt, Robert Bruce Findler, Sam Tobin-Hochstadt, and Matthias Felleisen. Towards practical gradual typing. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015. `doi:10.4230/LIPIcs.ECOOP.2015.4`.

74   Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Gradual typing for first-class classes. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 793–810, 2012. `doi:10.1145/2384616.2384674`.

75   Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to Java. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2005)*, pages 211–230. ACM Press, 2005. `doi:10.1145/1094811.1094828`.

76   Janina Voigt, Warwick Irwin, and Neville Churcher. *Comparing and Evaluating Existing Software Contract Tools*, pages 49–63. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. `doi:10.1007/978-3-642-32341-6_4`.

77   Wikipedia contributors. Class invariant, 2018. URL: `https://en.wikipedia.org/wiki/Class_invariant`.

78   Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. Integrating typed and untyped code in a scripting language. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 377–388, 2010. `doi:10.1145/1706299.1706343`.

79   Yoav Zibin, Alex Potanin, Paley Li, Mahmood Ali, and Michael D. Ernst. Ownership and immutability in generic Java. In *ACM SIGPLAN Conference on Object-Oriented Programming,Systems, Languages and Applications (OOPSLA 2010)*, pages 598–617, 2010. `doi:10.1145/1869459.1869509`.

A   Proof and Axioms

Axiomatic Type Properties

As previously discussed, instead of providing a concrete set of type rules, we provide a set of properties that the type system needs to respect. To express these properties, we first need some auxiliary definitions.

The encapsulated ROG of l_0 is composed of all the objects in the ROG of its immutable and capsule fields:

$l \in erog(\sigma, l_0)$ iff $\exists f, \Sigma^\sigma(l_0).f \in \{\texttt{imm \_}, \texttt{capsule \_}\}$ and $l \in rog(\sigma, \sigma(l_0).f)$

An object is *mutatable* in a σ and e if there is an occurrence of l in e, that when seen as $\texttt{imm}$ makes the expression ill typed:

$mutatable(l, \sigma, e)$ iff for some $T = \texttt{imm} \, \Sigma^\sigma(l)$ and $\mathcal{E}[l] = e$,

$\Sigma^\sigma; x : T \vdash \mathcal{E}[x] : T'$ does not hold for any T'.

Here we assume the usual Progress and Subject Reduction Base. Note that Subject Reduction Base only ensures properties about type checking, not invariant checking.

▶ **Assumption 1** (Progress). if $\Sigma^{\sigma_0}; \emptyset \vdash e_0 : T_0$, and e_0 is not form l or *error*, then $\sigma_0|e_0 \rightarrow \sigma_1|e_1$.

▶ **Assumption 2** (Subject Reduction Base). if $\Sigma^{\sigma_0}; \emptyset \vdash e_0 : T_0$, $\sigma_0|e_0 \rightarrow \sigma_1|e_1$, then $\Sigma^{\sigma_1}; \emptyset \vdash e_1 : T_1$.

If the result of a field access is $\texttt{mut}$, the receiver is also $\texttt{mut}$; field updates are only allowed on $\texttt{mut}$ receivers:

▶ **Assumption 3** (Mut Field).
    (1) if $\Sigma; \Gamma \vdash e.f : \texttt{mut \_}$ then $\Sigma; \Gamma \vdash e : \texttt{mut \_}$ and
    (2) if $\Sigma; \Gamma \vdash e_0.f = e_1 : T$ then $\Sigma; \Gamma \vdash e_0 : \texttt{mut \_}$.

An object is not part of the ROG of its immutable or capsule fields[22]:

▶ **Assumption 4** (Head Not Circular). if $\Sigma^\sigma; \Gamma \vdash l : T$, then $l \notin \text{erog}(\sigma, l)$.

In a well typed σ and e, if mutatable l_2 is reachable through the *erog* of l_1, and l_1 is reachable through the *erog* of l_0, then all the paths connecting l_0 and l_2 pass trough l_1; thus if we were to remove l_1 from the object graph, l_0 would no longer reach l_2:

▶ **Assumption 5** (Capsule Tree). If $\Sigma^\sigma; \Gamma \vdash e : T$, $l_2 \in \text{erog}(\sigma, l_1)$, $l_1 \in \text{erog}(\sigma, l_0)$, and $mutatable(l_2, \sigma, e)$ then $l_2 \notin \text{erog}(\sigma \setminus l_1, l_0)$.

Capsule Tree and Head Not Circular together imply that capsule fields section the object graph into a tree of nested 'balloons', where nodes are mutable encapsulated objects and edges are given by reachability between those objects in the original memory: if l_2 is in the encapsulated ROG of l_1, and l_2 is mutatable and reachable through l_1, then l_2 must be reachable by a $\texttt{capsule}$ field. Thanks to Head Not Circular and $l_1 \in \text{erog}(\sigma, l_0)$ we can derive that $l_0 \notin \text{erog}(\sigma, l_1)$.

The execution of an expression with no $\texttt{mut}$ free variables is deterministic and does not mutate pre existing memory (and thus does not not perform I/O by mutating the pre existing c):

▶ **Assumption 6** (Determinism). if $\emptyset; \Gamma \vdash e : T$, $\forall x \, (\Gamma(x) \neq \texttt{mut \_})$, and $\sigma|e' \rightarrow^+ \sigma'|e''$ then $\sigma|e' \Rightarrow^+ \sigma, \_|e''$, where $e' = e[x_1 = l_1, ..., x_n = l_n]$ and $\Sigma^\sigma; \emptyset \vdash e' : T$

---

[22] This is not strictly true in L42, as 42 allows circular objects with $\texttt{fwd imm}$ fields, however such objects cannot have an invariant.

1351   For each **try**–**catch**, execution preserves the memory needed to continue the execution
1352   in case of an error (the memory visible outside of the **try**).

1353   ▶ **Assumption 7** (Strong Exception Safety). if $\Sigma^{\sigma,\sigma'}; \emptyset \vdash \mathcal{E}[\text{try}^\sigma \{e_0\} \text{ catch } \{e_1\}] : T$ and
1354 $\sigma, \sigma' | \mathcal{E}[\text{try}^\sigma \{e_0\} \text{ catch } \{e_1\}] \rightarrow \sigma'' | \mathcal{E}[\text{try}^\sigma \{e'\} \text{ catch } \{e_1\}]$ then $\sigma'' = \sigma, \_$ and $\Sigma^\sigma; \emptyset \vdash \mathcal{E}[e_1] : T$

1355   Note that our last well formedness rule requires UPDATE and MCALL to introduce mon-
1356   itor expressions only over locations that are not preserved by **try** blocks. This can be
1357   achieved, since monitors are introduced around mutation operations (and **new** expression),
1358   and Strong Exception Safety ensures no mutation happens on preserved memory.

1359   **Proof of Soundness**
1360   It is hard to prove Soundness directly, so we first define a stronger property, called Stronger Soundness,
1361   and show that it is preserved during reductions by means of conventional Progress and
1362   Subject Reduction (Progress is one of our assumptions, while Subject Reduction relies heavily upon
1363   Subject Reduction Base). That is:
1364   ▪ Progress ∧ Subject Reduction ⇒ Stronger Soundness, and
1365   ▪ Stronger Soundness ⇒ Soundness.

1366   **Stronger Soundness ⇒ Soundness**
1367   Stronger Soundness depends on *wellEncapsulated*, *monitored* and *OK*:
1368 $wellEncapsulated(\sigma, e, l_0)$ iff $\forall l \in erog(\sigma, l_0),$ not $mutatable(l, \sigma, e).$
1369   The main idea is that an object is well encapsulated if its encapsulated state cannot be
1370   modified by e.
1371   An object is *monitored* if execution is currently inside of a monitor for that object, and
1372   the monitored expression e_1 does not contain l as a *proper* subexpression:
1373 $monitored(e, l)$ iff $e = \mathcal{E}_v[\text{M}(l ; e_1 ; e_2)]$ and either $e_1 = l$, or l is not inside e_1.
1374   A monitored object is associated with an expression that can not observe it, but may reference
1375   its internal representation directly. In this way, we can safely modify its representation before
1376   checking its invariant.
1377   The idea is that at the start the object will be valid and e_1 will reference l; but during
1378   reduction, l will be used to modify the object; only after that moment, the object may
1379   become invalid.
1380   **Define** $OK(\sigma, e)$:
1381 $\forall l \in dom(\sigma)$ either
1382   1. $garbage(l, \sigma, e),$
1383   2. $valid(\sigma, l)$ and $wellEncapsulated(\sigma, e, l),$ or
1384   3. $monitored(e, l).$
1385   Finally, the system is in an *OK* state if all objects in memory, are either not (transitively)
1386   reachable from the expression (thus can be garbage collected), valid and encapsulated, or
1387   currently monitored.

1388   ▶ **Theorem 2** (Stronger Soundness). if $c : \text{Cap}; \emptyset \vdash e_0 : T_0$ and $c \mapsto \text{Cap}\{\_\} | e_0 \rightarrow^+ \sigma | e$, then
1389 $OK(\sigma, e).$

1390   Starting from only the capability object, any well typed expression e_0 can be reduced in an
1391   arbitrary number of steps, and *OK* will always hold.

1392   ▶ **Theorem 3.** Stronger Soundness ⇒ Soundness

1393   *Proof.* By Stronger Soundness, each l in the current redex must be *OK*:
1394   **1.** If l is *garbage*, it cannot be in the current redex, a contradiction.

2. If $valid(\sigma, l)$, then l is valid, so thanks to Determinism no invalid object could be observed.

3. Otherwise, if $monitored(e, l)$ then either:

- we are executing inside of e_1, thus the current redex is inside of a sub expression of the monitor that does not contain l, a contradiction.
- or we are executing inside e_2: by our reduction rules, all monitor expressions start with $e_2 = l.\texttt{invariant()}$, thus the first execution step of e_2 is trusted. Further execution steps are also trusted, since by well formedness the body of invariant methods only use **this** (now replaced with l) to read fields.

In any of the possible cases above, Soundness holds for l, and so it holds for all redexes.

Subject Reduction

Define $fieldGuarded(\sigma, e)$:

$\quad \forall \mathcal{E}$ such that $e = \mathcal{E}[l.f]$ and $\Sigma^\sigma(l).f = \texttt{capsule}\_$, and f *inside* $\Sigma^\sigma(l).\texttt{invariant()}$

$\qquad$ either $\forall T, \forall C, \Sigma^\sigma; x : \texttt{mut}\, C \nvdash \mathcal{E}[x] : T$, or

$\qquad \mathcal{E} = \mathcal{E}'[\texttt{M}(l; \mathcal{E}''; e)]$ and l is contained exactly once in $\mathcal{E}''$.

$\quad$ That is, all **mut** capsule field accesses are individually guarded by monitors. Note how we use C in $x : \texttt{mut}\, C$ to guess the type of the accessed field, and that we use the full context $\mathcal{E}$, instead of the evaluation context $\mathcal{E}_v$, to refer to field accesses everywhere in the expression e.

▶ **Theorem 4** (Subject Reduction). if $\Sigma^{\sigma_0}; \emptyset \vdash e_0 : T_0$, $\sigma_0|e_0 \rightarrow \sigma_1|e_1$, $OK(\sigma_0, e_0)$ and $fieldGuarded(\sigma_0, e_0)$ then $\Sigma^{\sigma_1}; \emptyset \vdash e_1 : T_1$, $OK(\sigma_1, e_1)$ and $fieldGuarded(\sigma_1, e_1)$

▶ **Theorem 5.** Progress + Subject Reduction $\Rightarrow$ Stronger Soundness

Proof. This proof proceeds by induction in the usual manner.

$\quad$ *Base case*: At the start of execution, memory only contains c: since c is defined to always be *valid*, and has only **mut** fields, it is trivially *wellEncapsulated*, thus $OK(c \mapsto \texttt{Cap}, e)$.

$\quad$ *Induction*: By Progress, we always have another evaluation step to take, by Subject Reduction such a step will preserve OK, and so by induction, OK holds after any number of steps.

$\quad$ Note how for the proof garbage collectability is important: when the `invariant()` method evaluates to **false**, execution can continue only if the offending object is classified as *garbage*.

Exposer Instrumentation

We first introduce a lemma derived from our well formedness criteria and the type system:

▶ **Lemma 1** (Exposer Instrumentation). If $\sigma_0|e_0 \rightarrow \sigma_1|e_1$ and $fieldGuarded(\sigma_0, e_0)$ then $fieldGuarded(\sigma_1, e_1)$.

Proof. The only rule that can introduce a new field access is MCALL. In that case, Exposer Instrumentation holds by well formedness (all field accesses in methods are of the form **this**.$\texttt{f}$), since MCALL inserts a monitor while invoking capsule mutator methods, and not field accesses themselves. If however the method is not a **mut** method but still accesses a capsule field, by Mut Field such a field access expression cannot be typed as **mut** and so no monitor is needed.

$\quad$ Note that MONITOR EXIT is fine because monitors are removed only when e_1 is a value.

Proof of Subject Reduction

Any reduction step can be obtained by exactly one application of the CTXV rule and one other rule. Thus the proof can simply proceed by cases on the other applied rule.

$\quad$ By Subject Reduction Base and Exposer Instrumentation, $\Sigma^{\sigma_1}; \emptyset \vdash e_1 : T_1$ and $fieldGuarded(\sigma_1, e_1)$. So we just need to proceed by cases on the reduction rule applied to verify that $OK(\sigma_1, e_1)$ holds:

1. (UPDATE) $\sigma|l.f = v \rightarrow \sigma'|e'$:

- By UPDATE $e' = \texttt{M}(l;l;l)\texttt{.invariant()}$, thus $monitored(e,l)$.
- Every l_1 such that $l \in rog(\sigma, l_1)$ will verify the same case as the former step:
  - If it was *garbage*, clearly it still is.
  - If it was *monitored*, it still is.
  - Otherwise it was *valid* and *wellEncapsulated*:
    * If $l \in erog(\sigma, l_1)$ we have a contradiction since $mutatable(l, \sigma, e)$, (by Mut Field)
    * Otherwise, by our well formedess criteria that `invariant` only accesses `imm` and `capsule` fields, and by Determinism, it is clearly the case that *valid* still holds; By Head Not Circular it cannot be the case that $l \in erog(\sigma', l_1)$, and so l_1 is still *wellEncapsulated*.
- Every other l_0 is not in the reachable object graph of l, thus it being *OK* could not have been effected by this reduction step.

2. (ACCESS) $\sigma|l.f \to \sigma|v$:
  - If l was *valid* and *wellEncapsulated*:
    - If we have now broken *wellEncapsulated*, we must have made something in its *erog mutatable*. As we can only type `capsule` fields as `mut` and not `imm` fields, by Mut Field we must have that f is `capsule` and $l.f$ is being typed as `mut`. By $fieldGuarded(\sigma_0, e_0)$, the former step must have been inside a monitor $\texttt{M}(l; \mathcal{E}_v[l.f]; e)$ and the l under reduction was the only occurrence of l. Since f is a capsule, we know that $l \notin erog(\sigma, l)$ by Head Not Circular. Thus in our new step l is not *inside* $\mathcal{E}_v[v]$. Thus l must be *monitored* and hence it is *OK*.
    - Otherwise, l is still *OK*
  - Suppose some other l_0 was *wellEncapsulated* and *valid*:
    - If l was in the *rog* of l_0, by Capsule Tree, if l was in the *rog* of l, then v can only be reached from l_0 by passing through l, and so we could not have made l_0 non *wellEncapsulated*. In addition, since only things in the *erog* can be referenced by `invariant`, validity can not depend on l, and by Determinism it is still the case that l_0 is *valid*. And so we can't have effected l_0 being *OK*.
    - Otherwise, this reduction step could not have affected l_0, so l_0 is still *OK*.
  - Nothing that was *garbage* could have been made reachable by this expression, since the only value we produced was v and it was reachable through l (and so could not have been *garbage*), thus l is still *OK*.
  - As we don't change any monitors here, nothing that was *monitored* could have been made un-*monitored*, and so it is still *OK*.

3. (MCALL, TRY ENTER and TRY OK):
  These reduction steps do not modify memory, the memory locations reachable inside of main expression, or any monitor expressions. Therefore it cannot have any effect on the *garbage*, *wellEncapsulated*, *valid* (due to Determinism), or *monitored* properties of any memory locations, thus *OK* still holds.

4. (NEW) $\sigma|\texttt{new } C(\overline{v}) \to \sigma, l \mapsto C\{\overline{v}\}|\texttt{M}(l;l;l\texttt{.invariant()})$:
  Clearly the newly created object, l, is *monitored*. As for MCALL, other objects and properties are not disturbed, and so *OK* still holds.

5. (MONITOR EXIT) $\sigma|\texttt{M}(l;v;\texttt{true}) \to \sigma|v$:
  - As monitor expressions are not present in the original source code, it must have been introduced by UPDATE, MCALL, or NEW. In each case the 3$^{\text{rd}}$ expression started of as $l\texttt{.invariant()}$, and it has now (eventually) been reduced to `true`, thus by Determinism l is *valid*.

- If the monitor was introduced by UPDATE, then $v = l$. We must have had that l was well encapsulated before UPDATE was executed (since it can't have been *garbage* and *monitored*, as UPDATE itself preserves this property and we haven't modified memory in anyway, we must still have that l is *wellEncapsulated*. As l is *valid* and *wellEncapsulated*, it is *OK*.
- If the monitor was introduced by MCALL, then it was due to calling a capsule mutator method that mutated a field f.
  - A location that was *garbage* obviously still is, and so is also *OK*.
  - No location that was *valid* could have been made invalid since this reduction rule performs no mutation of memory. If a location was *wellEncapsulated* before, the only way it could be non *wellEncapsulated* is if we somehow leaked a **mut** reference to something, but by our well-formedness rules, v cannot be typed as **mut** and so we can't have affected *wellEncapsulated*, hence such thing is still *OK*.
  - The only location that could have been made un *monitored* is l itself. By our well formedness criteria, l was only used to modify $l.f$, and we have no parameters by which we could have made $l.f$ non *wellEncapsulated*, since that would violate Capsule Tree. As nothing else in l was modified, and it must have been *wellEncapsulated* before the MCALL, and so it still is. In addition since l is valid, it is *OK*.
- Otherwise the monitor was introduced by NEW. Since we require that **capsule** fields and **imm** fields are only initialised to **capsule** and **imm** expressions, by Capsule Tree, the resulting value, l, must be *wellEncapsulated*, since l is also *valid* we have that l is *OK*.

6. (TRY ERROR) $\sigma, \sigma_0 | \text{try}^\sigma \{error\} \text{ catch } \{e\} \rightarrow \sigma | e$:

By Strong Exception Safety, we know that σ_0 is *garbage* with respect to $\mathcal{E}_v[e]$. By our well formedness criteria, no location inside σ could have been *monitored*. Since we don't modify memory, everything in σ_0 is *garbage* and nothing inside σ was previously monitored, it is still clearly the case that everything in σ is *OK*.

B   The Hamster Example in Spec#

In this section we describe exactly why we chose to annotate the example from Section 1 in the way we did. For brevity, we will assume the default accessibility is **public**, whilst in both Spec# and C#, it is actually **private**.

The `Point` Class

The typical way of writing a `Point` class in C# is as follows:

```
class Point {
  double x, y;
  Point(double x, double y) { this.x = x; this.y = y; }
}
```

This works exactly as is in Spec#, however we have difficulty if we want to define equality of `Point`s (see below).

The `Hamster` Class

The `Hamster` class in C# would simply be:

```
class Hamster {
  Point pos;
  Hamster(Point pos) { this.pos = pos; }
}
```

1536   Though this is legal in Spec#, it is practically useless. Spec# has no way of knowing
1537 whether `pos` is *valid* or *consistent*. If `pos` is not known to be valid, one will be unable to
1538 pass it to almost any method, since by default methods implicitly require their receivers
1539 and arguments to be valid (compare this with our invariant protocol, which guarantees that
1540 any reachable object is valid). If `pos` is not known to be consistent, one will be unable to
1541 mutate it, by updating one of its fields or by passing it as an argument (or receiver) to a non
1542 **Pure** method. Though we don't want `pos` to ever mutate, Spec# currently has no way of
1543 enforcing that an *instance* of a non immutable class is itself immutable[23], as such we will
1544 simply refrain from mutating it.

1545   To enable Spec# to reason about `pos`'s validity, we will require that it be a *peer* of the
1546 enclosing **Hamster**; we can do this by annotating `pos` with **[Peer]**. Peers are objects that
1547 have the same owner, implying that whenever one is valid and/or consistent, the other one
1548 also is. This means that if we have a **Hamster**, we can use its `pos`, in the same ways as we
1549 could use the **Hamster**.

1550   To simplify instantiation of **Hamster**s, their constructors will take unowned **Point**s, Spec#
1551 will then automatically make such **Point** a peer. This is achieved by taking a **[Captured]**
1552 **Point** in the constructor (note how similar this is to taking a **capsule Point**). Note that
1553 unlike our system, this prevents multiple **Hamster**s from sharing the same **Point**, unless both
1554 **Hamster**s have the same owner, if **Point** were immutable, there would be no such restriction.

1555   With the aforementioned modifications, the **Hamster** becomes:

```
class Hamster {
  [Peer] Point pos;
  Hamster([Captured] Point pos) { this.pos = pos; }
}
```

1562   We don't want **Point** to be an immutable/value type, however if it were, the original
1563 unannotated version would not have any problems.

The `Cage` Class

1565 The natural way to write this class in C#, if it had native support for class invariants like
1566 Spec#, would be:

```
class Cage {
  Hamster h;
  List<Point> path;
  Cage(Hamster h, List<Point> path){this.h=h; this.path=path;}
  invariant this.path.Contains(this.h.pos);
  void Move() {
    int index = this.path.IndexOf(this.h.pos);
    this.h.pos = this.path[index % this.path.Count]; }
}
```

1578   However for the above **invariant** to be admissible in Spec#, **this**.path and **this**.h
1579 must both be owned by **this**. In addition, the *elements* of **this**.path need to be owned
1580 by **this**, since **this**.path.Conatains will read them. Note that **this**.h.pos also needs

---

[23] There is a the describes a simple solution to this problem: assign ownership of the object to a special
predefined 'freezer' object, which never gives up mutation permission [47], however this does not appear
to have been implemented; this would provide similar flexibility to the TM system we use, which allows
an initially mutable object to be promoted to immutable.

to be owned by **this**, however since pos is declared as **[Peer]**, if **this** owns **this**.h, it also owns **this**.h.pos. To fix the invariant, we will declare h, path, and the elements of path as *reps* (i.e. they are owned by the containing object). Finally, since **Move** modifies **this**.h, **this**.h needs to be made consistent, which requires that the owner (**this**) be made invalid; this can be achieved by using an **expose(this)** statement. **expose(this)**{*body*} marks **this** as invalid, executes *body*, checks that the invariant of **this** holds, and then marks **this** valid again. As we did with the **Hamster**, we will simply take unowned h and path values, however we also need the elements of path to be unowned; since Spec# has no **[ElementsCaptured]** annotation, we will require path to be unowned, and its elements (denoted by **Owner.ElementProxy(path)**) to be owned by the same owner as path (which is **null**).

```
class Cage {
  [Rep] public Hamster h;
  [Rep, ElementsRep] List<Point> path;

  Cage([Captured] Hamster h, [Captured] List<Point> path)
    requires Owner.Same(Owner.ElementProxy(path), path);
  { this.h = h; this.path = path; }

  invariant this.path.Contains(this.h.pos);
  void Move() {
    int index = this.path.IndexOf(this.h.pos);
    expose(this){this.h.pos=this.path[index%this.path.Count]; }}
}
```

The above constructor now fails to verify, since Boogie is unconvinced that its precondition actually holds when we initialise **this.path**. This is because the constructor for **Object** (the default base class if none is provided) is not marked as **[Pure]**; since it is (implicitly) called upon entry to **Cage**'s constructor, Boogie has no idea as to what memory could've mutated, and so it doesn't know whether the precondition still holds. The solution is to explicitly call it, but at the end of the constructor: {**this**.h = h; **this**.path = path; **base**();}.

The above **Cage** code however does not work, since **List** operations, such as **Contains** and **IndexOf**, will call the virtual **Object.Equals** method to compute equality of **Point**s. However **Object.Equals** implements *reference* equality, whereas we want *value* equality.

Defining Equality of **Points**

The obvious solution in C# is to just override **Object.Equals** accordingly, and let dynamic dispatch handle the rest:

```
class Point {
  .. // as before
  override bool Equals(Object? o) {
    Point? that = o as Point;
    return that!=null && this.x == that.x && this.y == that.y;}
}
```

However this fails in Spec# since **Object.Equals** is annotated with **[Pure]** **[Reads(ReadsAttribute.Reads.Nothing)]**, and of course every overload of it must also satisfy this. The **Reads** annotations specifies that the method cannot read fields of *any* object, not even the receiver, this makes overloading the method useless.

We resorted to making our own `Equal` method. Since it is called in `Cage`'s invariant, Spec# requires it to be annotated as [`Pure`], and either annotated with [`Reads(ReadsAttribute.Reads.Nothing)`] or [`Reads(ReadsAttribute.Reads.Owned)`] (the default, if the method is [`Pure`]). The latter annotation means it can only read fields of objects owned by the *receiver* of the method, so a [`Pure`] `bool` `Equal(Point that)` method can read the fields of `this`, but not the fields of `that`. Of course this would make the method unusable in `Cage` since the `Point`s we are comparing equality against do not own each other. As such, the simplest solution is to pass the fields of the other point to the method:

```
[Pure] bool Equal(double x, double y) {
  return x == this.x && y == this.y;}
```

Sadly however this mean we can no longer use `List`'s `Contains` and `IndexOf` methods, rather we have to expand out their code manually; making these changes takes us to the version we presented in Section 1.

C   More Case Studies

Family

The following test case was designed to produce a worst case in the number of invariant checks. We have a `Family` that (indirectly) contains a list of `parents` and `children`. The `parents` and `children` are of type `Person`. Both `Family` and `Person` have an invariant, the invariant of `Family` depends on its contained `Person`s.

```
class Person {
  final String name;
  Int daysLived;
  final Int birthday;
  Person(String name, Int daysLived, Int birthday) { .. }
  mut method Void processDay(Int dayOfYear) {
    this.daysLived += 1;
    if (this.birthday == dayOfYear) {
      Console.print("Happy birthday " + this.name + "!"); }}
  read method Bool invariant() {
    return !this.name.equals("") && this.daysLived >= 0 &&
      this.birthday >= 0 && this.birthday < 365; }
}
class Family {
  static class Box {
    mut List<Person> parents;
    mut List<Person> children;
    Box(mut List<Person> parents, mut List<Person> children){..}
    mut method Void processDay(Int dayOfYear) {
      for(Person c : this.children) { c.processDay(dayOfYear); }
      for(Person p : this.parents) { p.processDay(dayOfYear); }}
  }
  capsule Box box;
  Family(capsule List<Person> ps,capsule List<Person> cs) {
    this.box = new Box(ps, cs); }
```

```
1679  mut method Void processDay(Int dayOfYear) {
1680     this.box.processDay(dayOfYear); }
1681  mut method Void addChild(capsule Person child) {
1682     this.box.children.add(child); }
1683  read method Bool invariant() {
1684     for (Person p : this.box.parents) {
1685        for (Person c : this.box.children) {
1686           if (p.daysLived <= c.daysLived) {
1687              return false; }}}
1688     return true; }
1689
1690 }
```

Note how we created a **Box** class to hold the **parents** and **children**. Thanks to this pattern, the invariant only needs to hold at the end of **Family.processDay**, after all the **parents** and **children** have been updated. Thus **Family.processDay** is atomic: it updates all its contained **Person**s together. Had we instead made the **parents** and **children** **capsule** fields of **Family**, the invariant would be required to also hold between modifying the two lists. This could cause problems if, for example, a child was updated before their parent.

We have a simple test case that calls **processDay** on a **Family** 1,095 (3×365) times.

```
1699  // 2 parents (one 32, the other 34), and no children
1700  var fam = new Family(List.of(new Person("Bob", 11720, 40),
1701     new Person("Alice", 12497, 87)), List.of());
1702
1703  for (Int day = 0; day < 365; day++) { // Run for 1 year
1704     fam.processDay(day);
1705  }
1706  for (Int day = 0; day < 365; day++) { // The next year
1707     fam.processDay(day);
1708     if (day == 45) {
1709        fam.addChild(new Person("Tim", 0, day)); }}
1710
1711  for (Int day = 0; r < 365; day++) { // The 3rd year
1712     fam.processDay(day);
1713     if (day == 340) {
1714        fam.addChild(new Person("Diana", 0, day)); }}
1715
```

The idea is that everything we do with the **Family** is a mutation; the **fam.processDay** calls also mutate the contained **Person**s.

This is a worst case scenario for our approach compared to visible state semantics since it reduces our advantages: our approach avoids invariant checks when objects are not mutated but in this example most operations are mutations; similarly, our approach prevents the exponential explosion of nested invariant checks[24] when deep object graphs are involved, but in this example the object graph of **fam** is very shallow.

We ran this test case using several different languages: L42 (using our protocol) performs 4,000 checks, D and Eiffel perform 7,995, and finally, Spec# performs only 1,104.

---

[24] As happened in our GUI case study, see Section 7.

Our protocol performs a single invariant check at the end of each constructor, `processDay` and `addChild` call (for both **Person** and **Family**).

The visible state semantics of both D and Eiffel perform additional invariant checks at the beginning of each call to `processDay` and `addChild`.

The results for Spec# are very interesting, since it performs less checks than L42. This is the case since `processDay` in **Person** just does a simple field update, which in Spec# do not invoke runtime invariant checks. Instead, Spec# tries to statically verify that the update cannot break the invariant; if it is unable to verify this, it requires that the update be wrapped in an **expose** block.

Spec# relies on the absence of arithmetic overflow, and performs runtime checks to ensure this[25], as such the verifier concludes that the field increment in `processDay` cannot break the invariant. Spec# is able to avoid some invariant checks in this case by relying on all arithmetic operations performing runtime overflow checks; whereas integer arithmetic in L42 has the common wrap around semantics.

The annotations we had to add in the Spec# version[26] were similar to our previous examples, however since the fields of **Person** all have immutable classes/types, we only needed to add the invariant itself. The **Family** class was similar to our **Cage** example (see section 1), however in order to implement the `addChild` method we were forced to do a shallow clone of the new child (this also caused a couple of extra runtime invariant checks). Unlike L42 however, we did not need to create a box to hold the **parents** and **children** fields, instead we wrapped the body of the **Family.processDay** method in an **expose** (**this**) block. In total we needed 16 annotations, worth a total of 45 tokens, this is worse than the code following our approach that we showed above, which has 14 annotations and 14 tokens.

Spec# Papers

Their are many published papers about the pack/unpack methodology used by Spec#. To compare against their expressiveness we will consider the three mains ones that introduced their methodology and extensions:

- *Verification of Object-Oriented Programs with Invariants:* [5] this paper introduces their methodology. In their examples section (pages 41–47), they show how their methodology would work in a class heirarchy with **Reader** and **ArrayReader** classes. The former represents something that reads characters, whereas the latter is a concrete implementation that reads from an owned array. They extend this further with a **Lexer** that owns a **Reader**, which it uses to read characters and parse them into tokens. They also show an example of a **FileList** class that owns an array of filenames, and a **DirFileList** class that extends it with a stronger invariant. All of these examples can be represented in L42[27]. The most interesting considerations are as follow:
  - Their **ArrayReader** class has a `relinquishReader` method that 'unpacks' the **ArrayReader** and returns its owned array. The returned array can then be freely mutated and passed around by other code. However, afterwards the **ArrayReader** will be 'invalid', and so one can only call methods on it that do not require its invariant to hold. However, it may later be 'packed' again (after its invariant is checked). In contrast, our approach requires the invariant of all usable objects to hold. We can still relinquish the array, but at the cost of making the **ArrayReader** forever unreachable. This can be done by declaring `relinquishReader` as a **capsule method**, this works since our type

---

[25] Runtime checks are enabled by a compilation option; when they fail, unchecked exceptions are thrown.
[26] The Spec# code is in the artifact.
[27] Our encodings are in the artifact.

modifier system guarantees that the receiver of such a method is not aliased, and hence cannot be used again. Note that Spec# itself cannot represent the `relinquishReader` method at all, since it does not provide explicit pack and unpack operations, rather its **expose** statement performs both an unpack and a pack, thus we cannot unpack an **ArrayReader** without repacking it in the same method.

-   Their **DirFileList** example inherits from a **FileList** which has an invariant, and a final method, this is something their approach was specifically designed to handle. As L42 does not have traditional subclassing, we are unable to express this concept fully, but L42 does have code reuse via trait composition, in which case **DirFileList** can essentially copy and paste the methods from **FileList**, and they will automatically enforce the invariant of **DirFileList**.

-   *Object Invariants in Dynamic Contexts:* [46] this paper shows how one can specify an invariant for a doubly linked list of **int**s (which is an immutable value type). Unlike our protocol however, it allows the invariant of **Node** to refer to sibling **Node**s which are not owned/encapsulated by itself, but rather the enclosing **List**. Our protocol can verify such a linked list[28] (since its elements are immutable), however we have to specify the invariant inside the **List** class. We do not see this as a problem, as the **Node** type is only supposed to be used as part of a **List**, thus this restriction does not impact users of **List**.

-   *Friends Need a Bit More: Maintaining Invariants Over Shared State:* [6] this paper shows how one can verify invariants over interacting objects, where neither owns/contains the other. They have multiple examples which utilise the 'subject/observer' pattern, where a 'subject' has some state that an 'observer' wants to keep track of. In their **Subject**/**View** example, **View**s are created with references to **Subject**s, and copies of their state. When a **Subject**'s state is modified, it calls a method on its attached **View**s, notifying them of this update. The invariant is that a **View**'s copy of its **Subject**'s state is up to date. Their **Master**/**Clock** example is similar, a **Clock** contains a reference to a **Master**, and saves a copy of the **Master**'s time. The **Master** has a **Tick** method that increases its time, but unlike the **Subject**/**View** example, the **Clock** is not notified. The invariant is that the **Clock**'s time is never ahead of its **Master**'s. Our protocol is unable to verify these interactions, because the interacting objects are not immutable or encapsulated by each other.

---

[28] Our protocol allows for encoding this example, but to express the invariant we would need to use reference equality, which the L42 language does not support.