Using Capabilities for Strict Runtime Invariant Checking

Isaac Oscar Gariano, Marco Servetto*, Alex Potanin

Victoria University of Wellington, Kelburn, 6012, Wellington, New Zealand

Abstract

In this paper we use pre-existing language support for both reference and object capabilities to enable sound runtime verification of representation invariants. Our invariant protocol is stricter than the other protocols, since it guarantees that invariants hold for all objects involved in execution. Any language already offering appropriate support for reference and object capabilities can support our invariant protocol with minimal added complexity. In our protocol, invariants are simply specified as methods whose execution is statically guaranteed to be deterministic and to not access any externally mutable state. We formalise our approach and prove that our protocol is sound, in the context of a language supporting mutation, dynamic dispatch, exceptions, and non-deterministic I/O. We present case studies showing that our system requires a lighter annotation burden compared to Spec#, and performs orders of magnitude less runtime invariant checks compared to the 'visible state semantics' protocols of D and Eiffel. [Isaac: When we said "widlely" used, we mean visible state semantics is, not the D and Eiffel languages]

Keywords: reference capabilities, object capabilities, runtime verification, class invariants

1. Introduction

Representation invariants (sometimes called class invariants or object invariants) are a useful concept when reasoning about software correctness in OO (Object Oriented) languages. Such invariants are predicates on the state of an object and its ROG (Reachable Object Graph). 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 method called `invariant()` that returns a boolean. We say that an object's invariant holds when its `invariant()` method would return **true**.[1]

Invariants are designed to hold most of the time, however it is commonly required to (temporarily) violate invariants while performing complex sequences of mutations. To support this behaviour, most invariant protocols present in the literature allow invariants to be broken and observed broken. The two main forms of invariant protocols are *visible state semantics* [2] and the *Pack-Unpack/Boogie methodology* [3]. In visible state semantics, invariants can be broken when a method on the object is active (that is, currently executing). Some interpretations of the visible state are more permissive, requiring the invariants of receivers to hold only before and after every public method call, and after constructors. In the pack-unpack approach, objects are either in a 'packed' or 'unpacked' state, the invariant of 'packed' objects must hold, whereas unpacked objects can be broken.

In this paper we propose a much stricter invariant protocol: at all times, the invariant of every object involved in execution must hold; thus they can be broken when the object is not (currently) involved in

*Corresponding Author

Email addresses: isaac@ecs.vuw.ac.nz (Isaac Oscar Gariano), marco.servetto@ecs.vuw.ac.nz (Marco Servetto), alex@ecs.vuw.ac.nz (Alex Potanin)

[1] We do this (as in Dafny [1]) to minimise the special treatment of invariants, whereas other approaches often treat invariants as a special annotation with its own syntax.

execution. An object is *involved in execution* when it is in the ROG of any of the objects mentioned in the method call, field access, or field update that is about to be reduced; we state this more formally later in the paper.

Our strict protocol supports easier reasoning: an object can never be observed broken. However at first glance it may look overly restrictive, preventing useful program behaviour. Consider the iconic example of a `Range` class, with a `min` and `max` value, where the invariant requires that `min<max`:

```
class Range{
  private field min; private field max;
  method invariant(){ return min<max; }
  method set(min, max){
    if(min>=max){ throw new Error(/**/); }
    this.min = min;
    this.max = max;
  }
}
```

In this example we omit types to focus on the runtime semantics. The code of `set` does not violate visible state semantics: `this.min = min` may temporarily break the invariant of `this`, however it will be fixed after executing `this.max = max`. Visible state allows such temporary breaking of invariants since we are inside a method on `this`, and by the time it returns, the invariant will be re-established. However, if `min` is $\geq$ `this.max`, `set` will violate our stricter approach. The execution of `this.min = min` will break the invariant of `this` and `this.max = max` would then involve a broken object. If we were to inject a call `Do.stuff(this);` between the two field updates, arbitrary user code could observe a broken object; adding such a call is however allowed by visible state semantics.

Using the *box pattern*, we can provide a modified `Range` class with the desired client interface, while respecting the principles of our strict protocol:

```
class BoxRange{//no invariant in BoxRange
  field min; field max;
  BoxRange(min, max){ this.set(min, max); }
  method Void set(min, max){
    if(min>=max){ throw new Error(/**/); }
    this.min = min; this.max = max;
  }
}
class Range{
  private field box; //box contains a BoxRange
  Range(min, max){ this.box = new BoxRange(min, max); }
  method invariant(){ return this.box.min < this.box.max; }
  method set(min, max){ return this.box.set(min,max); }
}
```

The code of `Range.set(min,max)` does not violate our protocol. The call to `BoxRange.set(min,max)` works in a context where the `Range` object is unreachable, and thus not involved in execution. That is, the `Range` object is not in the ROG of the receiver or the parameters of `BoxRange.set(min,max)`. Thus `Range.set(min,max)` can temporarily break the `Range`'s invariant. By using the `box` field as an extra level of indirection, we restrict the set of objects involved in execution while the state of the object `Range` is modified.[2] With appropriate type annotations, the code of `Range` and `BoxRange` is accepted as correct by our system: no matter how `Range` objects are used, a broken `Range` object will never be involved in execution.

[2] Due to its simplicity and versatility, we do not claim this pattern to be a contribution of our work, as we expect others to have used it before. We have however not been able to find it referenced with a specific name in the literature, though technically speaking, it is a simplification of the Decorator pattern, but with a different goal. While in very specific situations the overhead of creating such additional box object may be unacceptable, we designed our work for environments where such fine performance differences are negligible. Also note that many VMs and compilers can optimize away wrapper objects in many circumstances. [4] This is even more applicable in languages with inlined structs, like C++ or C#.

Contributions

Invariant protocols allow for objects to make necessary changes that might make their invariant temporarily broken. In visible state semantics any object that has an active method call anywhere on the call stacks is potentially invalid; arguably not a very useful guarantee as observed by Gopinathan *et al.*'s work. [5] Approaches such as *pack/unpack* [3] represent potentially invalid objects in the type system; this encumbers the type system and the syntax with features whose only purpose is to distinguish objects with broken invariants. The core insight behind our work is that we can use a small number of decorator-like design patterns to avoid exposing those potentially invalid objects in the first place, thus avoiding the need for representing them at the type level.

In the remainder of this paper, we discuss how to combine runtime checks and capabilities to soundly enforce our strict invariant protocol. Our solution only requires that all code is well-typed, and works in the presence of mutation, I/O, non-determinism, and exceptions, all under an open world assumption.

We formalise our approach and, in Appendix A, prove that our use of Reference and Object Capabilities soundly enforces our invariant protocol.

We have fully implemented our protocol in L42[3], we used this implementation to implement many case studies, showing that our protocol is more succinct than the pack/unpack approach and much more efficient then the visible state semantic. It is important to note that unlike most prior work, we soundly handle catching of invariant failures and I/O. We describe our case studies in Section 6. Our approach may seem very restrictive; the programming patterns in Section 7 show how our approach does not hamper expressiveness; in particular we show how batch mutation operations can be performed with a single invariant check, and how the state of a 'broken' object can be safely passed around.

2. Background on Reference and Object Capabilities

Reasoning about imperative 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; instead of using automated theorem proving, it is becoming more popular to verify aliasing and immutability properties using a type system. For example, three languages: L42 [6, 7, 8, 9], Pony [10, 11], and the language of Gordon *et al.* [12] use RCs (Reference Capabilities)[4] and OCs (Object Capabilities) to statically ensure deterministic parallelism and the absence of data races. While studying those languages, we discovered an elegant way to enforce invariants: we use capabilities to restrict how/when the result of invariant methods changes; this is done by restricting I/O, and how mutation through aliases can affect the state seen by invariants.

Reference Capabilities

RCs, as used in this paper, are a type system feature that allows reasoning about aliasing and mutation. A more recent 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.*, Pony, and L42. 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 RCs of these languages are used to provide automatic and correct parallelism [12, 10, 11, 7].

Reference capabilities are a well known mechanism [13, 14, 15, 10, 9, 12] that allow statically reasoning about the mutability and aliasing properties of objects. Here we refer to the interpretation of [12], that introduced the concept of recovery/promotion. This concept is the basis for L42, Pony, and Gordon *et al.*'s type systems [12, 7, 6, 10, 11]. With slightly different names and semantics, those languages all support the following RCs for object references:

[3]Our implementation is implemented by checking that a given class conforms to our protocol, and injecting invariant checks in the appropriate places. An anonymised 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/InvariantArtifact.zip`.

[4]RCs are called *Type Modifiers* in former works on L42.

- Mutable (**mut**): the referenced object can be mutated and shared/aliased without restriction; as in most imperative languages without reference capabilities.

- Immutable (**imm**): the referenced object cannot mutate, not even through other aliases. An object with any **imm** aliases is an *immutable object*. Any other object is a *mutable object*. All objects are born mutable and may later become immutable. Thus, an object can be classified as *mutable* even if it has no fields that can be updated or mutated.

- Readonly (**read**): the referenced object cannot be mutated by such references, but there may also be mutable aliases to the same object, thus mutation can be observed. Readonly references can refer to both mutable and immutable objects, as **read** types are supertypes of both their **imm** and **mut** variants.

- Encapsulated (**capsule**): every mutable object in the ROG of a capsule reference (including itself) is only reachable through that reference. Immutable objects in the ROG of a capsule reference are not constrained, and can be freely referred to without passing through that reference.

That is, there are only two kinds of objects: mutable and immutable, but there are more kinds of RCs.

RCs are different to field or variable qualifiers like Java's **final**: RCs 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.[5]

RCs are applied to all types. This includes types in the receiver and parameters of methods. A **mut** method is a method where **this** is typed **mut**; An **imm** method is a method where **this** is typed **imm**, and so on for all the other RCs.

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;
//rp.x = 3; // type error
mp.x = 5; // ok, now we can observe rp.x == 5
ip = new Point(3, 5); // ok, ip is not final
```

RCs influence the access to the whole ROG; not just the referenced object itself, as in the full/deep interpretation of type modifiers [16, 17]:

- A **mut** field accessed from a **read** reference produces a **read** reference; thus a **read** reference cannot be used to mutate the ROG of the referenced object.

- Any field accessed from an **imm** reference produces an **imm** reference; thus all the objects in the ROG of an immutable object are also immutable.

A common misconception of this line of work is that a **mut** field will always refer to a mutable object. Classes declare RCs for their methods and field types, but what kinds of object is stored in a field also depends on the kind of the object: a **mut** field of a mutable object will contain a mutable object; but a **mut** field of an immutable object will contain an immutable object. This is different with respect to work prior to Gordon *et al.*'s [12], where the declaration fully determines what values can be stored. In those other approaches, any contextual information must be explicitly passed through the type system, for example, with a generic RC parameter.

[5] 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`.

Another common misconception is the belief that **capsule** fields and **capsule** local variables always hold **capsule** references, i.e. the referenced object cannot be reached except via that field/variable. How **capsule** local variables are handled differs widely in the literature:

In L42, a **capsule** local variable always holds a **capsule** reference: this is ensured by allowing them to be read only once (similar to linear and affine types [18]). Pony and Gordon *et al.* follow a more complicated approach: **capsule** variables can be accessed multiple times, however in those cases the result will not be a **capsule** reference but another kind of reference, that can be promoted to **capsule**, but only under certain conditions. Pony and Gordon also provide destructive reads, where the variable's old value is returned as **capsule**. Like **capsule** variables, how **capsule** fields are handled differs widely in the literature, however they must always be initialised and updated with **capsule** references. In order for access to a **capsule** field to safely produce a **capsule** reference, Gordon *et al.* only allows them to be read destructively (i.e. by replacing the field's old value with a new one, such as **null**). In contrast, Pony does not guarantee that **capsule** fields contain a **capsule** reference at all times, as it provides non-destructive reads. L42 is even more radical: an L42 **capsule** field never contains a **capsule** reference; it is simply initialised with one. [7, 19] Pony and L42's **capsule** fields are useful for safe parallelism, but not invariant checking. [6]

In Section 3 we present a novel kind of **capsule** field useful for invariant checking; we added support for these fields to L42, and believe they could be easily added to Pony and Gordon *et al.*'s language.

Promotion and Recovery

Many different techniques and type systems handle the RCs above [16, 20, 21, 12, 6]. 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 [12, 10, 6] the reference capability, adapting them to their use context. To see a glimpse of this flexibility, consider the following:

```
    mut Circle mc = new Circle(new Point(0, 0), 7);
capsule Circle cc = new Circle(new Point(0, 0), 7);
    imm Circle ic = new Circle(new Point(0, 0), 7);
```

Here `mc`, `cc`, and `ic` are all syntactically initialised with the same exact expression. All **new** expressions return a **mut** [10, 19], so `mc` is well typed. The declarations of `cc` and `ic` are also well typed, since any expression (not just **new** expressions) of a **mut** type that has no **mut** or **read** free variables can be implicitly promoted to **capsule** or **imm**. This requires the absence of **read** and **mut** *global/static* variables, as in L42, Pony, and Gordon *et al.*'s language. This is the main improvement on the flexibility of RCs in recent literature [7, 6, 12, 10, 11]. From a usability perspective, this improvement means that these RCs are opt-in: a programmer can write many classes simply using **mut** types and be free to have rampant aliasing. Then, at a later stage, another programmer may still be able to encapsulate instances of those data structures into an **imm** or **capsule** reference.

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 while mutating an object, what state is that object in? Does its invariant hold? The concept of *strong exception safety* [22, 8] 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; this prevents exposing objects whose invariant was left broken in the middle of mutations.

L42 enforces strong exception safety for unchecked exceptions using RCs[7] in the following way:[8]

- Code inside a **try** block that captures unchecked exceptions is typed as if all variables declared outside

[6]It may seem surprising that those weaker forms of encapsulation are still sufficient to ensure safe parallelism. The detailed way L42 parallelism works is unrelated to the presented work. Please see `L42.is/tutorial.xhtml` (specifically, section 5 and 6) for more information on parallelism in L42.

[7]This is needed to support safe parallelism. Pony takes a drastic approach and not support exceptions. We are not aware of how Gordon *et al.* handles exceptions, however to have sound unobservable parallelism it must have some restrictions.

[8]Formal proof that these restriction are sufficient is in the work of Lagorio and Servetto [8].

of the block are `final` and all those of a `mut` type were `read`.

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

This strategy does not restrict when exceptions can be *thrown*, but only restricts when unchecked exceptions can be *caught*. Strong exception safety allows us to throw invariant failures as unchecked exceptions: if an object's ROG was mutated into a broken state within a try block, when the invariant failure is caught, the mutated object will be unreachable/garbage-collectable. This works since strong exception safety guarantees that no object mutated within a try block is visible when it catches an unchecked exception.[9]

Similarly to Java, L42 distinguishes between checked and unchecked exceptions, and `try`-`catch`es over checked exceptions impose no limits on object mutation during the `try`. That is, strong exception safety is only enforced for unchecked exceptions.

Object Capabilities

OCs, which L42, Pony, and Gordon *et al.*'s work have, are a widely used [23, 24, 25] programming technique where access rights to resources are encoded as references to objects. When this style is respected, code unable to reach a reference to such an object cannot use its associated resource. Here, as in Gordon *et al.*'s work, we enforce the OC pattern with RCs in order 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 by instance methods of classes whose instantiation is kept under control by carefully designing their implementation.

For example, in Java, `System.in` is a *capability object* that provides access to the standard input resource. However, since 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

```
public static void main(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(s)`. This design has been explored by Joe-E [26].

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, L42 has no predefined standard library, but many can be defined by the community. Thus, the only way to perform I/O operations is via foreign function calls. Since enforcing the OC pattern can not be done via a unique standard library, the type system of L42 directly enforces the OC pattern as follows:

- Foreign methods (which have not been whitelisted as deterministic) and methods whose names start with #$ are *capability operations*.
- Classes containing capability operations are *capability classes*.
- Constructors of capability classes are also *capability operations*.
- Capability operations can only be called by other capability operations or `mut`/`capsule` methods of capability classes.
- In L42 there is no `main` method, rather it has several *main expressions*; such expressions can also call capability operations, thus they can instantiate OCs and pass them around to the rest of the program.

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

3. Our Invariant Protocol

All classes contain a `read method Bool invariant() {..}`, if no `invariant()` method is explicitly present, a trivial one returning `true` is assumed.

Our protocol guarantees that the whole ROG of any object involved in execution (formally, in a redex) is *valid*: if you can use an object, *manually* calling `invariant()` on it is guaranteed to return `true` in a finite number of steps.[10]

As the `invariant()` is used to determine whether `this` is broken, it may receive a broken `this`; however this will only occur for calls to `invariant()` inserted by our approach. User written calls to `invariant()` are guaranteed to receive a valid `this`.

We restrict `invariant()` methods so that they represent a predicate over the receiver's `imm` and `capsule` fields. To ensure that `invariant()` methods do not expose a potentially broken `this` to the other objects, we require that all occurrences of `this`[11] in the `invariant()`'s body are the receiver of a field access (`this.f`) of an `imm`/`capsule` field, or the receivers of a method call (`this.m(..)`) of a final (non-virtual) method that in turn satisfies these restrictions. No other uses of `this` are allowed, such as as the right hand side of a variable declaration, or an argument to a method. An equivalent alternative design could instead rely on static `invariant(..)` methods taking each `imm` and `capsule` field as a parameter.

Invariants can only refer to immutable and encapsulated state. Thus 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. We do not make it harder to correctly implement such data structures, but the `invariant()` method is unable to access the list's nodes, since they may contain `mut` references to shared/unencapsulated objects. There is a line of work [27] striving to allow invariants over other forms of state. We have not tried to integrate such solutions into our work, as we believe it would make our system more complex and ad hoc, probably requiring numerous specialised kinds of RCs. Thus we have traded some expressive power in order to preserve safety and simplicity.

Purity

L42's enforcement of RCs and OCs statically guarantees that any method with only `read` or `imm` parameters (including the receiver) is *pure*; we define pure as being deterministic and not mutating existing memory. This holds because (1) the ROG of the parameters (including `this`) is only accessible as `read` (or `imm`), thus it cannot be mutated (2) if a capability object is in the ROG of any of the arguments (including the receiver), then it can only be accessed as `read`, preventing calling any non-deterministic (capability) methods; (3) no other pre-existing objects are accessible (as L42 does not have global variables). In particular, this means that our `invariant()` methods are pure, since their only parameter (the receiver) is `read`.

Capsule Fields

Former work on L42 discusses "depending on how we expose the owned data, we can closely model both *owners-as-dominators*[...] and *owners-as-qualifiers*[...]"[19], and "`lent` getter[s], a third variant"[19].

Those informal considerations have then influenced the L42 language design, bringing to the creation of syntactic sugar and programming patterns to represent various kinds of `capsule` fields aimed to model various forms of ownership. Under the hood, all those forms of `capsule` fields are just private `mut` fields with some extra restrictions. Describing in the details those restrictions would be outside of the scope of this paper.

Here we present a novel kind of `capsule` field[12] (which can coexist with other kinds of `capsule` fields), enforcing the following key property: the ROG of a capsule field $o.f$ can only be mutated under the control of a `mut` method of o, and during such mutation, o itself cannot be seen. This is similar to owner-as-modifier [28, 29], where we could consider an object to be the 'owner' of all the mutable objects in the ROG of its `capsule` fields, but with the extra restriction that the owner is unobservable during mutation of this ROG.

[10] We will show later how we satisfy this constraint without solving the halting problem or requiring all `invariant()` methods to be total.

[11] Some languages allow the `this` receiver to be implicit. For clarity in this work we require `this` to be always used explicit.

[12] As for the other kinds of `capsule` fields, our new kind is also just a private `mut` fields with extra restrictions.

More precisely, if a reference to an object in the ROG of a capsule field $o.f$ is involved in execution as `mut`, then: (1) no reference to o is involved in execution, (2) a call to a `mut` method for o is present in a previous stack frame, and (3) mutable references to the ROG of $o.f$ are not leaked out of such method execution, either as return values, exception values, or stored in the ROG of a parameter, or in any other field of the method's receiver.

To show how our `capsule` fields ensure these properties, we first define some terminology: `x.f` is a *field access*, `x.f=e` is a *field update*,[13] a `mut` method with a field access on a capsule field of `this` is a *capsule mutator*. Note that a method performing a field *update* of a `capsule` field (instead of a field access) is not a capsule mutator, but just a normal method performing a field update. Capsule mutators handle the more subtle case where the fields of an object with invariant are not updated, but a mutation deep within their ROG may potentially break the invariant.

The following rules define our novel `capsule` fields:

- A `capsule` field can only be initialised/updated using the result of an expression with `capsule` type.
- A `capsule` field access will return a:
  - `mut` reference, when accessed on `this` within a capsule mutator,
  - `read` reference, when accessed on any other `mut` receiver,
  - `imm` if the receiver is `imm`, `read` if the receiver is `read`, or `capsule` if the receiver is `capsule`. This last case is safe since a `capsule` receiver object will then be garbage collectable, so we do not need to preserve its invariant.
- A capsule mutator must:
  - use `this` exactly once: to access the `capsule` field,
  - have no `mut` or `read` parameters (except the `mut` receiver),
  - not have a `mut` return type,
  - not throw any checked exceptions[14].

The above rules ensure that capsule mutators control the mutation of the ROG of capsule fields, and ensures our points (1), (2), and (3): o will not be in the ROG of $o.f$ and only a capsule mutator on o can see $o.f$ as `mut`; this means that the only way to mutate the ROG of $o.f$ is through such methods. If execution is (indirectly) in a capsule mutator, than o is only used as the receiver of the `this.f` expression in the capsule mutator. Thus we can be sure that the ROG of $o.f$ will only be mutated within a capsule mutator, and only after the single use of o to access $o.f$. Since such mutation could invalidate the invariant of o, we call the `invariant()` method at the end of the capsule mutator body; before o can be used again. Provided that the invariant is re-established before a capsule mutator returns, no invariant failure will be thrown, even if the invariant was temporarily broken *during* the body of the method.

These properties are stronger then those of the pre-existing `capsule` fields of L42, but still *weaker* than those of `capsule` *references*: we do not need to prevent arbitrary `read` aliases to the ROG of a `capsule` field, and we do allow arbitrary `mut` aliases to exist during the execution of a capsule mutator. In particular, our rules allow unrestricted read only access to our `capsule` fields.

Runtime Monitoring

The language runtime will automatically perform calls to `invariant()`, if such a call returns `false`, an unchecked exception will be thrown. Such calls are performed at the following points:

- After a constructor call, on the newly created object.

[13] Thus a field update `x.f=e` is not a field access followed by an assignment.

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

- After a field update, on the receiver.
- After a capsule mutator method returns, on the receiver of the method[15].

In Section 5, we show that these checks, together with our aforementioned restrictions, are sufficient to ensure our guarantee that the invariants of all objects involved in execution hold.

Traditional Constructors and Subclassing
L42 constructors directly initialise all the fields using the parameters, and L42 does not provide traditional subclassing. This works naturally with our invariant protocol. We can support traditional constructors as in Pony and Gordon *et al.*'s language, by requiring that constructors only use `this` as the receiver of a field initialisation. Subclassing can be supported by forcing that a subclass invariant method implicitly starts with a check that `super.invariant()` returns `true`. We would also perform invariant checks at the end of `new` expressions, as happens in [30], and not at the end of `super(..)` constructor calls.

4. Essential Language Features

Our invariant protocol relies on many different features and requirements. In this section we will show examples of using our system, and how relaxing any of our requirements would break the soundness of our protocol. In our examples and in L42, the reference capability `imm` is the default, and so it can be omitted. 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 emphasise this. To save space, we omit the bodies of constructors that simply initialise fields with the values of the constructor's parameters, but we show their signature in order to show any annotations.

First we consider `Person`: it has a single immutable (and non final) field `name`.

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

The `name` field is not final: `Person`s can change state during their lifetime. The ROGs of all of a `Person`'s fields are immutable, but `Person`s themselves may be mutable. We enforce `Person`'s invariant by generating checks on the result of calling `this.invariant()`: immediately after each field update, and at the end of the constructor. Such checks are generated/injected, and not directly written by the programmer.

```
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(...); }
    }
}
```

We now show how if we were to relax (as in Rust), or even eliminate (as in Java), the support for OCs, RCs, or strong exception safety, the above checks would not be sufficient to enforce our invariant protocol.

[15]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.

Unrestricted Access to Capability Objects?

Allowing invariant() methods to (indirectly) perform non-deterministic operations by creating new capability objects or mutating existing ones would break our guarantee that (manually) calling invariant() always returns **true**. Consider this use of person; where myPerson.invariant() may randomly return **false**:

```
class EvilString extends String {//INVALID EXAMPLE
  @Override read method Bool isEmpty() { return new Random().bool(); }
}//Creates a new object capability out of thin air
...
method mut Person createPersons(String name) {
  // we can not be sure that name is not an EvilString
  mut Person schrodinger = new Person(name); // exception here?
  assert schrodinger.invariant(); // will this fail?
  ...}
```

Despite the code for **Person**.invariant() intuitively looking correct and deterministic (!name.isEmpty()), 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 object capabilities, and restricts their creation, the above example is prevented.

Moreover, since our system allows non-determinism only through **mut** methods on object capabilities, even if an object has a **capsule** field referring to a "file" object, it would be unable to read such file during an invariant, since a **mut** reference would be required, but only a **read** reference would be available.

Allowing Internal Mutation Through Back Doors?

Rust [31] and Javari [13] allow interior mutability: the ROG of an 'immutable' object can be mutated through back doors. Such back doors would allow invariant() methods to store and read information about previous calls. The example class **MagicCounter** breaks determinism by remotely breaking the invariant of person without any interaction with the person object itself:

```
class MagicCounter {//INVALID EXAMPLE
  Int counter = 0;
  method Int incr(){return unsafe{counter++}; /*using internal mutability*/}}
class NastyS extends String {..
  MagicCounter c = new MagicCounter(0);
  @Override read method Bool isEmpty(){return this.c.incr()!=2;}}
...
NastyS name = new NastyS(); //RCs believe name's ROG is immutable
Person person = new Person(name); // person is valid, counter=1
name.incr(); // counter == 2, person is now broken
person.invariant(); // returns false, counter == 3
person.invariant(); // returns false, counter == 4
```

Such back doors are usually motivated by performance reasons, however in [12] they discuss how a few trusted language primitives can be used to perform caching and other needed optimisations, without the need for back doors.

No Strong Exception Safety?

The ability to catch and recover from invariant failures allows programs to take corrective actions. 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 [18]. This property ensures 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. The following code is ill-typed since we try to mutate bob in a **try-catch** block that captures all unchecked exceptions; thus also including invariant failures:

```
mut Person bob = new Person("Bob");//INVALID EXAMPLE
```

```
// Catch and ignore invariant failure:
try { bob.name(""); } catch (Error t) { }// bob mutated
assert bob.invariant(); // fails!
```

The following variant is instead well typed, since `bob` is now declared inside of the **try** and it is guaranteed to be garbage collectable after the **try** is completed.

```
try { mut Person bob = new Person("Bob");     bob.name(""); }
catch (Error t) { }
```

Note how soundly catching exceptions like stack overflows or out of memory cannot be allowed in `invariant()` methods, since they are not deterministically thrown. L42 allows catching them only as a capability operation, which thus can't be used inside an invariant.

Relaxing restrictions on capsule fields?

Capsule fields allow expressing invariants over mutable object graphs. 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
  }
}
```

We inject calls to `invariant()` at the end of the constructor and the `addItem(item)` method. This is safe since the `items` field is declared **capsule**. Relaxing our system to allow a **mut** RC for the `items` field and the corresponding constructor parameter would make the above checks insufficient: it would be possible for external code with no knowledge of the **ShippingList** to mutate its items. In order to write correct library code in mainstream languages like Java and C++, defensive cloning [32] is needed. For performance reasons, this is hardly done in practice and is a continuous source of bugs and unexpected behaviour.

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

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 of `items` at initialisation time, such control may be lost later, and code unaware of the **ShippingList** could break it:

```
//INVALID EXAMPLE: l.items can be exposed as mut
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!
```

Relaxing our requirements for capsule mutators would break our protocol: if capsule mutators could have a **mut** return type the following would be accepted:

```
//INVALID EXAMPLE: capsule mutator expose(c) return type is mut
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 would allow the following code, where **this** may be reachable from `f`, thus `f.hi()` may observe an object that does not satisfying its invariant:

```
mut method Void multiThis(C c) {//INVALID EXAMPLE: two 'this'
  read Foo f = c.foo(this);
```

```
  this.items.add(new HeavyItem());
  f.hi(); }//‘this’ could be observed here if it is in ROG(f)
```

In order to ensure that a second reference to **this** is not reachable through arguments to such methods, we only allow **imm** and **capsule** parameters. Accepting a **read** parameter, as in the example below, would cause the same problems as before, where `f` may contain a reference to **this**:

```
mut method Void addHeavy(read Foo f) {//INVALID EXAMPLE
  this.items.add(new HeavyItem());
  f.hi(); }//‘this’ could be observed here if it is in ROG(f)
...
mut ShippingList l = new ShippingList(new Items());
read Foo f = new Foo(l);
l.addHeavy(f); // We pass another reference to ‘l’ through f
```

5. Formal Language Model

To model our system we need to formalise an imperative OO language with exceptions, object capabilities, and type system support for reference capabilities and strong exception safety. Formal models of the runtime semantics of such languages are simple, but defining and proving the correctness of such a type system is quite complex, and indeed many such papers exist that have already done this [7, 6, 12, 10, 8]. Thus we parametrise our language formalism, and assume we already have an expressive and sound type system enforcing the properties we need, so that we can separate our novel invariant protocol, from the non-novel reference capabilities. We clearly list in Appendix A the requirements we make on such a type system, so that any language satisfying them can soundly support our invariant protocol. In Appendix B we show an example type system, a restricted subset of L42, and prove that it satisfies our requirements. Conceptually our approach can parametrically be applied to any type system supporting these requirements, for example you could extend our type system with additional promotions or generic. To keep our small step reduction semantics as conventional as possible, we base our formalism on Featherweight Java [33, 34, Chapter 19], which is a turing complete [35] minimalistic subset of Java. As such, we model an OO language where receivers are always specified explicitly, and the receivers of field accesses and updates in method bodies are always **this**; that is, all fields are instance-private. Constructor declarations are not present explicitly, instead we assume they are all of the form C`(`$T_1\,x_1, \ldots, T_n\,x_n$`){`**this**.$f_1$ `=` x_1`;`…`;`**this**.f_n `=` x_n`}`, for appropriate types $T_1, \ldots, T_n$. Note that we do not model variable updates or traditional subclassing, since this would make the proofs more involved without adding any additional insight.

Notational Conventions

We use the following notational conventions:

- Class, method, parameter, and field names are denoted by C, m, x, and f, respectively.
- We use "ls" and "vs" as metavariables denoting a sequence of form $v_1, \ldots, v_n$ and $l_1, \ldots, l_n$, similarly with other metavariables ending in "s".
- We use "_" to stand for any single piece of syntax.
- Memory locations are denoted by l.
- We assume an implicit program/class table; we use the notation $C.m$ to get the method declaration for m within class C, similarly we use $C.f$ to get the declaration of field f, and $C.i$ to get the declaration of the i^{th} field.
- Memory, denoted by $\sigma : l \to C\{ls\}$, is a finite map from locations, l, to annotated tuples, $C\{ls\}$, representing objects; here C is the class name and ls are the field values. We use the notation C^{σ}_{l} to get the class name of l, $\sigma[l.f = l']$ to update a field of l, $\sigma[l.f]$ to access one, and $\sigma \setminus l$ to delete l (this is only used in our proofs since our small step reduction does not need to delete individual locations). The notation σ, σ' combines the two memories, and requires that $dom(\sigma)$ is disjoint from $dom(\sigma')$.

- We assume a typing judgment of form $\sigma; \Gamma \vdash e : T$, this says that the expression e has type T, where the classes of any locations are stored in σ and the types of variables are stored in the environment $\Gamma : x \rightarrow T$.

To encode object capabilities and I/O, we assume a special location c of class `Cap`. This location can be used in the main expression and would refer to an object with methods that behave non-deterministically, such methods would model operations such as file reading/writing. In order to simplify our proof, we assume that:

- `Cap` has no fields,
- instances of `Cap` cannot be created with a `new` expression,
- `Cap`'s `invariant()` method is defined to have a body of '`new True()`', and
- `mut` methods on `Cap`, unlike all other methods, can have the same method name declared multiple times. To enable a typesystem to be sound, we require that methods with the same name have identical signatures. Such methods will model I/O, for example reading a byte from a file could be modelled by having several different `read` method implementations, each of which returns a different byte value, a call to such a method will then non-deterministically reduce to one of these values.

We only model a single `Cap` capability class for simplicity, as modelling user-definable capability classes as described in 2 is unnecessary for the soundness of our invariant protocol.

We encode booleans as ordinary objects, in particular we assume:

- There is a `Bool` interface, a "boolean" value is any instance of this interface.
- There is a `True` class that implements `Bool`, an instance of this class represents "true".
- The `True` class has no fields, so it can be created with `new True()`.
- The `True` class has a trivial invariant (i.e. its body is `new True()`).
- Any other implementation of `Bool`, such as a `False` class, represent "false".

Other than the `invariant` method of `True`, we impose no requirements on the methods of the `Bool` interface or its classes, in particular, they could be used to provide logical operations.[16]

For simplicity, we do not formalise actual exception objects, rather we have expressions which are "*error*"s, these correspond to expressions which are currently 'throwing' an unchecked exception; in this way there is no value associated with an *error*. Our L42 implementation instead allows arbitrary `imm` values to be thrown as (unchecked) exceptions, formalising exceptions in such way would not cause any interesting variation of our proofs.

Grammar

The grammar is defined in Figure 1.

We use μ for our reference capabilities, and κ for field kinds. Recall that we don't model traditional `capsule` fields, but instead model our novel `rep` fields, which can only be initialised/updated with `capsule` values.

We use v, of form $\mu\, l$, to keep track of the reference capabilities in the runtime, as it allows multiple references to the same location to co-exist with different reference capabilities; however μ's are not stored in memory. The reduction rules do not change behaviour based on these μ's, they are merely used by our proofs to keep track of the guarantees enforced by the typesystem.

[16] In particular, `if` statements can be supported using Church encoding: we would have a `Bool.if` method of form `read method` T `if(`T `ifTrue,`T `ifFalse)`, for an appropriate type T. The body of `True.if` will then be `ifTrue`, and the body of `False.if` will be `ifFalse`. In this way, x`.if(`t`,`f`)` will return t if x is "true" and b if it is "false".

$$
\begin{array}{lcll}
e & ::= & x \mid \texttt{new}\ C\texttt{(}es\texttt{)} \mid \texttt{this.}f \mid \texttt{this.}f\texttt{=}e \mid e\texttt{.}m\texttt{(}es\texttt{)} & \text{expression} \\
 & \mid & e\ \texttt{as}\ \mu \mid \texttt{try}\ \texttt{\{}e\texttt{\}}\ \texttt{catch}\ \texttt{\{}e'\texttt{\}} & \\
 & \mid & v \mid v\texttt{.}f \mid v\texttt{.}f\texttt{=}e \mid \texttt{try}^{\sigma}\texttt{\{}e\texttt{\}}\ \texttt{catch}\ \texttt{\{}e'\texttt{\}} \mid \texttt{M(}l\texttt{;}\ e\texttt{;}\ e'\texttt{)} & \text{runtime expression} \\
v & ::= & \mu\, l & \text{value} \\
\mathcal{E}_v & ::= & \square \mid \texttt{new}\ C\texttt{(}vs, \mathcal{E}_v, es\texttt{)} \mid v\texttt{.}f\texttt{=}\mathcal{E}_v \mid \mathcal{E}_v\texttt{.}m\texttt{(}es\texttt{)} \mid v\texttt{.}m\texttt{(}vs, \mathcal{E}_v, es\texttt{)} & \text{evaluation context} \\
 & \mid & \mathcal{E}_v\ \texttt{as}\ \mu \mid \texttt{try}^{\sigma}\texttt{\{}\mathcal{E}_v\texttt{\}}\ \texttt{catch}\ \texttt{\{}e\texttt{\}} \mid \texttt{M(}l\texttt{;}\ \mathcal{E}_v\texttt{;}\ e\texttt{)} \mid \texttt{M(}l\texttt{;}\ v\texttt{;}\ \mathcal{E}_v\texttt{)} & \\
\mathcal{E} & ::= & \square \mid \texttt{new}\ C\texttt{(}es, \mathcal{E}, es'\texttt{)} \mid \mathcal{E}\texttt{.}f \mid \mathcal{E}\texttt{.}f\texttt{=}e \mid e\texttt{.}f\texttt{=}\mathcal{E} \mid \mathcal{E}\texttt{.}m\texttt{(}es\texttt{)} & \text{full context} \\
 & \mid & e\texttt{.}m\texttt{(}es, \mathcal{E}, es'\texttt{)} \mid \mathcal{E}\ \texttt{as}\ \mu \mid \texttt{try}\ \texttt{\{}\mathcal{E}\texttt{\}}\ \texttt{catch}\ \texttt{\{}e\texttt{\}} \mid \texttt{try}\ \texttt{\{}e\texttt{\}}\ \texttt{catch}\ \texttt{\{}\mathcal{E}\texttt{\}} & \\
 & \mid & \texttt{try}^{\sigma}\texttt{\{}\mathcal{E}\texttt{\}}\ \texttt{catch}\ \texttt{\{}e\texttt{\}} \mid \texttt{try}^{\sigma}\texttt{\{}e\texttt{\}}\ \texttt{catch}\ \texttt{\{}\mathcal{E}\texttt{\}} \mid \texttt{M(}l\texttt{;}\ \mathcal{E}\texttt{;}\ e\texttt{)} \mid \texttt{M(}l\texttt{;}\ e\texttt{;}\ \mathcal{E}\texttt{)} & \\
CD & ::= & \texttt{class}\ C\ \texttt{implements}\ Cs\ \texttt{\{}Fs\texttt{;}Ms\texttt{\}} \mid \texttt{interface}\ C\ \texttt{implements}\ Cs\ \texttt{\{}As\texttt{\}} & \text{class declaration} \\
F & ::= & \kappa\, C\, f & \text{field} \\
A & ::= & \mu\,\texttt{method}\, T\, m\texttt{(}T_1\, x_1, \ldots, T_n\, x_n\texttt{)} & \text{abstract method} \\
M & ::= & \mu\,\texttt{method}\, T\, m\texttt{(}T_1\, x_1, \ldots, T_n\, x_n\texttt{)}\ e & \text{method} \\
T & ::= & \mu\, C & \text{type} \\
\mu & ::= & \texttt{mut} \mid \texttt{imm} \mid \texttt{read} \mid \texttt{capsule} & \text{reference capability} \\
\kappa & ::= & \texttt{mut} \mid \texttt{imm} \mid \texttt{rep} & \text{field kind} \\
\mathcal{E}_r & ::= & \mathcal{E}_v[\texttt{new}\ C\texttt{(}vs, \square, vs'\texttt{)}] \mid \mathcal{E}_v[\square\texttt{.}f] \mid \mathcal{E}_v[\square\texttt{.}f\texttt{=}v] \mid \mathcal{E}_v[v\texttt{.}f\texttt{=}\square] & \text{redex context} \\
 & \mid & \mathcal{E}_v[\square\texttt{.}m\texttt{(}vs\texttt{)}] \mid \mathcal{E}_v[v\texttt{.}m\texttt{(}vs, \square, vs'\texttt{)}] \mid \mathcal{E}_v[\square\ \texttt{as}\ \mu] &
\end{array}
$$

Figure 1: Grammar

Our expressions (e), include variables (x), object creations (`new` C`(`es`)`), field accesses (`this.`f and v`.`f), field updates (`this.`f`=`e and v`.`f`=`e), method calls (e`.`m`(`es`)`), and values (v). Note that these are sufficient to model standard constructs, for example a sequencing "`;`" operator could be simulated by a method which simply returns its last argument. The expressions with `this` will only occur in method bodies, at runtime `this` will be substituted for a $\mu\, l$.

The three other expressions are:

- `as` expressions (e `as` μ), these evaluate e and change the reference capability of the result to μ. This is important for our proofs in Appendix A, were we require the typesystem to ensure certain properties for all references with a given μ. The typesystem is then responsible for rejecting any `as` expression that could violate this. For example, a `mut` l `as read` could be used to prevent l from being used for further mutation, and a `mut` l `as capsule` (if accepted by the typesystem) will guarantee that l is properly *encapsulated*. These `as` expressions are merely a proof device, they do not effect the runtime behaviour, and as in L42, they could simply be inferred by the typesystem when it would be sound to do so.

- Monitor expressions (`M(`l`;` e`;` e'`)`) represent our runtime injected invariant checks. The location l refers to the object whose invariant is being checked, e represents the behaviour of the expression, and e' is the invariant check, which will initially be `read` l`.invariant()`. The body of the monitor, e, is evaluated first, then the invariant check in e' is evaluated. If e' evaluates to an `imm True` (i.e. an `imm` reference to an instance of `True`), then the whole monitor expression will return the value of e, otherwise if it evaluates to a reference to a non-`True` value (i.e. an `imm` reference to an instance of a class other than `True`), the monitor expression is an *error*, and evaluation will proceed with the nearest enclosing `catch` block, if any.

- `try`–`catch` expressions (`try {`e`} catch {`e'`}`), which as in many other expression based languages[17], evaluate e, and if successful, return its result, otherwise if e is an *error*, evaluation will reduce to e'. During reduction, `try`–`catch` expressions will be annotated as `try`$^{\sigma}$`{`e`} catch {`e'`}`, where σ is the state of the memory before the body of the `try` block begins execution. This annotation has no effect on the

[17] This differs from *statement* based languages like Java, were a `try-catch`, does not return a value. The expression-based form can be translated to a call to a method whose body is "`try {return` e`;} catch (Throwable` t`) {return` e'`;}`".

runtime, but is used by the proofs to model strong exception safety: objects in σ are not mutated by the body of the `try`. Note that as mentioned before, this strong limitation is only needed for unchecked exceptions, in particular, invariant failures. Our calculus only models unchecked exceptions/errors, however L42 also supports checked exceptions, and `try-catch`es over them impose no limits on object mutation during the `try`.

We say that an e is an *error* if it represents an uncaught invariant failure, i.e. a runtime-injected invariant check that has failed and is not enclosed in a `try` block:

$error(\sigma, e)$ iff:

- $e = \mathcal{E}_v[\texttt{M(}l\texttt{;}\ v\texttt{;}\ \texttt{imm}\ l'\texttt{)}]$
- $\mathrm{C}^{\sigma}_{l'} \neq \texttt{True}$
- $\mathcal{E}_v$ is not of form $\mathcal{E}_v{}'[\texttt{try}^{\sigma'}\texttt{\{}\mathcal{E}_v{}''\texttt{\}}\ \texttt{catch}\ \texttt{\{\_\}}]$

This ensures that the body of a `try` block will only be an *error* if there is no inner `try–catch` that should catch it instead.

Locations (l), annotated tries ($\texttt{try}^{\sigma}\texttt{\{}e\texttt{\}}\ \texttt{catch}\ \texttt{\{}e'\texttt{\}}$), and monitors $\texttt{M(}l\texttt{;}\ e\texttt{;}\ e'\texttt{)}$ are runtime expressions: they are not written by the programmer, instead they are introduced internally by our reduction rules.

We provide several expression contexts, $\mathcal{E}$, $\mathcal{E}_v$, and $\mathcal{E}_r$. The standard evaluation context [34, Chapter 19], $\mathcal{E}_v$, represents the left-to-right evaluation order, an $\mathcal{E}_v$ is like an e, but with a *hole* ($\square$) in place of a sub-expression, but all the expression to the left of the hole must already be fully evaluated. This is used to model the standard left to right evaluation order: the hole denotes the location of the next sub-expression that will be evaluated. We use the notation $\mathcal{E}_v[e]$ to fill in the hole, i.e. $\mathcal{E}_v[e]$ returns $\mathcal{E}_v$ but with the single occurrence of $\square$ replaced by e. For example, if $\mathcal{E}_v = \square\texttt{.}m\texttt{()}$ then $\mathcal{E}_v[\texttt{new}\ C\texttt{()}] = \texttt{new}\ C\texttt{()}\texttt{.}m\texttt{()}$.

The full expression context, $\mathcal{E}$, is like an $\mathcal{E}_v$, but nothing needs to have been evaluated yet, i.e. the hole can occur in place of any sub-expression. The context $\mathcal{E}_r$ is also like an $\mathcal{E}_v$, but instead has a hole in an argument to a *redex* (i.e. an expression that is about to be reduced). This captures our previously informal notion: a value v is *involved in execution* if we have an $\mathcal{E}_r[v]$. For example, if $\mathcal{E}_r$ is of form $\mathcal{E}_v[\texttt{new}\ C\texttt{(}v_1, \square, v_3\texttt{)}]$, then $\mathcal{E}_r[v_2] = \mathcal{E}_v[\texttt{new}\ C\texttt{(}v_1, v_2, v_3\texttt{)}]$, i.e. we are about to perform an operation (creating a new object) that is involving the value v_2.

The rest of our grammar is standard and follows Java, except that types (T) contain a reference capability (μ), and fields (F) contain a field kind (κ).

Reference Capability Operations

We define the following properties of our reference capabilities and field kinds:

- $\mu \leq \mu'$ indicates that a reference of capability μ can be be used whenever one of kind μ' is expected. This defines a partial order:
  - $\mu \leq \mu$, for any μ
  - $\texttt{imm} \leq \texttt{read}$
  - $\texttt{mut} \leq \texttt{read}$
  - $\texttt{capsule} \leq \texttt{mut}$, $\texttt{capsule} \leq \texttt{imm}$, and $\texttt{capsule} \leq \texttt{read}$
- $\widetilde{\kappa}$ denotes the reference capability that a field with kind κ requires when initialised/updated:
  - $\widetilde{\texttt{rep}} = \texttt{capsule}$
  - $\widetilde{\kappa} = \kappa$, otherwise (in which case κ is also of form μ)
- $\mu{::}\kappa$ denotes the reference capability that is returned when accessing a field with kind κ, on a receiver with capability μ:
  - $\texttt{imm}{::}\kappa = \mu{::}\texttt{imm} = \texttt{imm}$
  - $\texttt{read}{::}\kappa = \texttt{read}$, if $\kappa \neq \texttt{imm}$
  - $\texttt{mut}{::}\texttt{rep} = \texttt{mut}{::}\texttt{mut} = \texttt{mut}$
  - $\texttt{capsule}{::}\texttt{rep} = \texttt{capsule}{::}\texttt{mut} = \texttt{capsule}$

The $\leq$ notation and $\widetilde{\kappa}$ notations are used later in Appendix A and Appendix B.

Well-Formedness Criteria

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

- `invariant()` methods must follow the requirements of Section 3, except that for simplicity method calls on `this` are not allowed.[18] This means that for every non-interface class C, $C.\texttt{invariant} = \texttt{read method imm Bool invariant()}\ e$, where e can only use `this` as the receiver of an `imm` or `rep` field access. Formally, this means that forall $\mathcal{E}$ where $e = \mathcal{E}[\texttt{this}]$, we have:
  - $\mathcal{E} = \mathcal{E}'[\square.f]$, for some $\mathcal{E}'$
  - $C.f = \kappa\ \_\ f$
  - $\kappa \in \{\texttt{imm}, \texttt{rep}\}$

- Rep mutators must also follow the requirements in 3, except that a `mut` method that reads a `rep` field is *always* considered a rep mutator, even if it only needs to use the field value as `read`[19] Such methods must not use `this`, except for the single access to the `rep` field, and they must not have `mut` or `read` parameters, or a `mut` return type. Formally, this means that for any C, m, and f, if $C.f = \texttt{rep}\ \_\ f$ and $C.m = \texttt{mut method}\ \mu'\ \_\ m\texttt{(}\mu_1\ \_\ \_, ..., \mu_n\ \_\ \_\texttt{)}\ \mathcal{E}[\texttt{this.}f]$:
  - $\texttt{this} \notin \mathcal{E}$
  - $\mu_1 \notin \{\texttt{mut}, \texttt{read}\}$, ..., $\mu_n \notin \{\texttt{mut}, \texttt{read}\}$
  - $\mu' \neq \texttt{mut}$

- We require that the method bodies do not contain runtime expressions. Formally, for all C_0 and m with $C_0.m = \_\texttt{method}\ \_\ m\texttt{(}\_\ \_, ..., \_\ \_\texttt{)}\ e$, e contains no l, $\texttt{M(\_; \_; \_)}$, or $\texttt{try}^{\sigma'}\texttt{\{\_\} catch \{\_\}}$ expressions.

- We also assume some general sanity requirements: every C mentioned in the program or in any well typed expression has a single corresponding `class`/`interface` definition; the Cs in an `implements` are all names of `interface`s; the C in a `new` C`(`es`)` expression denotes a `class`; the `implements` relationship is acyclic; the fields of a `class` have unique names; methods within a `class`/`interface` (other than `mut` methods in `Cap`) have unique names; and parameters of a method have unique names and are not named `this`.

- For simplicity of the type-system and associated proof, we require that every method in the (indirect) super-interfaces of a class be implemented with exactly the same signature, i.e. if we have a $\texttt{class}\ C\ \texttt{implements}\ \_\ \texttt{\{\_;}Ms\texttt{\}}$, and $\texttt{interface}\ C'\ \texttt{implements}\ \_\ \texttt{\{}As\texttt{\}}$, where C' is reachable through the `implements` clauses starting from C, then for all $\mu\,\texttt{method}\,T\,m\texttt{(}Ps\texttt{)} \in As$, there is some e with $\mu\,\texttt{method}\,T\,m\texttt{(}Ps\texttt{)}\ e \in Ms$.

A typesystem, such as our example one in Appendix B, may impose additional restrictions on method bodies, for example that they are well typed. Our typesystem requirements in Appendix A however only refer to the main expression, and hence only the methods that could actually be called need to be restricted.

Reduction Rules

Our reduction rules are defined in Figure 2. We use the function $\mathit{fresh}(\sigma)$ to return an arbitrary l such that $l \notin \mathit{dom}(\sigma)$. The rules use $\mathcal{E}_v$ to ensure that the sub-expression to be reduced is the left-most unevaluated one:

- NEW/NEW TRUE creates a new object. NEW is used when creating a non-`True` object, it returns a monitor expression that will check the new object's invariant, and if that succeeds, return a `mut` reference to the object. NEW TRUE is for creating an instance of `True`, it simply returns a `mut` reference to the new object, *without* checking its invariant. The separate NEW TRUE rule is needed as the invariant of `True` is itself defined to perform `new True()`, so using the NEW rule would cause an infinite recursion. This is sound since *manually* calling invariant on `True` will return a `True` reference. Note that although we do not define what *fresh* actually returns, since it is a *function* these reduction rules are deterministic: l_0 is uniquely defined for any given σ.

[18] Such method calls could be inlined or rewritten to take the field values themselves as parameters.

[19] This restriction is merely for simplicity, it does not limit expressivity as one can write a getter of form $\texttt{read method read}\ C\ m\texttt{(this.}f\texttt{)}$, where $C.f = \texttt{rep}\ C\ f$, and then call this m on a `mut this`.

(NEW) $\sigma|\mathcal{E}_v[\texttt{new}\ C\texttt{(}\_l_1, ..., \_l_n\texttt{)}] \rightarrow \sigma, l_0 \mapsto C\{l_1, ..., l_n\}|\mathcal{E}_v[\texttt{M(}l_0\texttt{;}\ \texttt{mut}\ l_0\texttt{;}\ \texttt{read}\ l_0\texttt{.invariant())}]$, where:
$l_0 = \mathit{fresh}(\sigma)$ and $C \neq \texttt{True}$

(NEW TRUE) $\sigma|\mathcal{E}_v[\texttt{new True()}] \rightarrow \sigma, l_0 \mapsto \texttt{True}\{\}|\mathcal{E}_v[\texttt{mut}\ l_0]$, where:
$l_0 = \mathit{fresh}(\sigma)$

(ACCESS) $\sigma|\mathcal{E}_v[\mu\, l\texttt{.}f] \rightarrow \sigma|\mathcal{E}_v[\mu'\, l']$, where:
$\mathrm{C}^\sigma_l.f = \kappa\ \_\ f$, $\mu' = \mu{::}\kappa$, and $l' = \sigma[l.f]$

(UPDATE) $\sigma|\mathcal{E}_v[\_\, l\texttt{.}f\ \texttt{=}\ \_\, l'] \rightarrow \sigma[l.f = l']|\mathcal{E}_v[\texttt{M(}l\texttt{;}\ \texttt{mut}\ l\texttt{;}\ \texttt{read}\ l\texttt{.invariant())}]$

(CALL) $\sigma|\mathcal{E}_v[\_\, l_0\texttt{.}m\texttt{(}\_l_1, ..., \_l_n\texttt{)}] \rightarrow \sigma|\mathcal{E}_v[e[\texttt{this} \coloneqq \mu_0\, l_0, x_1 \coloneqq \mu_1\, l_1, ..., x_n \coloneqq \mu_n\, l_n]\ \texttt{as}\ \mu']$, where:
$\mathrm{C}^\sigma_{l_0}.m = \mu_0\ \texttt{method}\ \mu'\ \_\ m\texttt{(}\mu_1\ \_\ x_1, ..., \mu_n\ \_\ x_n\texttt{)}\ e$
if $\mu_0 = \texttt{mut}$, $\nexists f, \mathcal{E}$ with $\mathrm{C}^\sigma_{l_0}.f = \texttt{rep}\ \_\ f$ and $e = \mathcal{E}[\texttt{this.}f]$

(CALL MUTATOR) $\sigma|\mathcal{E}_v[\_\, l_0\texttt{.}m\texttt{(}\_l_1, ..., \_l_n\texttt{)}] \rightarrow \sigma|\mathcal{E}_v[\texttt{M(}l_0\texttt{;}\ e\texttt{;}\ \texttt{read}\ l_0\texttt{.invariant())}]$, where:

$\mathrm{C}^\sigma_{l_0}.m = \texttt{mut method}\ \mu'\ \_\ m\texttt{(}\mu_1\ \_\ x_1, ..., \mu_n\ \_\ x_n\texttt{)}\ \mathcal{E}[\texttt{this.}f]$
$\mathrm{C}^\sigma_{l_0}.f = \texttt{rep}\ \_\ f$
$e = \mathcal{E}[\texttt{this.}f][\texttt{this} \coloneqq \texttt{mut}\ l_0, x_1 \coloneqq \mu_1\, l_1, ..., x_n \coloneqq \mu_n\, l_n]\ \texttt{as}\ \mu'$

(AS) $\sigma|\mathcal{E}_v[\_\, l\ \texttt{as}\ \mu] \rightarrow \sigma|\mathcal{E}_v[\mu\, l]$

(TRY ENTER) $\sigma|\mathcal{E}_v[\texttt{try \{}e\texttt{\} catch \{}e'\texttt{\}}] \rightarrow \sigma|\mathcal{E}_v[\texttt{try}^\sigma\texttt{\{}e\texttt{\} catch \{}e'\texttt{\}}]$

(TRY OK) $\sigma|\mathcal{E}_v[\texttt{try}^{\sigma'}\texttt{\{}v\texttt{\} catch \{\_\}}] \rightarrow \sigma|\mathcal{E}_v[v]$

(TRY ERROR) $\sigma|\mathcal{E}_v[\texttt{try}^{\sigma'}\texttt{\{}e\texttt{\} catch \{}e'\texttt{\}}] \rightarrow \sigma|\mathcal{E}_v[e']$, where $\mathit{error}(\sigma, e)$

(MONITOR EXIT) $\sigma|\mathcal{E}_v[\texttt{M(}l\texttt{;}\ v\texttt{;}\ \texttt{imm}\ l'\texttt{)}] \rightarrow \sigma|\mathcal{E}_v[v]$, where $\mathrm{C}^\sigma_{l'} = \texttt{True}$

Figure 2: Reduction rules

- ACCESS looks up the value of a field in the memory and returns it, annotated with the appropriate reference capability (see above for the definition of $\mu{::}\kappa$).

- UPDATE updates the value of a field, returning a monitor that re-checks the invariant of the receiver, and if successful, will return the receiver of the update as `mut`. Note that this does *not* check that the receiver of a the field update has an appropriate reference capability, it is the responsibility of the type-system to ensure that this rule is only applied to a `mut` or `capsule` receiver. For soundness, we return a `mut` reference even when the receiver is `capsule`, promotion can then be used to convert the result to a `capsule`, provided the new field value is appropriately encapsulated.

- CALL/CALL MUTATOR looks for a corresponding method definition in the receiver's class, and reduces to its body with parameters appropriately substituted. The parameters are substituted with the reference capabilities of the method's signature, not the capabilities at the call-site, this is used by the proofs to show that further reductions will respect the capabilities in the method signature. We wrap the body of the method call in a `as` expression to ensure that the returned μ is actually as the method signature specified; for example, a method declared as returning a `read` might actually return a `mut`, but the `as` expressions will soundly change it to a `read`, thus preventing it from being used for mutation. As with `as` expressions in general, the type system is required to ensure that this will not break our reference capability guarantees in Appendix A. The CALL MUTATOR rule is like CALL, but is used when the method is a rep mutator (a `mut` method that accesses a `rep` field): it additionally wraps the method body in a monitor expression that will re-check the invariant of the receiver once the body of the method has finished reducing. Note that as `Cap` has no `rep` fields and can have multiple definitions of the same method, the CALL rule allows for non-determinism, but only if the receiver is of class `Cap` and the method is a `mut` method.

- AS simply changes the reference capability to the one indicated. Note that our requirements on the type-system, given in Appendix A, ensure that inappropriate promotions (e.g. `imm` to `mut`) will be ill-typed.

- TRY ENTER will annotate a `try–catch` with the current memory state, before any reduction occurs within the `try` part. In Appendix A, we require the type system to ensure strong exception safety: that the objects in the saved σ are never modified. Note that the grammar for $\mathcal{E}_v$ prevents the body of an *unannotated* `try` block from being reduced, thus ensuring that this rule is applied first.

- TRY OK simply returns the body of a `try` block once it has successfully reduced to a value. TRY ERROR on the other hand reduces to the body of the `catch` block if its `try` block is an *error* (an invariant failure that is *not* enclosed by an inner `try` block). Note that the grammar for $\mathcal{E}_v$ prevents the body of a `catch` block from being reduced, instead TRY ERROR must be applied first; this ensures that the body of a `catch` is only reduced if the `try` part has reduced to an *error*.

- MONITOR EXIT reduces a successful invariant check to the body of the monitor. If the invariant check on the other hand has failed, i.e. has returned a non-`True` reference, it will be an *error*, and TRY ERROR will proceed to the nearest enclosing `catch` block.

Note that as with most OO languages, an expression e can always be reduced, unless: e is already a value, e contains an uncaught invariant failure, or e attempts to perform an ill-defined operation (e.g. calling a method that doesn't exist). The latter case can be prevented by any standard sound OO typesystem. However, invalid use of reference capabilities (e.g. having both an `imm` and `mut` reference to the same location) does *not* cause reduction to get stuck, instead, in Appendix A we explicitly require that the typesystem prevents such things from happening, which our example type system in **??** proves to be the case.

Note that the monitor expressions are only a proof device, they need not be implemented directly as presented. For example, in L42 they are implemented by statically injecting calls to `invariant()` at the end of setters (for `imm` and `rep` fields), factory methods, and capsule mutators; this works as L42 follows the uniform access principle, so it does not have primitive expression forms for field updates and constructors, rather they are uniformly represented as method calls.

Statement of Soundness

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

$\sigma|e \Rightarrow \sigma'|e'$ iff $\sigma|e \rightarrow \sigma'|e'$, and $\forall \sigma'', e''$, $\sigma|e \rightarrow \sigma''|e''$, implies $\sigma''|e'' = \sigma'|e'$

Set style definition:

$\sigma|e \Rightarrow \sigma'|e'$ iff $\{\sigma'|e'\} = \{\sigma''|e''$ where $\sigma|e \rightarrow \sigma''|e''\}$

We say that an object is *valid* when calling its `invariant()` method would deterministically produce an `imm True` in a finite number of steps, i.e. assuming the typesystem is sound, this means it does not evaluate to a non-`True` reference, fail to terminate, or produce an *error*. We also require that evaluating `invariant()` preserves existing memory, however new objects can be freely created and mutated:

$\mathit{valid}(\sigma, l)$ iff $\sigma|\texttt{read}\, l\texttt{.invariant()} \Rightarrow^{+} \sigma, \sigma'|\texttt{imm}\, l$ where $\mathrm{C}_l^{\sigma,\sigma'} = \texttt{True}$.

To allow the `invariant()` method to be called on an invalid object, and access fields on such an objects, we define the set of trusted execution steps as the call to `invariant()` itself, and any field accesses inside its evaluation:

$\mathit{trusted}(\mathcal{E}_r, l)$ iff, either:

- $\mathcal{E}_r = \mathcal{E}_v[\texttt{M}(l;\, \_;\, \Box\texttt{.invariant()})]$, or
- $\mathcal{E}_r = \mathcal{E}_v[\texttt{M}(l;\, \_;\, \mathcal{E}_v{}'[\Box\texttt{.}f])]$.

The idea being that the $\mathcal{E}_r$ is like an $\mathcal{E}_v$ but it has a hole where a reference can be, thus $\mathit{trusted}(\mathcal{E}_r, l)$ holds when the very next reduction we are about to perform is $\mu\, l\texttt{.invariant()}$ or $\mu\, l\texttt{.}f$. As we discuss in our proof of Soundness, any such $\mu\, l\texttt{.}f$ expression came from the body of the `invariant()` method itself, since l can not occur in the *rog* of any of its fields mentioned in the `invariant()` method.[20]

We define a *validState* as one that was obtained by any number of reductions from a well typed initial main expression and memory:

$\mathit{validState}(\sigma, e)$ iff $c \mapsto \texttt{Cap}\{\}|e_0 \rightarrow^* \sigma|e$, for some e_0 such that:

- $c \mapsto \texttt{Cap}\{\}; \emptyset \vdash e_0 : T$, for some T
- e_0 contains no $\texttt{M}(\_;\, \_;\, \_)$, $\texttt{try}^{\sigma'}\texttt{\{\_\} catch \{\_\}}$, or $\_\, \texttt{as}\, \mu$ expressions
- $\forall \mu\, l \in e_0$, $\mu\, l = \texttt{mut}\, c$

By restricting which initial expressions are well-typed, the type-system (such as the one presented in Appendix B) can ensure the required properties of our reference-capabilities (see Appendix A); any standard OO type system can also be used to reject expressions that might try to perform an ill-defined reduction (like reading a field that does not exist). The initial expression cannot contain any runtime expressions, except for `mut` references to the single pre-existing `Cap` object. Note that as `Cap` has no fields and `this` is not of form l, field accesses/updates in the initial main expression can never be reduced. To make the type system and proofs presented in Appendix B simpler, we require that c can only be initially referenced as `mut` and that there are no `as` expressions in e_0. This restriction does not effect expressivity, as you can pass c to a method whose parameters have the desired reference capability, and whose body contains the desired `as` expressions.

Finally, we define what it means to soundly enforce our invariant protocol:

Theorem 1 (Soundness)**.**

If $\mathit{validState}(\sigma, \mathcal{E}_r[\_l])$, then either $\mathit{valid}(\sigma, l)$ or $\mathit{trusted}(\mathcal{E}_r, l)$.

Except for the injected invariant checks (and fields they directly access), any redex in the execution of a well typed program takes as input only valid objects. In particular, no method call (other than *injected* invariant checks themselves) can see an object which is being checked for validity.

This is a very strong statement because $\mathit{valid}(\sigma, l)$ requires the invariant of l to deterministically terminate. Our setting does ensure termination of the invariant of any l that is now within a redex (as opposed to an l that is on the heap, or is being monitored). This works because non terminating `invariant()` methods would cause the monitor expression to never terminate. Thus, an l with a non terminating `invariant()` is never involved in an untrusted redex. This works as invariants are deterministic computations that depend only on the state reachable from l. In particular, if l is in a redex, a monitor expression must have terminated after the object instantiation and after any updates to the state of l.

[20] Invariants only see `imm` and `rep` fields (as `read`), neither of which can alias the current object.

6. Case Studies

To perform compelling case studies, we used our system on many examples, including one designed to be a worst case scenario for our approach. We also replicate many examples originally proposed by other papers, so that not all the code examples come from us.

6.1. An interactive GUI

We start by presenting our GUI example; a program that interacts with the real world using I/O. It demonstrates how to verify invariants over cyclic mutable object graphs. Our example is particularly relevant since, as with most GUI frameworks, it uses the *composite* programming pattern; arguably one of the most fundamental patterns in OO.

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 graphically 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` are stored in 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[21], 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; Int width = 300; Int height = 300;
  @Override read method Int left() { return this.box.l; }
  @Override read method Int top() { return this.box.t; }
  @Override read method Int width() { return this.width; }
  @Override read method Int height() { return this.height; }
  @Override read method read Widgets children() { return this.box.c; }
  @Override mut method Void dispatch(Event e) {
    for (Widget w:this.box.c) { w.dispatch(e); }
  }
  read method Bool invariant() {../* presented later */..}
  SafeMovable(capsule Widgets c) { this.box = makeBox(c); }
  static method capsule Box makeBox(capsule Widgets c) {
    mut Box b = new Box(5, 5, c);
    b.c.add(new Button(0, 0, 10, 10, new MoveAction(b));
    return b;// mut b is soundly promoted to capsule
  }
}
class Box { Int l; Int t; mut Widgets c; Box(Int l, Int t, mut Widgets c) {..} }
class MoveAction implements Action {
  mut Box outer;
  MoveAction(mut Box outer) { this.outer = outer; }
  mut method Void process(Event e) { this.outer.l += 1; }
}
... //main expression
//#$ is a capability operation making a Gui object
Gui.#$().display(new SafeMovable(...));
```

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 cyclic: 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(e)`. Once a `Button` receives an `Event` with a matching ID, it will call its `Action`'s `process(e)` method.

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

Our example shows how 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 events. Importantly, neither our application, nor the underlying GUI library requires back doors, into either RCs or OCs.

The Invariant

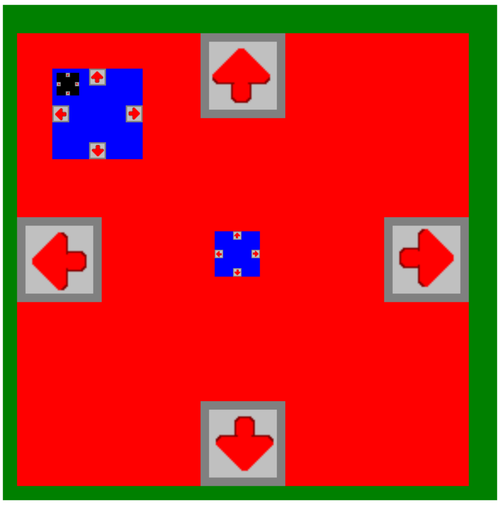
SafeMovable is the only class in our GUI that has an invariant, our system automatically checks it in two places: the end of its constructor and the end of its `dispatch(e)` method (which is a capsule mutator). There are no other checks inserted since we never do a direct field update on a **SafeMovable**. The code for the invariant is just a couple of simple nested loops:[22]

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

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(w)` only uses **this** to access its **imm** and **capsule** fields.

Our Experiment

As shown in the figure below, 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 our invariant protocol, 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 [36, 37], where the invariant of the receiver is instead checked at the start and end of *every* public (in D) and qualified[23] (in Eiffel) method call. In our **SafeMovable** class, all methods are public, and all calls (outside the invariant) are qualified, thus this difference is irrelevant. Neither protocol performs invariant checks on field accesses or updates, however due to the 'uniform access principle' [37], 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 perform 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;[24]in comparison our

---

[22] We could make the code sigtly more efficient by avoiding comparing twice each pair of widgets. Code efficiency is not the priority here.

[23] That is, the receiver is not **this**.

[24] This difference is caused by Eiffel treating getters specially, and skipping invariant checks when calling a getter. Thus, even ignoring getter methods, the visible state semantic would still run 14 millions of invariant checks.

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*. Concluding, we have shown that when an invariant check queries other objects with invariants the visible state semantics may cause an exponential explosion in the number of checks.

Spec# Comparison

We also encoded our example in Spec#[25]; that relies on pack/unpack; also called inhale/exhale or the Boogie methodology. In pack/unpack, an object's invariant is checked only by the explicit pack operations. In order for this to be sound, some form of aliasing and/or mutation control is necessary. Spec# uses a theorem prover, together with source code annotations. Spec# can be used for full static verification, but 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#.

As the back-end of the L42 GUI library is written in Java, we did not port it to Spec#, rather we just simulated it, and don't actually display a GUI in Spec#. We ran our code through the Spec# verifier (powered by Boogie [38]), which only gave us 2 warnings[26], because the invariant of `SafeMovable` was not known to hold at the end of its constructor and `dispatch(e)` method. Thus, like our system, Spec# checks the invariant at those two points at runtime. Thus the code is equivalently verified in both Spec# and L42; in particular it performed exactly the same number (77) of runtime invariant checks.

While the same numbers of checks are performed, we do not have the same guarantee provided by our approach: Spec#/Boogie does not soundly handle the non-deterministic impact of I/O, thus it does not properly prevent us from writing unsound invariants that may be non-deterministic. We also encoded our GUI in Microsoft Code Contracts [39], whose unsound heuristic also calls the invariant 77 times. However Code Contract does not enforce the encapsulation of `children()`, thus this approach is even less sound than Spec#.

Note how both our L42 and Spec# code required us to use the box pattern for our `SafeMovable`, due to the cyclic object graph caused by the `Action`s of `Button`s needing to change their enclosing `SafeMovable`'s position. 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. The following table shows 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. We only count constructs Spec# adds over C# as annotations, we also do not count annotations related to array bounds or null checks:

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

To encode the GUI example in L42, the only annotations we needed were the 3 reference capabilities: `mut`, `read`, and `capsule`. Our Spec# code requires 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#. The table presents token and character counts to compare against Spec#'s annotations, which can be quite long and involved, whereas ours are just single keywords. Consider for example the Spec# pre-condition on `SafeMovable`'s constructor:

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

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

[26] 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.

The Spec# code also required us to deviate from the code style shown 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 indirection we had to do 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 in comparision with Spec#, that is more verbose but supports a wider range of verification applications.

6.2. *A Comparison of a Simple Example in Spec#*

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`). This example is interesting since it relies on `List`s and `Point`s that are not designed with `Hamster/Cage`s in mind.

```
class Point { Double x; Double y; Point(Double x, Double y) {..}
  @Override read method Bool equals(read Object that) {
    if (!(that instanceof Point)) { return false; }
    Point p = (Point)that;
    return this.x == p.x && this.y == p.y;
  }
}
class Hamster { Point pos; Hamster(Point pos) {..} }//pos is imm by default
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.pos());
    this.moveTo(this.path.get(index % this.path.size())); }
  read method Point pos() { return this.h.pos; }
  mut method Void moveTo(Point p) { this.h.pos = p; }
}
```

The `invariant()` method on `Cage` simply verifies that the `pos` of `this.h` is within the `this.path` list. This is accepted by our invariant protocol since `path` is an `imm` field (hence deeply immutable) and `h` is a `capsule` field (hence fully encapsulated). The `path.contains` call is accepted by our type system as it only needs `read` access: it merely needs to be able to access each element of the list and call `Point`'s equal method, which takes a `read` receiver and parameter. The `move` method actually moves the hamster along the path, but to ensure that our restrictions on `capsule` fields are respected we forwarded some of the behaviour to separate methods: `pos()` which returns the position of `h` and `moveTo(p)` which updates the position of `h`. The `pos` method is needed since `move()` is a `mut` method, and so any direct `this.h` access would cause it to be a capsule mutator, which would make the program erroneous as `move()` uses `this` multiple times. Similarly, we need the `moveTo(p)` method to modify the ROG of the `h` field, this must be done within a capsule mutator that uses `this` only once.

As our `path` and `h` fields are never themselves updated, the only point where the ROG of our `Cage` can mutate is in the `moveTo(p)` capsule mutator, thus our invariant protocol will insert runtime invariant checks only here and at the end of the constructor.

Note: since only `Cage` has an invariant, only the code of `Cage` needs to be handled carefully; allowing the code for `Point` and `Hamster` to be unremarkable. Thus our verification approach is more self contained and modular. This contrasts with Spec#: all code involved in verification needs to be designed with verification in mind [40].

Comparison with Spec#

We now show our hamster example in the system most similar to ours, Spec#:

```
// Note: assume everything is 'public'
class Point { double x; double y; Point(double x, double y) {..}
  [Pure] bool Equal(double x, double y) { return x == this.x && y == this.y; } }
class Hamster{[Peer] Point pos; Hamster([Captured] Point pos){..} }
class Cage {
  [Rep] 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; base(); }
  invariant exists {int i in (0 : this.path.Count);
    this.path[i].Equal(this.h.pos.x, this.h.pos.y) };
  void Move() {
    int i = 0;
    while(i<path.Count && !path[i].Equal(h.pos.x,h.pos.y)){ i++; }
    expose(this) { this.h.pos = this.path[i%this.path.Count]; }
  }
}
```

In both this and our original version, we designed `Point` and `Hamster` in a general way, and not solely to be used by classes with an invariant: thus `Point` is not an immutable class.

The Spec# approach uses ownership: the `Rep` attribute on the `h` and `path` fields means its value is owned by the enclosing `Cage`, similarly the `ElementsRep` attribute on the `path` field means its *elements* are owned by the `Cage`. Conversely, in the `Hamster` class, the `Peer` annotation on the `pos` field means its value is owned by the owner of the enclosing `Hamster`, thus if a `Cage` owns a `Hamster`, it also owns the `Hamster`'s `pos`. The `Captured` annotations on the constructor parameters of `Cage` and `Hamster` means that the passed in values must be un-owned and the body of the constructor may modify their owners (the owner is automatically updated when the parameter is assigned to a `Rep` or `Peer` field).

Though we don't want either `pos` or `path` to ever mutate, Spec# currently has no way of enforcing that an *instance* of a non-immutable class is itself immutable.[27] In Spec#, an `invariant()` can only access fields on owned or immutable objects, thus necessitating our use of the `Peer` and `Rep` annotations on the `pos` and `path` fields.

Note that this prevents multiple `Cage`s from sharing the same point instance in their `path`. Had we made `Point` an immutable class, we would get no such restriction. A similar problem applies to our `pos` field: the `pos` of `Hamster`s in different `Cage`s cannot be the same `Point` instance. Note how if we consider being in the ROG of an object's capsule fields as being 'owned' by the object, our `capsule` fields behave like `Rep` fields; similarly, `mut` fields (that are in the ROG of a `capsule` field) behave like `Peer` fields.

The `expose(this)` block is needed, since in Spec# in order to modify a field of an object (like `this.h.pos`), we must first "expose" its owner (the `Cage`). During an `expose` block, Spec# will not assume the invariant of the exposed object, but will ensure it is re-established at the end of the block. This is similar to our concept of capsule mutators (like our `moveTo` method above), however it is supported by adding an extra syntactic construct (the `expose` block), which we avoid.

Finally, note the custom `Equal(x,y)` method on `Point`: this is needed since we can't overload the usual `Object.Equals(other)` method because it is marked as `Reads(ReadsAttribute.Reads.Nothing)`, which requires the method not read any fields, even those of its receiver. We resorted to making our own `Equal(x,y)` method. Since it is called in `Cage`'s invariant, Spec# requires it to be annotated as `Pure`, this requires that it can only read fields of objects owned by the *receiver* of the method, so a method `[Pure] bool Equal(Point that)` 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 just pass the fields of the other point to the method. Sadly this mean we can no longer use `List`'s `Contains(elem)` and `IndexOf(elem)` methods, rather we have to expand out their code manually.

[27] There is a paper [41] that describes a simple solution to this problem: assign ownership of the object to a special predefined 'freezer' object, which never gives up mutation permission. However, this does not appear to have been implemented. This would provide similar flexibility to the RC system we use, which allows an initially mutable object to be promoted to immutable.

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

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

In Spec# objects start their life as un-owned, so each `new` instruction above returns an unowned object. However when the `Point`s are placed inside the `pl` list, Spec# loses track of this. Thus the `AssignSame` call is needed to mark the elements of `pl` as still being unowned (since `pl` itself is unowned). Contrast this with our system which requires no such operation; we can simply write:

```
Cage c=new Cage(new Hamster(new Point(0,0)),List.of(new Point(0,0),new Point(0,1)));
```

In Spec#, we had to add 10 different annotations, of 8 different kinds, some of which are quite involved. In comparison, our approach requires only 8 simple keywords of 3 different kinds. However, we needed to write separate `pos()` and `moveTo(p)` methods.

6.3. A Worst Case for the Number of Invariant Checks

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);}
  mut method Void processDay(Int dayOfYear) { this.box.processDay(dayOfYear); }
  mut method Void addChild(capsule Person child) { this.box.children.add(child); }
  read method Bool invariant() {
    for (Person p : this.box.parents) {
      for (Person c : this.box.children) {
        if (p.daysLived <= c.daysLived) { return false; }
      }
    }
    return true;
  }
}
```

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(dayOfYear), after all the parents and children have been updated. Thus processDay(dayOfYear) 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 semantic problems if, for example, a child was updated before their parent.

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

```
// 2 parents (one 32, the other 34), and no children
var fam = new Family(List.of(new Person("Bob", 11720, 40),
    new Person("Alice", 12497, 87)), List.of());

for (Int day = 0; day < 365; day++) { fam.processDay(day); } // Run for 1 year
for (Int day = 0; day < 365; day++) { // The next year
  fam.processDay(day);
  if (day == 45) { fam.addChild(new Person("Tim", 0, day)); }
}
for (Int day = 0; day < 365; day++) { // The 3rd year
  fam.processDay(day);
  if (day == 340) { fam.addChild(new Person("Diana", 0, day)); }
}
```

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 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.

Our protocol performs a single invariant check at the end of each constructor, processDay(dayOfYear) and addChild(child) 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(dayOfYear) and addChild(child).

The results for Spec# are very interesting, since it performs fewer checks than L42. This is the case since processDay(dayOfYear) 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, which will perform a runtime invariant check.

Spec# relies on the absence of arithmetic overflow, and performs runtime checks to ensure this[28], as such the verifier concludes that the field increment in processDay(dayOfYear) 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[29] 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. In order to implement the addChild(child) 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(dayOfYear) method in an **expose** (**this**) block. In total we needed 16 annotations, worth a total of 45 tokens, this is slightly worse than the code following our approach that we showed above, which has 14 annotations and 14 tokens.

[28] Runtime checks are enabled by a compilation option; when they fail, unchecked exceptions are thrown.
[29] The Spec# code is in the artifact.

6.4. Encoding Examples from Spec# Papers

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

- *Verification of Object-Oriented Programs with Invariants* [3]: this paper introduces their methodology. In their examples section (pages 41–47), they show how their methodology would work in a class hierarchy 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 file names, and a **`DirFileList`** class that extends it with a stronger invariant. All of these examples can be represented in L42[30]. 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 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 include the methods from **`FileList`**, and they will automatically enforce the invariant of **`DirFileList`**.
- *Object Invariants in Dynamic Contexts* [42]: this paper shows how one can specify an invariant for a doubly linked list of **`int`**s (here **`int`** 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[31] (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* [27]: this paper shows how one can verify invariants over interacting objects, where neither owns/contains the others. 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

[30] Our encodings are in the artifact.

[31] 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.

`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.

7. Patterns

In this section we show programming patterns that allow various kinds of invariants. Our goal is not to verify existing code or patterns, but to create a simple system that allows soundly verifying the correctness of data structures. In particular, as we show, in order to use our approach to ensure invariants, one has to program in an uncommon and very defensive style.

The SubInvariant Pattern

We showed how the box pattern can be used to write invariants over cyclic mutable object graphs, the latter also shows how a complex mutation can be done in an 'atomic' way, with a single invariant check. However the box pattern is much more powerful.

Suppose we want to pass a temporarily 'broken' object to other code as well as perform multiple field updates with a single invariant check. Instead of adding new features to the language, like an `invalid` modifier (denoting an object whose invariant does not need to hold), and an `expose` statement like Spec#, we can use a 'box' class and a capsule mutator to the same effect:

```
interface Person{ mut method Bool accept(read Account a,read Transaction t); }
interface Transaction{ mut method ImmList<Transfer> compute(); }
//Here ImmList<T> represents a list of immutable Ts.
class Transfer{ Int money;
  method Void execute(mut AccountBox that){// Gain some money, or lose some money
    if(this.money>0){ that.income+=money; }
    else{ that.expenses -= money; }
  }
}
class AccountBox{
  UInt income=0; UInt expenses=0;
  read method Bool subInvariant(){ return this.income >= this.expenses; }
  //An 'AccountBox' is like a 'potentially invalid Account':
  //we may observe income >= expenses
}
class Account{
  capsule AccountBox box; mut Person holder;
  read method Bool invariant(){ return this.box.subInvariant(); }
  // 'h' could be aliased elsewhere in the program
  Account(mut Person h){ this.holder=h; this.box=new AccountBox(); }
  mut method Void transfer(mut Transaction ts){
    if(this.holder.accept(this, ts)){ this.transferInner(ts.compute()); }
  }
  // capsule mutator, like an 'expose(this)' statement
  private mut method Void transferInner(ImmList<Transfer> ts){
     mut AccountBox b = this.box;
     for (Transfer t : ts) { t.execute(b); }
  }// check the invariant here
}
```

The idea here is that `transfer(ts)` will first check to see if the account holder wishes to accept the transaction, it will then compute the full transaction (which could cache the result and/or do some I/O), and then execute each transfer in the transaction. We specifically want to allow an individual `Transfer` to raise the `expenses` field by more than the `income`, however we don't want an entire `Transaction` to do this. Our capsule mutator (`transferInner`) allows this by behaving like a Spec# `expose` block: during its body (the `for` loop) we don't know or care if `this.invariant()` is `true`, but at the end it will be checked. For this to make sense, we make

`Transfer.execute` take an `AccountBox` instead of an `Account`: it cannot assume that the invariant of `Account` holds, and it is allowed to modify the fields of `that` without needing to check it. Though capsule mutators can be used to perform batch operations like the above, they can only take immutable and capsule objects. This means that they can perform no non-deterministic I/O (due to our OC system), and other externally accessible objects (such as a `mut Transaction`) cannot be mutated during such a batch operation.

As you can see, adding support for features like `invalid` and `expose` is unnecessary, and would likely require making the type system significantly more complicated as well as burdening the language with more core syntactic forms.

In particular, the above code demonstrates that our system can:

- Have useful objects that are not entirely encapsulated: the `Person holder` is a `mut` field; this is fine since it is not mentioned in the `invariant()` method.
- Wrap normal methods over capsule mutators: `transfer` is not a capsule mutator, so it can use `this` multiple times and take a `mut` parameter.
- Perform multiple state updates with only a single invariant check: the loop in `transferInner(ts)` can perform multiple field updates of `income` and `expenses`, however the `invariant()` will only be checked at the end of the loop.
- Temporarily break an invariant: it is fine if during the `for` loop, `expenses > income`, provided that this is fixed before the end of the loop.
- Pass the state of an 'invalid' object around, in a safe manner: an `AccountBox` contains the state of `Account`, but not the invariant method.

Under our strict invariant protocol, the invariant holds for all reachable objects. The sub invariant pattern allows to control when an object is required to be valid. Instead, other protocols strive to allow the invariant to be observed broken in controlled conditions defined by the protocol itself.

The sub invariant pattern offers interesting guarantees: any object '`a`' with a `subInvariant()` method that is checked by the `invariant()` method of an object '`b`' will respect its `subInvariant()` in all contexts where '`b`' is involved in execution. This is because whenever '`b`' is involved in execution, its invariant holds. Moreover, `a`'s `subInvariant()` can be observed as `false` only if a capsule mutator of '`b`' is currently active (that is, being executed), or `b` is now garbage collectable. Thus, even when there is no reachable reference to `b` in the current stack frame, if no capsule mutator on `b` is active, `a`'s `subInvariant()` will hold.

In the former example, this means that if you can refer to an `Account`, you can be sure that its `income >= expenses`; if you have an `AccountBox` then you can be sure that either `income >= expenses` or a capsule mutator of the corresponding `Account` object is currently active. This closely resembles some visible state semantic protocols, aiming to ensure that either an object's invariant holds, or one of its methods is currently active.

Another interesting and natural application of the sub invariant pattern would be to support a version of the GUI such that, when a `Widget`'s position is updated, the `Widget` can in turn update the coordinates of its parent `Widget`s, in order to re-establish their `subInvariants`. This would also make the GUI follow the versions of the composite pattern were objects have references to their 'parent' nodes. The main idea is to define an interface `HasSubInvariant`, that denotes `Widgets` with a `subInvariant()` method. Then, `WidgetWithInvariant` is a decorator over a `Widget`; the invariant method of a `WidgetWithInvariant` checks the `subInvariant()` of each widget in its ROG.

We define `SafeMovable` as a `Widget` and `HasSubInvariant`. Since `subInvariant()` methods don't have the restrictions of invariant methods, it allows `SafeMovable` to be significantly simpler than the version shown before in Section 6.1.

```
interface HasSubInvariant{ read method Bool subInvariant(); }
class SafeMovable implements Widget,HasSubInvariant {
  Int width = 300; Int height = 300;
  Int left; Int top;  // Here we do not use a box, thus all the state
```

```
  mut Widgets c;      // is in SafeMovable.
  mut Widget parent;//We add a parent field
  @Override read method Int left(){ return this.left; }
  @Override read method Int top(){ return this.top; }
  @Override read method Int width(){ return this.width; }
  @Override read method Int height(){ return this.height; }
  @Override read method read Widgets children(){ return this.c; }
  @Override mut method Void dispatch(Event e){
    for(mut Widget w :this.c){ w.dispatch(e); }
  }
  @Override read method Bool subInvariant(){ /*same of original GUI*/ }
  SafeMovable(mut Widget parent,mut Widgets c){
    this.c=c;          //SafeMovable no longer has an invariant,
    this.left=5;       //so we impose no restrictions on its constructor
    this.top=5;
    this.parent=parent;
    c.add(new Button(0,0,10,10,new MoveAction(this));
  }
}
class MoveAction implements Action{
  mut SafeMovable o;
  MoveAction(mut SafeMovable o){ this.o = o; }
  mut method Void process(Event e){
    this.o.left+=1;
    Widget p = this.o.parent;
    ... // mutate p to re-establish its subInvariant
  }
}
class WidgetWithInvariant implements Widget{
  capsule Widget w;
  @Override read method Int left(){ return this.w.left; }
  @Override read method Int top(){ return this.w.top; }
  @Override read method Int width(){ return this.w.width; }
  @Override read method Int height(){ return this.w.height; }
  @Override read method read Widgets children(){ return this.w.c; }
  @Override mut method Void dispatch(Event e){ w.dispatch(e); }
  @Override read method Bool invariant(){ return wInvariant(w); }
  static method Bool wInvariant(read Widget w){
    for(read Widget wi:w.children()){ if(!wInvariant(wi)){ return false; } }
    //Check that the subInvariant of all of w's descendants holds
    if(!(w instanceof HasSubInvariant)){ return true; }
    HasSubInvariant si = (HasSubInvariant)w;
    return si.subInvariant();
  }
  WidgetWithInvariant(capsule Widget w){ this.w = w; }
}
... // main expression
//#$ is a capability operation making a Gui object
mut Widget top=new WidgetWithInvariant(new SafeMovable(...))
Gui.#$().display(top);
```

In this way, the method `WidgetWithInvariant.dispatch()` is the only capsule mutator, hence the only invariant checks will be at the end of `WidgetWithInvariant`'s constructor and dispatch methods.

Importantly, this allows the graph of widgets to be cyclic and for each to freely mutate each other, even if such mutations (temporarily) violate their `subInvariant`'s. In this way a widget can access its parent (whose `subInvariant()` may not hold) in order to re-establish it. Note that this trade off is logically unavoidable:

in order to manipulate a parent in order to fix it, the parent must be reachable, but by mutating a `Widget`'s position, its parent may become invalid. Thus if `Widgets` were to encode their validity in their `invariant()` methods they could not have access to their parents. Instead, by encoding their validity in a `subInvariant()` method, they can access invalid widgets, but this comes at a cost: the programmer must reason as to when `Widgets` are valid, as we described above.

The Transform Pattern

Recall the GUI case study from Section 6.1, where we had a `Widget` interface and a `SafeMovable` (with an invariant) that implements `Widget`. Suppose we want to allow `Widgets` to be scaled, 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(w)` 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 mention their `children()` in their `invariant()`. A simple and practical solution would be to define a `transform(t)` method in `Widget`, and a `Transformer` interface like so:

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

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

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

In the context of reference capabilities, `imm` lambdas/closures will only be allowed to capture `imm` and `capsule` local variables. Note that the `Transformer` parameter to `transform` is `capsule` and the method `Trasformer.apply` takes an `capsule` receiver. In particular, this means that `transform` will be able to call the lambda at most once, and lambdas cannot be saved and passed to multiple calls to `transform`. However, we could instead make `transform` take an `imm Transformer`, and make `Transformer.apply` be an `imm` method, this would allow lambdas to be freely copied and called multiple times, however they would not be able to capture pre-existing mutable objects.

Using Patterns Together: A general and flexible Graph class

Here we rely on all the patterns shown above to encode a general library for `Graph`s of `Node`s. Users of this library can define personalised kinds of nodes, with their own personalised sub invariant. The library will ensure that no matter how the library is used, for any accessible `Graph`, each user defined sub invariant of its `Node`s holds. Note that those sub invariants are not restricted to the local state of a node; since they can explore the state of all reachable nodes, they may even depend upon the whole graph.

The `Nodes` are guaranteed to be encapsulated by the `Graph`, however they can be arbitrarily modified by user defined transformations using the Transform Pattern.

```
interface Transform<T>{ method read T apply(mut Nodes nodes); }

interface Node{
  read method Bool subInvariant(read Nodes nodes)
  mut method mut Nodes directConnections()
}
class Nodes{//just an ordered set of nodes
```

```
  mut method Void add(mut Node n){..}
  read method Int indexOf(read Node n){..}
  mut method Void remove(read Node n){..}
  mut method mut Node get(Int index){..}
}
class Graph{
  capsule Nodes nodes; //box pattern
  Graph(capsule Nodes nodes){..}
  read method read Nodes getNodes(){ return this.nodes; }
  <T> mut method read T transform(Transform<T> t){
    mut Nodes ns=this.nodes;//capsule mutator with a single use of 'this'
    return t.apply(ns);
  }
  read method Bool invariant(){
    for(read Node n: this.nodes){if(!n.subInvariant(this.nodes)){return false;}}
    return true;
  }
}
```

We now show how our **Graph** library allows the invariant of the various **Node**s to be customised by the library user, and arbitrary transformations can be performed on the **Graph**s. This is a generalisation of the example proposed by [43](section 4.2) as one of the hardest problems when it comes to enforcing invariants.

Note how there are only a minimal set of operations defined in the above code, others can be freely defined by the user code, as demonstrated below:

```
class MyNode{
  mut Nodes directConnections;
  mut method mut Nodes directConnections(){ return this.directConnections; }
  MyNode(mut Nodes directConnections){..}
  read method Bool subInvariant(read Nodes nodes){
    /* any condition on this or nodes */}
  capsule method read MyNode addToGraph(mut Graph g){..}
  read method Void connectWith(read Node other, mut Graph g){..}
}
...
mut Graph g = new Graph(new Nodes());
read MyNode n1 = new MyNode(new Nodes())).addToGraph(g);
read MyNode n2 = new MyNode(new Nodes())).addToGraph(g);
//lets connect our two nodes
n1.connectWith(n2,g);
```

Here we define a **MyNode** class, where the `subInvariant(nodes)` can express any property over **this** and `nodes`, such as properties over their direct connections, or any other reachable node.

We can define methods in **MyNode** to add our nodes to graphs and to connect them with other nodes. Note that the method `addToGraph(g)` is marked as **capsule**: this ensures that the node is not in any other graph. In contrast, the method `connectWith(other, g)` is marked as **read**, even though it is clearly intend to modify the ROG of **this**. It works by recovering a **mut** reference to **this** from the **mut Graph**.

These methods can be implemented like this:

```
read method Void connectWith(read Node other,mut Graph g){
  Int i1=g.getNodes().indexOf(this);
  Int i2=g.getNodes().indexOf(other);
  if(i1==-1 || i2==-1){throw /*error nodes not in g*/;}
  g.transform(ns->{
    mut Node n1=ns.get(i1);
    mut Node n2=ns.get(i2);
    n1.directConnections().add(n2);
  });
```

```
}
capsule method read MyNode addToGraph(mut Graph g){
  return g.capTransform(ns->{
  mut MyNode n1=this;//single usage of capsule 'this'
  ns.add(n1);
  });
}
```

As you can see, both methods rely on the transform pattern.

These transformation operations are very general since they can access the `mut Nodes` of the `Graph` and any `capsule` or `imm` data from outside. Note how the body of the `capsule` lambda in `connectWith(other,g)`, can not capture the `read this` or the `read other`, but we get their (immutable) indexes and recover the concrete objects from the `mut Nodes ns` object. In this way, we also obtain more useful `mut` references to those nodes. On the other hand, note how in `addToGraph(g)` we use the reference to the `capsule this` within the lambda, this allows the lambda to be safely typed as `capsule`, since there can be no other aliases to `this`, and the `this` variable cannot be used again in the method.

8. Integration in L42

In the last version of L42, invariants have been integrated with caching and automatic parallelism; it would be out of this article's scope to explain in detail this integration, but the overall idea is that an invariant is seen as a `Void @Cache.Now` method, who's result is pre-computed. The language ensures that `@Cache.Now` methods are recomputed whenever their result may change; any exceptions are propagated immediately, and are not cached. The type-system requires that any method that could alter the result of a `Cache.Now` method (except via a field update) must be marked with `@Cache.Clear` and respect our capsule-mutator restrictions. L42 requires an explicit `@Cache.Clear` so as to make it clear in the code that such methods has special type-system restrictions. This is more general than invariant checking however, as `Cache.Now` methods can return a meaningful result, and not simply success or exception. L42 also supports other kinds of cached methods, which get computed in parallel when an instance of the corresponding class is created, or when their result may be altered.

L42 libraries rely on a very expressive form of metaprogramming to generate a lot of boilerplate/redundant code. In L42 many tasks can be either manually performed by writing code directly, or partially automated by code generation. L42 allows writing `class` methods (similar to a static method in Java) with appropriate parameters instead of `invariants` method and capsule mutators. The bodies of such methods don't have special restrictions as they cannot see `this`, instead the meta-programming generates appropriate instance methods, conforming to our restrictions, which call the user provided `class` methods.

Our restrictions are also checked by the type system, so even if the user manually writes these methods, instead of relying on the metaprogramming, they still cannot break our invariant protocol.

To make this work more accessible to programmers familiar with Java/C#, we have shown our examples in a more Java-like syntax. Here you can see our `ShippingList` example from Section 4 in the full L42 Syntax:

```
ShippingList = Data:{
  capsule Items items
  @Cache.Now
  class method Void invariant(read Items items) =
    X[items.weight()<=300Num]
  @Cache.Clear
  class method Void addItem(mut Items items,Item item) =
    items.add(item)
}
```

In this example, the `Data` decorator generates a factory method, a `mut method Void addItem(Item item)` and a lot of other utility methods, including equality and conversion to string. The `@Cache.Now` annotation causes the `invariant` method to be automatically computed, and recomputed every time a `@Cache.Clear` method is called. The `X[...]` notation used in `invariant` is an assert statement: it throws an unchecked exception if it's argument is false. Please refer to `L42.is/tutorial.xhtml` for more information.

9. Related Work

Reference Capabilities

We rely on a combination of RCs supported by at least three languages/lines of research: L42 [6, 7, 8, 9], Pony [10, 11], and Gordon *et al.* [12]. They all support full/deep interpretation (see page 5), without back doors. Former works [44, 45, 46, 47, 48] (which eventually enabled the work of Gordon *et al.*) do not consider promotion and infers uniqueness/isolation/immutability only when starting from references that have been tracked with restrictive annotations along their whole lifetime. Other approaches like Javari [13, 49] and Rust [31] provide back doors, which are not easily verifiable as being used properly.

Ownership [50, 16, 51] is a popular form of aliasing control often used as a building block for static verification [52, 40]. However, ownership does not require the whole ROG of an object to be 'owned'. This complicates restricting the data accessible by invariants.

Object Capabilities

In the literature, OCs are used to provide a wide range of guarantees, and many variations are present. Object capabilities [53], in conjunction with reference capabilities, are able to enforce purity of code in a modular way, without requiring the use of effects or monads. L42 and Gordon *et al.* use OCs simply to reason about I/O and non-determinism. This approach is best exemplified by Joe-E [26], 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 a simplified model of immutability: immutable classes must be final and have 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 grained representation of mutability: it is *reference-based* instead of *class-based*. Thanks to this crucial difference, in our work every method taking only **`read`** or **`imm`** *references* receivers and parameters is pure, regardless of their class type. In particular, we allow the parameter of such a method to be mutated later on by other code.

Invariant protocols

Invariants are a fundamental part of the design by contract methodology. Invariant protocols differ wildly and can be unsound or complicated, particularly due to re-entrancy and aliasing [42, 54, 55].

While invariant protocols all check and assume the invariant of an object after its construction, they handle invariants differently across object lifetimes. Popular 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 [5]. Invariants need to be constantly maintained between calls to public methods.
- 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 [56, 54]. Some approaches ensure the invariant of the receiver of the *calling* method, rather than the *called* method [57]. JML [58] relaxes these requirements for helper methods, whose semantics are the same as if they were inlined.
- The same as above, but only for the bodies of 'selectively exported' (i.e. not instance-private) methods, and only for 'qualified' (i.e. not **`this`**) calls [55].
- 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' [3].

These different protocols can be deceivingly similar. Note that all those approaches fail our strict requirements and allow for broken objects to be observed. 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).

Security and Scalability

Our approach allows verifying an object's invariant independently of the execution context. This is in contrast to 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. 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 [59, 60], it is possible to consider a verified core and a runtime verified boundary. One can see our approach as an extremely modularized version of such a system: every class is its own verified core, and the rest of the code could have Byzantine behaviour. Our proofs show that every class that compiles/type checks is soundly handled by our protocol, independently of the behaviour of code that uses such a 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 [61]: 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`: checking contracts at load time is impractical and is not done by any verification systems we know of.

Static Verification

AutoProof [62] is 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 [1] is a language where all code is statically verified. It supports invariants with its `{:autocontracts}` annotation, which treats a class's `Valid()` function as the invariant and injects 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.

Spec# [63] 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 [64] 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 [3], where an invariant only depends on objects it owns; or *visibility* based [27, 65], 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 allows invariants to query owned mutable objects whose ROG is not fully encapsulated. However as we showed in Section 6.1, 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 its methods. Spec# also works with existing .NET libraries by annotating them with contracts, however such annotations are not verified. Spec#, like our approach, 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. [66]

Specification languages

Using a specification language based on the mathematical metalanguage and different from the programming 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 [67] discovered that developers expect specification languages to follow the semantics of the underlying 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 [68]. Dafny [1] 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 [69] '*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 [70]. 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. In Spec# the following 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 restrict 'pure' functions so that they can only read final fields and call other pure functions. This is the approach used by [71]. 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.

Runtime Verification Tools

By looking to a survey by Voigt *et al.* [72] and the extensive MOP project [73], it seems that most runtime verification (RV) tools 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 [73]. 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 [74] 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 simple 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 [56] and Microsoft Code Contracts [75].

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 [76]. However, such work is only interested in pre and post-conditions, not invariants.

Our invariant protocol is much stricter than visible state semantics, and keeps the invariant under tight control. Gopinathan *et al.*'s. [5] 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 soundess guarantee does not scale to programs using such features.

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 rollback 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 RCs to enforce strong exception safety is a much simpler alternative, providing the same level of safety, albeit being more restrictive.

Chaperones and impersonators [77] lifts the techniques of gradual typing [78, 59, 60] 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 been 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.

10. Conclusion

In this paper we (1) identified the essential language features that support representation invariants in object-oriented verification; (2) presented a full formalism for our approach with capabilities that is proved to soundly guarantee that all objects involved in execution are valid; (3) conducted extensive case studies showing that we require many order of magnitude less runtime checking than *visible state semantics* and three times less annotation burden than an equivalent version in Spec#. We hope that as a result of this work, the software verification community will make more use of the advanced general purpose language features, such as capabilities, appearing in modern languages to achieve its goals.

Our approach follows the principles of *offensive programming* [79] where no attempt to fix or recover an invalid object is performed. 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 RCs and OCs, whose popularity is growing, and we expect future languages to support some variations of these. Crucially, any language already designed with such a support can also support our invariant protocol with minimal added complexity.

References

[1] K. R. M. Leino, Developing verified programs with dafny, in: Proceedings of the 2012 ACM Conference on High Integrity Language Technology, HILT '12, December 2-6, 2012, Boston, Massachusetts, USA, 2012, pp. 9–10. doi:10.1145/2402676.2402682.

[2] B. Meyer, Object-Oriented Software Construction, 1st Edition, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[3] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, W. Schulte, Verification of object-oriented programs with invariants, Journal of Object Technology 3 (6) (2004) 27–56. doi:10.5381/jot.2004.3.6.a2.

[4] C. F. Bolz, A. Cuni, M. FijaBkowski, M. Leuschel, S. Pedroni, A. Rigo, Allocation removal by partial evaluation in a tracing jit, in: Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM '11, ACM, New York, NY, USA, 2011, pp. 43–52. doi:10.1145/1929501.1929508. URL http://doi.acm.org/10.1145/1929501.1929508

[5] M. Gopinathan, S. K. Rajamani, Runtime verification, Springer-Verlag, Berlin, Heidelberg, 2008, Ch. Runtime Monitoring of Object Invariants with Guarantee, pp. 158–172. doi:10.1007/978-3-540-89247-2_10.

[6] M. Servetto, E. Zucca, Aliasing control in an imperative pure calculus, in: X. Feng, S. Park (Eds.), Programming Languages and Systems - 13th Asian Symposium (APLAS), Vol. 9458 of Lecture Notes in Computer Science, Springer, 2015, pp. 208–228. doi:10.1007/978-3-319-26529-2_12.

[7] M. Servetto, D. J. Pearce, L. Groves, A. 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.

[8] G. Lagorio, M. Servetto, Strong exception-safety for checked and unchecked exceptions, Journal of Object Technology 10 (2011) 1:1–20. doi:10.5381/jot.2011.10.1.a1.

[9] P. Giannini, M. Servetto, E. Zucca, Types for immutability and aliasing control, in: ICTCS'16 - Italian Conf. on Theoretical Computer Science, Vol. 1720 of CEUR Workshop Proceedings, CEUR-WS.org, 2016, pp. 62–74. URL http://ceur-ws.org/Vol-1720/full5.pdf

[10] S. Clebsch, S. Drossopoulou, S. Blessing, A. McNeil, Deny capabilities for safe, fast actors, in: Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, ACM, 2015, pp. 1–12. doi:10.1145/2824815.2824816.

[11] S. Clebsch, J. Franco, S. Drossopoulou, A. M. Yang, T. Wrigstad, J. Vitek, Orca: Gc and type system co-design for actor languages, Proceedings of the ACM on Programming Languages 1 (OOPSLA) (2017) 72. doi:10.1145/3133896.

[12] C. S. Gordon, M. J. Parkinson, J. Parsons, A. Bromfield, J. Duffy, Uniqueness and reference immutability for safe parallelism, in: ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2012), ACM Press, 2012, pp. 21–40. doi:10.1145/2384616.2384619.

[13] M. S. Tschantz, M. D. Ernst, Javari: Adding reference immutability to Java, in: ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2005), ACM Press, 2005, pp. 211–230. doi:10.1145/1094811.1094828.

[14] A. Birka, M. 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), 2004, pp. 35–49. doi:10.1145/1035292.1028980.

[15] J. Östlund, T. Wrigstad, D. Clarke, B. Åkerblom, Ownership, uniqueness, and immutability, in: R. F. Paige, B. Meyer (Eds.), International Conference on Objects, Components, Models and Patterns, Vol. 11 of Lecture Notes in Computer Science, Springer, 2008, pp. 178–197. doi:10.1007/978-3-540-69824-1_11.

[16] Y. Zibin, A. Potanin, P. Li, M. Ali, M. D. Ernst, Ownership and immutability in generic Java, in: ACM SIGPLAN Conference on Object-Oriented Programming,Systems, Languages and Applications (OOPSLA 2010), 2010, pp. 598–617. doi:10.1145/1869459.1869509.

[17] A. Potanin, J. Östlund, Y. Zibin, M. D. Ernst, Immutability, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 233–269. doi:10.1007/978-3-642-36946-9_9.

[18] J. Boyland, Alias burying: Unique variables without destructive reads, Software: Practice and Experience 31 (6) (2001) 533–553. doi:10.1002/spe.370.

[19] P. Giannini, M. Servetto, E. Zucca, J. Cone, Flexible recovery of uniqueness and immutability, Theoretical Computer Science 764 (2019) 145 – 172. doi:10.1016/j.tcs.2018.09.001.

[20] D. Clarke, T. Wrigstad, External uniqueness is unique enough, in: ECOOP'03 - Object-Oriented Programming, Vol. 2473 of Lecture Notes in Computer Science, Springer, 2003, pp. 176–200. doi:10.1007/978-3-540-45070-2_9.

[21] P. Haller, M. Odersky, Capabilities for uniqueness and borrowing, in: T. D'Hondt (Ed.), ECOOP'10 - Object-Oriented Programming, Vol. 6183 of Lecture Notes in Computer Science, Springer, 2010, pp. 354–378. doi:10.1007/978-3-642-14107-2_17.

[22] D. Abrahams, Exception-Safety in Generic Components, Springer Berlin Heidelberg, Berlin, Heidelberg, 2000, pp. 69–79. doi:10.1007/3-540-39953-4_6.

[23] M. S. Miller, K.-P. Yee, J. Shapiro, et al., Capability myths demolished, Tech. rep., Technical Report SRL2003-02, Johns Hopkins University Systems Research Laboratory, 2003. http://www. erights. org/elib/capability/duals (2003).

[24] J. Noble, S. Drossopoulou, M. S. Miller, T. Murray, A. Potanin, Abstract data types in object-capability systems (2016).

[25] P. A. Karger, Improving security and performance for capability systems, Ph.D. thesis, Citeseer (1988).

[26] M. Finifter, A. Mettler, N. Sastry, D. Wagner, Verifiable functional purity in java, in: Proceedings of the 15th ACM conference on Computer and communications security, ACM, 2008, pp. 161–174. doi:10.1145/1455770.1455793.

[27] M. Barnett, D. 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, 2004, pp. 54–84. doi:10.1007/978-3-540-27764-4_5.

[28] W. Dietl, P. Müller, Universes: Lightweight ownership for jml, JOURNAL OF OBJECT TECHNOLOGY 4 (8) (2005) 5–32.

[29] D. Cunningham, W. Dietl, S. Drossopoulou, A. Francalanza, P. Müller, A. J. Summers, Universe types for topology and encapsulation, in: Formal Methods for Components and Objects, 2008, pp. 72–112.

[30] Y. A. Feldman, O. Barzilay, S. Tyszberowicz, Jose: Aspects for design by contract, in: Software Engineering and Formal Methods, 2006. SEFM 2006. Fourth IEEE International Conference on, IEEE, 2006, pp. 80–89. doi:10.1109/SEFM.2006.26.

[31] N. D. Matsakis, F. S. Klock II, The rust language, in: ACM SIGAda Ada Letters, Vol. 34, ACM, 2014, pp. 103–104. doi:10.1145/2663171.2663188.

[32] J. Bloch, Effective Java (2Nd Edition) (The Java Series), 2nd Edition, Prentice Hall PTR, 2008.

[33] A. Igarashi, B. C. Pierce, P. Wadler, Featherweight Java: a minimal core calculus for Java and GJ, ACM Transactions on Programming Languages and Systems 23 (3) (2001) 396–450.

[34] B. C. Pierce, Types and programming languages, MIT press, 2002.

[35] R. N. S. Rowe, S. J. Van Bakel, Semantic types and approximation for featherweight java, Theor. Comput. Sci. 517 (2014) 34–74. doi:10.1016/j.tcs.2013.08.017. URL https://doi.org/10.1016/j.tcs.2013.08.017

[36] A. Alexandrescu, The D Programming Language, 1st Edition, Addison-Wesley Professional, 2010.

[37] B. Meyer, Eiffel: The Language, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[38] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, K. R. 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, 2005, pp. 364–387. doi:10.1007/11804192_17.

[39] M. Fähndrich, M. Barnett, F. Logozzo, Embedded contract languages, in: Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010, 2010, pp. 2103–2110. doi:10.1145/1774088.1774531.

[40] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, H. Venter, Specification and verification: the spec# experience, Communications of the ACM 54 (6) (2011) 81–91. doi:10.1145/1953122.1953145.

[41] K. R. M. Leino, P. Müller, A. Wallenburg, Flexible immutability with frozen objects, in: Verified Software: Theories, Tools, Experiments, Second International Conference, VSTTE 2008, Toronto, Canada, October 6-9, 2008. Proceedings, 2008, pp. 192–208. doi:10.1007/978-3-540-87873-5_17.

[42] K. R. M. Leino, P. Müller, Object invariants in dynamic contexts, in: European Conference on Object-Oriented Programming, Springer, 2004, pp. 491–515. doi:10.1007/978-3-540-24851-4_22.

[43] A. J. Summers, S. Drossopoulou, P. Müller, The need for flexible object invariants, in: International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming, IWACO '09, ACM, New York, NY, USA, 2009, pp. 6:1–6:9. doi:10.1145/1562154.1562160.
URL http://doi.acm.org/10.1145/1562154.1562160

[44] J. Boyland, Semantics of fractional permissions with nesting, ACM Transactions on Programming Languages and Systems 32 (6) (2010). doi:10.1145/1749608.1749611.

[45] J. Boyland, Checking interference with fractional permissions, in: International Static Analysis Symposium, Springer, 2003, pp. 55–72.

[46] J. Hogg, Islands: Aliasing protection in object-oriented languages, in: ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1991, ACM Press, 1991, pp. 271–285.

[47] F. Smith, D. Walker, J. G. Morrisett, Alias types, in: Proceedings of the 9th European Symposium on Programming Languages and Systems, ESOP '00, Springer-Verlag, London, UK, UK, 2000, pp. 366–381.
URL http://dl.acm.org/citation.cfm?id=645394.651903

[48] A. Aiken, J. S. Foster, J. Kodumal, T. 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, 2003, pp. 129–140. doi:10.1145/781131.781146.

[49] J. Boyland, Why we should not add readonly to Java (yet), Journal of Object Technology 5 (5) (2006) 5–29. doi:10.5381/jot.2006.5.5.a1.

[50] D. Clarke, J. Östlund, I. Sergey, T. Wrigstad, Ownership types: A survey, in: D. Clarke, J. Noble, T. Wrigstad (Eds.), Aliasing in Object-Oriented Programming. Types, Analysis and Verification, Vol. 7850 of Lecture Notes in Computer Science, Springer, 2013, pp. 15–58. doi:10.1007/978-3-642-36946-9_3.

[51] W. Dietl, S. Drossopoulou, P. Müller, Generic universe types, in: ECOOP'07 - Object-Oriented Programming, Vol. 4609 of Lecture Notes in Computer Science, Springer, 2007, pp. 28–53. doi:10.1007/978-3-540-73589-2_3.

[52] P. Müller, Modular specification and verification of object-oriented programs, Springer-Verlag, 2002. doi:10.1007/3-540-45651-1.

[53] M. S. Miller, Robust composition: Towards a unified approach to access control and concurrency control, Ph.D. thesis, Johns Hopkins University, Baltimore, Maryland, USA (May 2006).

[54] S. Drossopoulou, A. Francalanza, P. Müller, A. J. Summers, A unified framework for verification techniques for object invariants, in: European Conference on Object-Oriented Programming, Springer, 2008, pp. 412–437. doi:10.1007/978-3-540-70592-5_18.

[55] B. Meyer, Class invariants: Concepts, problems, solutions, arXiv preprint arXiv:1608.07637 (2016).

[56] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, E. Poll, An overview of jml tools and applications, International Journal on Software Tools for Technology Transfer 7 (3) (2005) 212–232. doi:10.1007/s10009-004-0167-4.

[57] P. Müller, A. Poetzsch-Heffter, G. T. Leavens, Modular invariants for layered object structures, Sci. Comput. Program. 62 (3) (2006) 253–286. doi:10.1016/j.scico.2006.03.001.
URL https://doi.org/10.1016/j.scico.2006.03.001

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

[59] A. Takikawa, T. S. Strickland, C. Dimoulas, S. Tobin-Hochstadt, M. 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, 2012, pp. 793–810. doi:10.1145/2384616.2384674.

[60] T. Wrigstad, F. Z. Nardelli, S. Lebresne, J. Östlund, J. 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, 2010, pp. 377–388. doi:10.1145/1706299.1706343.

[61] M. Parkinson, Class invariants: The end of the road?, Aliasing, Confinement and Ownership in Object-oriented Programming (IWACO) (2007) 9.

[62] N. Polikarpova, J. Tschannen, C. A. Furia, B. Meyer, Flexible invariants through semantic collaboration, in: FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings, 2014, pp. 514–530. doi:10.1007/978-3-319-06410-9_35.

[63] M. Barnett, K. R. M. Leino, W. 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, Springer-Verlag, Berlin, Heidelberg, 2005, pp. 49–69. doi:10.1007/978-3-540-30569-9_3.

[64] D. A. Naumann, M. Barnett, Towards imperative modules: Reasoning about invariants and sharing of mutable state,

Theor. Comput. Sci. 365 (1-2) (2006) 143–168. doi:10.1016/j.tcs.2006.07.035.
URL https://doi.org/10.1016/j.tcs.2006.07.035

[65] K. R. M. Leino, P. Müller, Object invariants in dynamic contexts, in: ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings, 2004, pp. 491–516. doi:10.1007/978-3-540-24851-4_22.

[66] K. R. M. Leino, W. Schulte, Exception safety for c#, Proceedings of the Second International Conference on Software Engineering and Formal Methods, 2004. SEFM 2004. (2004) 218–227.

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

[68] P. Chalin, F. Rioux, Jml runtime assertion checking: Improved error reporting and efficiency using strong validity, FM 2008: Formal Methods (2008) 246–261doi:10.1007/978-3-540-68237-0_18.

[69] M. Barnett, D. A. Naumann, W. Schulte, Q. 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.

[70] M. Barnett, W. Schulte, Runtime verification of .net contracts, Journal of Systems and Software 65 (3) (2003) 199–208. doi:10.1016/S0164-1212(02)00041-9.

[71] C. Flanagan, Hybrid types, invariants, and refinements for imperative objects, in: In International Workshop on Foundations and Developments of Object-Oriented Languages, 2006.

[72] J. Voigt, W. Irwin, N. Churcher, Comparing and Evaluating Existing Software Contract Tools, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 49–63. doi:10.1007/978-3-642-32341-6_4.

[73] P. O. Meredith, D. Jin, D. Griffith, F. Chen, G. Roşu, An overview of the mop runtime verification framework, International Journal on Software Tools for Technology Transfer 14 (3) (2012) 249–289. doi:10.1007/s10009-011-0198-6.

[74] M. Gorbovitski, T. Rothamel, Y. A. Liu, S. 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.

[75] M. Fähndrich, M. Barnett, F. Logozzo, Embedded contract languages, in: Proceedings of the 2010 ACM Symposium on Applied Computing, ACM, 2010, pp. 2103–2110. doi:10.1145/1774088.1774531.

[76] R. B. Findler, M. Felleisen, Contract soundness for object-oriented languages, in: ACM SIGPLAN Notices, Vol. 36, ACM, 2001, pp. 1–15. doi:10.1145/504311.504283.

[77] T. S. Strickland, S. Tobin-Hochstadt, R. B. Findler, M. 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, 2012, pp. 943–962. doi:10.1145/2384616.2384685.

[78] A. Takikawa, D. Feltey, E. Dean, M. Flatt, R. B. Findler, S. Tobin-Hochstadt, M. Felleisen, Towards practical gradual typing, in: LIPIcs-Leibniz International Proceedings in Informatics, Vol. 37, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015. doi:10.4230/LIPIcs.ECOOP.2015.4.

[79] R. Stephens, Beginning Software Engineering, Wiley, 2015.

Appendix A. Invariant Protocol Proof and Type System Requirements

As previously discussed, we provide a set of requirements that the type system needs to ensure, and prove the soundness of our invariant protocol over these, in this way we are parametric over the concrete typesystem. In Appendix B, we present an example typesystem and prove that it satisfies these requirements.

Auxiliary Definitions

To express our type system assumptions, we first need some auxiliary definitions.

First, we inductively define the set of objects in the reachable object graph (*rog*) of a location l:

$l' \in rog(\sigma, l)$ iff:

- $l' = l$, or
- $\exists f$ such that $l' \in rog(\sigma, \sigma[l.f])$

We define the *mrog* of an l to be the locations reachable from l by traversing through any number of **mut** and **rep** fields:

$l' \in mrog(\sigma, l)$ iff:

- $l' = l$, or
- $\exists f$ such that $\mathrm{C}^{\sigma}_{l}.f = \kappa\,\_\,f$, $\kappa \in \{\mathbf{mut}, \mathbf{rep}\}$, and $l' \in mrog(\sigma, \sigma[l.f])$

Thus the *mrog* of l are the objects that could be mutated via a reference to l.

We define what it means for an l to be *reachable* from an expression or context:

- $reachable(\sigma, e, l)$ iff $\exists l' \in e$ such that $l \in rog(\sigma, l')$

- $\mathit{reachable}(\sigma, \mathcal{E}, l)$ iff $\exists l' \in \mathcal{E}$ such that $l \in \mathit{rog}(\sigma, l')$

We now define what it means for an object to be *immutable*: it is in the *rog* of an `imm` reference or a *reachable* `imm` field:

$\mathit{immutable}(\sigma, e, l)$ iff $\exists l'$ such that:

- $\texttt{imm}\, l' \in e$, and $l \in \mathit{rog}(\sigma, l')$, or
- $\mathit{reachable}(\sigma, e, l')$, $\mathrm{C}^{\sigma}_{l'}.f = \texttt{imm}\ \_\ f$, and $l \in \mathit{rog}(\sigma, \sigma[l'.f])$, for some f

Now we can define what it means for an l to be *mutatable*[32] by an expression e: something reachable from l can also be reached by using a `mut` or `capsule` reference in e, and traversing through any number of `mut` or `rep` fields:

$\mathit{mutatable}(\sigma, e, l)$ iff $\exists l', l''$ such that:

- $l' \in \mathit{rog}(\sigma, l)$
- $\mu\, l'' \in e$ with $\mu \in \{\texttt{mut}, \texttt{capsule}\}$
- $l' \in \mathit{mrog}(\sigma, l'')$

The idea is that e could mutate something reachable from l: by using l'' to get a `mut` reference to l', and then performing a field update on it; the new field value for l' would then be observable through l. In particular, we will require the typesystem to ensure that e can only mutate state observable from l if l is *mutatable*.

Finally, we model the *encapsulated* property of `capsule` references:

$\mathit{encapsulated}(\sigma, \mathcal{E}, l)$ iff $\forall l' \in \mathit{rog}(\sigma, l)$, if $\mathit{mutatable}(\sigma, \mathcal{E}[\texttt{capsule}\, l], l')$, then not $\mathit{reachable}(\sigma, \mathcal{E}, l')$.

That is, a location l found in a context $\mathcal{E}$ is encapsulated if all *mutatable* objects in its *rog* would be unreachable with that single use of l removed. That single use of l is the connection preventing those *mutatable* objects from being garbage collectable.

Type System Requirements

As we do not have a concrete type system, we need to assume some properties about the expressions that it admits. Rather than requiring each expression during reduction to be well-typed, we instead let the type-system impose restrictions on method bodies, and type-check the initial expression, we then require properties on all future memories and expressions (i.e. *validStates*). In Appendix B we show such a type-system and prove it satisfies these requirements, but these requirements do *not* hold for arbitrary well-typed $\sigma|e$ pairs, only for *validStates*. This allows the type-system to be simpler, in particular, as the initial main expression can only have `mut` references to c (an object with no fields(), the type-system need not check that the heap structure and reference capabilities in the main expressions are consistent.

First we require that fields and methods are only given values with the correct reference capabilities, i.e. the field initialisers of `new` expressions, the right hand sides of update expressions, and the receiver and parameters of method calls have the capabilities required by the field declarations/method signatures:

Requirement 1 (Type Consistency)**.**

1. If $\mathit{validState}(\mathcal{E}[\texttt{new}\ C\texttt{(}\mu_1\ \_, ..., \mu_n\ \_\texttt{)}])$, then:
   - there is a $\texttt{class}\ C\ \texttt{implements}\ \_\ \texttt{\{}Fs\texttt{;}\_\texttt{\}}$
   - $Fs = \kappa_1\ \_\ \_, ..., \kappa_n\ \_\ \_$
   - $\mu_1 \leq \widetilde{\kappa}_1, ..., \mu_n \leq \widetilde{\kappa}_n$
2. If $\mathit{validState}(\mathcal{E}[\_\ l.f \texttt{=} \mu\ \_])$, then:
   - $\mathrm{C}^{\sigma}_{l}.f = \kappa\ \_\ f$
   - $\mu \leq \widetilde{\kappa}$
3. If $\mathit{validState}(\mathcal{E}[\mu_0\, l.m\texttt{(}\mu_1\ \_, ..., \mu_n\ \_\texttt{)}])$, then:
   - $\mathrm{C}^{\sigma}_{l}.m = \mu'_0\ \texttt{method}\ \_\ m\texttt{(}\mu'_1\ \_\ \_, ..., \mu'_n\ \_\ \_\texttt{)}\ \_$
   - $\mu_0 \leq \mu'_0, ..., \mu_n \leq \mu'_n$

[32] We use the term *mutatable* and not '*mutable*' as an object might be neither *mutatable* nor im*mutable*, e.g. if there are only `read` references to it.

This requirement also ensure that objects are created with the appropriate number of fields, and that fields and methods that are accessed/updated/called actually exist.

Now we define formal properties about our reference capabilities, thus giving them meaning. First we require that an *immutable* object can not also be *mutatable*: i.e. if an object is reachable from an `imm` reference or field, then no part of its *rog* can be reached by starting at a `mut` or `capsule` reference, and then traversing through `mut` and `rep` fields:

Requirement 2 (Imm Consistency)**.**
If $validState(\sigma, \mathcal{E}[e])$ and $immutable(\sigma, e, l)$, then not $mutatable(\sigma, e, l)$.

Thus e cannot use field accesses to obtain a `mut` or `capsule` reference to anything reachable from an *immutable* l. Note that this does not prevent *promotion* from a `mut` to an `imm`: an `as` expression can change a reference from `mut` to `imm`, provided that in the new state there are no longer any `mut` references to the *rog* of l. Note that from the definition of *mutatable* and *immutable*, it follows that if l is *immutable* in any e, then it is *immutable* in $\mathcal{E}[e]$, and not *mutatable* in any $e' \in \mathcal{E}[e]$.

We require that if something was not *mutatable*, it remains that way:

Requirement 3 (Mut Consistency)**.**
If $validState(\sigma, \mathcal{E}[e])$, not $mutatable(\sigma, e, l)$, and $\sigma|e \rightarrow^* \sigma'|e'$, then not $mutatable(\sigma', e', l)$.

Note that this holds even if l is *mutatable* through $\mathcal{E}$, thus an `as` expression cannot change a `read` or `imm` reference to `mut`, as the associated location will not be *mutatable* within the body of the `as` expression, even if there are `mut` references to the same object outside the `as`.

We require that any `capsule` reference is *encapsulated*, i.e. that no *mutatable* part of its *rog* is reachable through any other reference:

Requirement 4 (Capsule Consistency)**.**
If $validState(\sigma, \mathcal{E}[\texttt{capsule}\, l])$, then $encapsulated(\sigma, \mathcal{E}, l)$.

As all objects are created as `mut`, the only way to actually get a `capsule` reference is via an `as` expression. As our reduction rules impose no constraints on such expressions, the type-system must ensure that it only accepts a `as capsule` expression if it is guaranteed to return an *encapsulated* reference. Note that a specific typesystem's idea of "capsuleness" may in fact be stronger then *encapsulated*, but *encapsulated* is sufficient for our invariant protocol.

We require that field updates are only performed on `mut`/`capsule` receivers:

Requirement 5 (Mut Update)**.**
If $validState(\mathcal{E}[\mu\_.\_ \texttt{=} \_])$, then $\mu \leq \texttt{mut}$.

Finally we require strong exception safety: the body of a `try` block does not mutate objects that existed before the enclosing `try`–`catch` began executing and are reachable outside the `try` block:

Requirement 6 (Strong Exception Safety)**.**
If $validState(\sigma', \mathcal{E}_v[\texttt{try}^\sigma\texttt{\{}e\texttt{\}}\ \texttt{catch}\ \texttt{\{}e'\texttt{\}}])$, then $\forall l \in dom(\sigma)$, if $reachable(\sigma, \mathcal{E}_v[e'], l)$, then $\sigma(l) = \sigma'(l)$.

Note that this strong requirement *only* needs to hold because our `try`–`catch` can catch invariant failures: in L42, `try`–`catch`'s that catch *checked* exceptions do not need this restriction. Note that as our reduction rules never modify the body of a `catch`, it follows that if $validState(\sigma', \mathcal{E}_v[\texttt{try}^\sigma\texttt{\{\_\}}\ \texttt{catch}\ \texttt{\{}e\texttt{\}}])$, then for any $l \in dom(\sigma')$, if $l \notin dom(\sigma)$, then l is not *reachable* in $\mathcal{E}_v[e]$.

Usefull Lemmas

First we prove a few useful lemmas about the properties of references in our language.

We show that a sub-expression can mutate an object only if it is *mutatable*:

Lemma 1 (Non Mutating)**.**
If $validState(\sigma, \mathcal{E}[e])$, $l \in dom(\sigma)$, not $mutatable(\sigma, e, l)$, and $\sigma|e \rightarrow^* \sigma'|e'$, then $\sigma'(l) = \sigma(l)$.

Proof. By Mut Consistency, l never becomes *mutatable*, and so we never obtain a **mut** or **capsule** reference to it, thus by Mut Update, we never update the fields of l, and there are no reduction rules that remove from σ.

By the definition of *validState* and the reduction rules themselves, we can show that the main expression and heap never contain dangling references:

Lemma 2 (No Dangling)**.**

If $validState(\sigma, e)$ then:

- $\forall l \in e,\ l \in dom(\sigma)$
- $\forall l \in dom(\sigma)$, if $\sigma(l) = C\{ls\}$ then $\{ls\} \subseteq dom(\sigma)$

Proof. The proof is by definition of *validState*, and induction on the number of reductions since the initial memory and main-expression. In the base case, by definition of *validState*, the only l in the main-expression and memory is c, which is defined in the memory. In the inductive case, each reduction rule only introduces ls into the memory or main-expression that were either already there, or in the case of NEW/NEW TRUE, that are simultaneously added to the *dom* of the memory. As a simple corollary of this, we have that if $l \in dom(\sigma)$, then $rog(\sigma, l) \subseteq dom(\sigma)$, similarly with *mrog*.

Similarly, we show that once an l becomes un-*reachable*, it remains that way:

Lemma 3 (Lost Forever)**.**

If $validState(\sigma, \mathcal{E}[e])$, and $\sigma|e \rightarrow^* \sigma'|e'$, then $\forall l \in dom(\sigma)$, if not $reachable(\sigma, e, l)$, then not $reachable(\sigma', e', l)$.

Proof. The proof follows from induction on the number of reductions, and the fact that each reduction either does not introduce an l into the main expression or heap, or only introduces ls that were already *reachable* (in the case of UPDATE and ACCESS), or only introduces an $l \notin dom(\sigma)$ (in the case of NEW/NEW TRUE)

We can use our object capability discipline (described in Section 5) to prove that the `invariant()` method is deterministic and does not mutate existing memory:

Lemma 4 (Determinism)**.**

If $validState(\sigma, \mathcal{E}[\texttt{read}\, l\texttt{.invariant()}])$ and $\sigma|\texttt{read}\, l\texttt{.invariant()} \rightarrow^n \sigma'|e'$, for some $n \geq 0$, then:

- $\sigma \subseteq \sigma'$
- $\sigma|\texttt{read}\, l\texttt{.invariant()} \Rightarrow^n \sigma'|e'$

Proof. As the only reference in $\texttt{read}\, l\texttt{.invariant()}$ is $\texttt{read}\, l$, it follows from the definition of *mutatable*, that there is no l' with $mutatable(\sigma, \texttt{read}\, l\texttt{.invariant()}, l')$, thus by Mutatatable Update we have that for all $l \in dom(\sigma)$, $\sigma(l) = \sigma(l')$, i.e. $\sigma \subseteq \sigma'$

We show the second part by induction on n: if $n = 0$, then no reduction was performed, $e' = \texttt{read}\, l\texttt{.invariant()}$, and it trivially holds that $\sigma|\texttt{read}\, l\texttt{.invariant()} \Rightarrow^0 \sigma|\texttt{read}\, l\texttt{.invariant()}$. In the inductive case, we have some σ'' and e'' with $\sigma|\texttt{read}\, l\texttt{.invariant()} \rightarrow^{n-1} \sigma''|e'' \rightarrow \sigma'|e'$, and assume our inductive hypothesis that $\sigma|\texttt{read}\, l\texttt{.invariant()} \Rightarrow^{n-1} \sigma''|e''$. As c is not *mutatable* in $\texttt{read}\, l\texttt{.invariant()}$, by Mut Consistency, $\texttt{mut}\, c \notin e''$ and $\texttt{capsule}\, c \notin e''$. Since, by definition, there are never any other instances of `Cap`, it follows from Type Consistency that the reduction $\sigma''|e'' \rightarrow \sigma'|e'$ was not due to CALL/CALL MUTATOR reducing a call to a **mut** method of `Cap`. As all other methods are uniquely defined, the reduction must have been deterministic, i.e. $\sigma''|e'' \Rightarrow \sigma'|e'$, and so by the inductive hypothesis, we have $\sigma|\texttt{read}\, l\texttt{.invariant()} \Rightarrow^n \sigma''|e''$.

Rep Field Soundness

[Isaac: Re-read everything from this point!] Now we define and prove important properties about our novel **rep** fields. We first start with a few core auxiliary definitions. To simplify the notation, we define the *repFields* of an l to be the set of **rep** field names for l:

$repFields(\sigma, l) = \{f \text{ where } \mathrm{C}_l^\sigma.f = \texttt{rep}\, \_\, f\}$

We say that an l and f is *circular* if l is reachable from $l.f$:

$circular(\sigma, l, f)$ iff $l \in rog(\sigma, \sigma[l.f])$.

We say that an l is *repCircular* if any its **rep** fields are *circular*:

$\exists f \in repFields(\sigma, l)$ such that $circular(\sigma, l, f)$.

We say that an l and f is *confined* if $l.f$ is not *mutatable* without passing through l:
$\mathit{confined}(\sigma, l, f)$ iff not $\mathit{mutatable}(\sigma \setminus l, e, \sigma[l.f])$.
We say that an l is *repConfined* if each of its **rep** fields are *confined*:
$\forall f \in \mathit{repFields}(\sigma, l)$ we have $\mathit{confined}(\sigma, l, f)$.

We say that an l is *repMutating* if we are in a monitor for l which must have been introduced by CALL MUTATOR:
$\mathit{repMutating}(\sigma, e, l)$ iff $e = \mathcal{E}[\mathtt{M(}l\mathtt{;}\, e'\mathtt{;}\, \_\mathtt{)}]$, with $e' \neq \mathtt{mut}\, l$.

Finally we say that l is *headNotObservable* if we are in a monitor introduced for a call to a rep mutator, and l is not reachable from inside this monitor, except perhaps through a single **rep** field access:
$\mathit{headNotObservable}(\sigma, e, l)$ iff $e = \mathcal{E}_v[\mathtt{M(}l\mathtt{;}\, e'\mathtt{;}\, \_\mathtt{)}]$, and either:

- not $\mathit{reachable}(\sigma, e', l)$, or
- $e' = \mathcal{E}[\mathtt{mut}\, l\mathtt{.}f]$, $f \in \mathit{repFields}(\sigma, l)$, and not $\mathit{reachable}(\sigma, \mathcal{E}, l)$

Now we formally state the core properties of our **rep** fields (informally described in Section 3):
Theorem 2 (Rep Field Soundness)**.**
If $\mathit{validState}(\sigma, e)$ then $\forall l$ with $\mathit{reachable}(\sigma, e, l)$, we have:

- not $\mathit{repCircular}(\sigma, l, f)$, and
- one of the following holds:
  - $\mathit{repConfined}(\sigma, l)$ and not $\mathit{repMutating}(\sigma, e, l)$, or
  - $\mathit{headNotObservable}(\sigma, e, l)$.

That is, for every reachable object l: l is not reachable through any of its **rep** fields, and either we are in a rep mutator for l and l is not observable (except perhaps through a single **rep** field access), or we are not *repMutating* l, and each of ls **rep** fields are *confined*. *Proof.* By *validState* we have $c \mapsto \mathtt{Cap}\{\}|e_0 \rightarrow^m \sigma|e$, so we proceed by induction on m, the number of reductions. The base case when $m = 0$ is trivial, since **Cap** has no **rep** fields and the initial main expression e_0 cannot contain monitors.

In the inductive case, where $m > 0$, we have $\sigma_0|e_0 \rightarrow \ldots \rightarrow \sigma_{m-1}|e_{m-1} \rightarrow \sigma|e$, for some $\sigma_0, \ldots, \sigma_{m-1}$ and $e_0, \ldots, e_{m-1}$, where $\sigma_0|e_0$ is a valid initial memory and expression. Our inductive hypothesis is then that the conclusion of our theorem holds for each $\sigma_i|e_i$, for $i \in [0, m-1]$. We then proceed by cases on the reduction rule applied, and prove the theorems conclusion for $\sigma|e$:

1. (NEW/NEW TRUE) $\sigma'|\mathcal{E}_v[\mathtt{new}\ C\mathtt{(}\mu_1\, l_1, \ldots, \mu_n\, l_n\mathtt{)}] \rightarrow \sigma|\mathcal{E}_v[e']$, where $\sigma = \sigma', l_0 \mapsto C\{l_1, \ldots, l_n\}$; by Type Cosnsistency, we have $\mathtt{class}\ C\ \mathtt{implements}\ \_\ \mathtt{\{}\kappa_1\ \_\ f_1, \ldots, \kappa_n\ \_\ f_n\mathtt{;\}}$.
   (a) We have that l_0 is not *repCircular*: by No Dangling, we have that $\forall l' \in \mathit{dom}(\sigma')$, $\mathit{rog}(\sigma', l') \subseteq \mathit{dom}(\sigma')$. By our notational conventions for ",", it follows that $l_0 \notin \mathit{dom}(\sigma')$. Now consider each $i \in [1, n]$, since the pre-existing σ' was not modified, it follows that $\mathit{rog}(\sigma', l_i) = \mathit{rog}(\sigma, \sigma[l_0.f_i])$. By No Dangling we have that $\mathit{rog}(\sigma, \sigma[l_0.f_i]) \subseteq \mathit{dom}(\sigma)$, and so $l_0 \notin \mathit{rog}(\sigma, \sigma[l_0.f_i])$, thus each $l_0.f_i$ is not *circular*.
   (b) Ever *reachable* $l' \neq l_0$ is not *repCircular*: Since reduction didn't modify the fields of any pre-existing l', by the inductive hypothesis, we have that l' is still not *repCircular*.
   (c) The new l_0 is *repConfined* and not *repMutating*:
      - Consider each $i \in [1, n]$ with $\kappa_i = \mathtt{rep}$. By Type Consistency and Capsule Consistency, l_i was *encapsulated* and so $\mathit{rog}(\sigma', l_i)$ cannot be *mutatable* from $\mathcal{E}_v$. Thus, we don't have $\mathit{mutatable}(\sigma \setminus l_0, \mathcal{E}_v[e'], l_i)$, and so each of l_0s **rep** fields is *confined*.
      - We trivially have that l_0 is not *repMutating* since $l_0 \notin \mathit{dom}(\sigma')$, by No Dangling, there can't be any monitor expressions for it in $\mathcal{E}_v$.
   (d) Every *reachable* $l' \neq l_0$ that was *repConfined* and not *repMutating* still is:
      - Suppose we have made it so that for some $f' \in \mathit{repFields}(\sigma', l')$, $l'.f'$ is no longer *confined*. Since we didn't modify the *rog* of l' nor the *rog* of any other pre-existing l'', we must have that $\sigma'[l'.f']$ is now *mutatable* through $l_0.f_i$, for some $i \in [1, n]$. This requires that l_i is an initialiser for a **mut** or **rep** field, which by Type Consistency means that $\mu_i \leq \mathtt{mut}$. But then $\sigma'[l'.f']$ was already *mutatable* through $\mu_i\, l_i$, so $l'.f'$ can't have already been *confined*, a contradiction.

- We can't have caused l' to be *repMutating* since we haven't introduced any monitor expressions, nor modified any existing ones.

(e) Every *reachable* $l' \neq l_0$ is *headNotObservable*: by No Dangling, $l' \in dom(\sigma')$, so by Lost Forever, l' must have already been *reachable*. Thus, by the inductive hypothesis, l' must be *headNotObservable*, but we haven't removed any monitor expression or field accesses (because the arguments to the constructor are all fully reduced values), thus l' is still *headNotObservable*.

2. (ACCESS) $\sigma|\mathcal{E}_v[\mu\, l\texttt{.}f] \rightarrow \sigma|\mathcal{E}_v[\mu{::}\kappa\,\sigma[l.f]]$, where $\mathrm{C}^{\sigma}_{l}.f = \kappa\,\_\,f$:

(a) No *reachable* l' is *repCircular*: this holds by the inductive hypothesis and the fact that we haven't mutated memory.

(b) If l is *reachable* and it was *repConfined* and not *repMutating*, than it still is:

- If $\kappa \neq \texttt{rep}$, then we can't have broken *confined* for any $f' \in repFields(\sigma, l)$, since by definition of *repConfined*, $\sigma[l.f']$ can't have been *mutatable* through $\sigma[l.f]$.
- If $\kappa = \texttt{rep}$, since l' was not *repMutating*, this field access can't have been inside a rep mutator (or else we would be inside a monitor). As fields are instance private, we have $\mu \neq \texttt{mut}$, or else the field access would have come from a rep mutator.
  If $\mu = \texttt{capsule}$, then by Capsule Consistency and *repCircular*, l is not *reachable* from $\mathcal{E}_v[\mu{::}\kappa\,\sigma[l.f]]$, so it is irrelevant if l is no longer *repConfined*. Otherwise, since $\mu \notin \{Kwcapsule, \texttt{mut}\}$, we have $\mu{::}\kappa \not\leq \texttt{mut}$, so $l.f$ is still *confined*. By the above case for $\kappa \neq \texttt{rep}$, every other $f' \in repFields(\sigma, l)$ is *confined*.
- We can't have made l' *repMutating* since we have introduced any monitor expressions.

(c) If l was *repMutating* or not *repConfined*, than it is *headNotObservable*: by the inductive hypothesis, l was *headNotObservable* before this reduction, thus $\mathcal{E}_v = \mathcal{E}_v{}'[\texttt{M(}l\texttt{;}\,\mathcal{E}_v{}''\texttt{;}\,\_\texttt{)}]$. As l is clearly *reachable* in $\mathcal{E}_v{}''[\mu\, l\texttt{.}f]$, by definition of *headNotObservable* we must have that l is not *reachable* from $\mathcal{E}_v{}''$, and $\kappa = \texttt{rep}$. By *repCircular*, l is not in the *rog* of $\sigma[l.f]$, and so l is not *reachable* from $\mathcal{E}_v{}''[\mu{::}\kappa\,\sigma[l.f]]$, and so it is still *headNotObservable*.

(d) Every *reachable* $l' \neq l$ that was *repConfined* and not *repMutating*, still is:

- Since this reduction doesn't modify memory, and $\mu{::}\kappa \leq \texttt{mut}$ only if $\mu \leq \texttt{mut}$, we can't have made the *rog* of any `rep` field f' of l' *mutatable* without going through l', so *repConfined* is preserved.
- As in the NEW/NEW TRUE case above, we can't have made *repMutating* hold as we haven't introduced any monitor expressions.

(e) If l was *repMutating* or not *repConfined*, than it is *headNotObservable*: if $f \in repFields(\sigma, l)$, with $\mathcal{E}_v$ of form $\mathcal{E}_v{}'[\texttt{M(}l\texttt{;}\,\mathcal{E}_v{}''\texttt{;}\,\_\texttt{)}$ and l not *reachable* through $\mathcal{E}_v{}''$, then e is of form $\mathcal{E}_v{}'[\texttt{M(}l\texttt{;}\,\mathcal{E}_v{}''[\sigma[l.f]]\texttt{;}\,]\texttt{)}$. By the above, l is not *repCircular*, and so l is not *reachable* through $\sigma[l.f]$, thus l is not *reachable* through $\mathcal{E}_v{}''[\sigma[l.f]]$, and so l is *headNotObservable*. Otherwise, by the inductive hypothesis, l was *headNotObservable*, by definition of *headNotObservable*, since the above case does not hold, then $\mathcal{E}_v$ is of form $\mathcal{E}_v{}'[\texttt{M(}l\texttt{;}\,\mathcal{E}_v{}''\texttt{;}\,\_\texttt{)}]$ with l not *reachable* through $\mathcal{E}_v{}''[\mu\, l\texttt{.}f]$, thus by Lost Forever, l is not *reachable* through $\mathcal{E}_v{}''[\sigma[l.f]]$, thus l is still *headNotObservable*.

(f) Every *reachable* $l' \neq l$ that was *repMutating* or not *repConfined* is *headNotObservable*: as this reduction doesn't create any new objects, by No Dangling and Lost Forever, anything *reachable* was already *reachable*, thus by the inductive hypothesis, l' must have been *headNotObservable*. but we haven't removed any monitor expression or field accesses on l', thus l' must still be *headNotObservable*.

3. (UPDATE) $\sigma'|\mathcal{E}_v[\mu\, l\texttt{.}f\,\texttt{=}\,\mu'\, l'] \rightarrow \sigma'[l.f = l']|\mathcal{E}_v[\texttt{M(}l\texttt{;}\,\texttt{mut}\, l\texttt{;}\,\texttt{read}\, l\texttt{.invariant())}]$:

(a) For each $f' \in repFields(\sigma, l)$, $l.f'$ is still not *repCircular*:

- if $f' = f$, then by Type Consistency and Capsule Consistency, *encapsulated*$(\sigma', \mathcal{E}_v[\mu\, l\texttt{.}f = \square], l')$. Hence l is not *reachable* from l', and so after the update, $l.f'$ cannot be *circular*.
- otherwise, by the inductive hypothesis, $l.f'$ was not *repCircular*, so $l \notin rog(\sigma', \sigma'[l.f'])$, and so this update couldn't have change the *rog* of $l.f'$, and so it is still *repCircular*.

(b) For every *reachable* $l'' \neq l$, and $f' \in repFields(\sigma, l'')$, $l''.f'$ is still not *circular*:

- By the inductive hypothesis, $l''.f'$ was not *circular*.
- If l'' was *repConfined*, by Mut Update, $\mu \leq \texttt{mut}$. By *repConfined*, the *rog* of $\sigma'[l''.f']$ is not *mutatable*, except through a *field* access on l'', but this rule doesn't perform a field access, so since $l'' \neq l$, we must have that $l \notin \mathit{rog}(\sigma', \sigma'[l''.f'])$. Since we can't have modified the *rog* of $\sigma'[l''.f']$, $l''.f'$ is still not *circular*.
- Otherwise, by the inductive hypothesis, l'' was *headNotObservable*, and so $l'' \notin \mathit{rog}(\sigma', l')$, so we can't have added l'' to the *rog* of anything, thus $l''.f'$ is still not *circular*.

(c) Any *reachable* l'' that was *repConfined* and not *repMutating* still is:

- Suppose $l'' = l$ and $f \in \mathit{repFields}(\sigma', l)$, by Type Consistency and Capsule Consistency, l' is *encapsulated*, thus l' is not *mutatable* from $\mathcal{E}_v$, and l is not *reachable* from l'. Hence l' is still *encapsulated*, and so $l.f$ is still *confined*.
- Now consider any $f' \in \mathit{repFields}(\sigma', l'')$, with $l''.f' \neq l.f$; by the above, l is not *repCircular* and so $l \notin \mathit{rog}(\sigma', \sigma'[l''.f'])$. If f was a $\texttt{mut}$ or $\texttt{rep}$ field, by Type Consistency, $\mu' \leq \texttt{mut}$, so by *repConfined*, $l' \notin \mathit{rog}(\sigma', \sigma'[l''.f'])$; thus we can't have made $\mathit{rog}(\sigma', \sigma'[l''.f'])$ *mutatable* through $l.f$; so $\sigma'[l''.f']$ can't now be *mutatable* through $\texttt{mut}\, l$. By Mut Consitency, we couldn't have have made $\sigma'[l''.f']$ *mutatable* some other way, so l'' is still *repConfined*.
- As in the above cases for NEW/NEW TRUE, l'' is still not *repMutating* as we haven't introduced any monitor expressions.

(d) Every *reachable* l' that was *repMutating* or not *repConfined* is *headNotObservable*: similarly to the above case for ACCESS, as this reduction doesn't create any new objects, by by No Dangling and Lost Forever, anything *reachable* was already *reachable*, thus by the inductive hypothesis, l' must have been *headNotObservable*. but we haven't removed any monitor expression or field accesses, thus l' must still be *headNotObservable*.

4. (CALL/CALL MUTATOR) $\sigma|\mathcal{E}_v[\mu_0\, l_0\texttt{.}m\texttt{(}\mu_1\, l_1, \ldots, \mu_n\, l_n\texttt{)}] \rightarrow \sigma|\mathcal{E}_v[e]$

(a) Every *reachable* l' is not *repCircular*: as this rule doesn't mutate memory, by the inductive hypothesis, every *reachable* l' is still not *repCircular*.

(b) If l_0 was *repConfined* and not *repMutating*, it either still is, or is now *headNotObservable*:

- As we haven't modified memory, and by our well-formedness rules on method bodies, we haven't introduce any new ls into the main-expression, we must have that l_0 is still *repConfined*.
- Suppose the rule applied was CALL, by our well-formedness rules for method bodies, e doesn't contain a monitor. Moreover, by the CALL rule, e is not a rep mutator, if $e = \mathcal{E}[\mu'\, l_0\texttt{.}f]$, for some $f \in \mathit{repFields}(\sigma, l_0)$, we must have that m was not a $\texttt{mut}$ method. Since fields are instance-private, we must have $\mu' \not\leq \texttt{mut}$, and by our well-formedness rules on method bodies, e doesn't contain any monitors, thus we can't have caused l_0 to be *repMutating*.
- Otherwise, the rule applied was CALL MUTATOR, and m is a rep mutator, so $e = \texttt{M(}l_0\texttt{;}\, e'\texttt{;}\, \texttt{read}\, l_0\texttt{.invariant())}$. By our rules for rep mutators, m must be a $\texttt{mut}$ method with only $\texttt{imm}$ and $\texttt{capsule}$ parameters, thus by Type Consistency, $\mu_0 \leq \texttt{mut}$, and for each $i \in [1, n]$, $\mu_i \in \{\texttt{imm}, \texttt{capsule}\}$.
  By Imm Consistency and Capsule Consistency, l_0 can't be reachable from any l_i. Since rep mutators use $\texttt{this}$ only once, to access a $\texttt{rep}$ field, $e' = \mathcal{E}[\texttt{mut}\, l_0\texttt{.}f]$, for some $f \in \mathit{repFields}(\sigma, l_0)$. By our rules for rep mutators, $l_0 \notin \mathcal{E}$, and l_0 is not *reachable* from any l_i, and by our well-formedness rules for method bodies, there are no other ls in $\mathcal{E}$, thus we have that l_0 is not *reachable* from any $\mathcal{E}$, thus *headNotObservable* now holds for l.

(c) Every $l' \neq l_0$ that was *repConfined* and not *repMutating*, still is:

- By the above, since we haven't modified memory or introduced any new ls, l' must still be *repConfined*.
- Since $l' \neq l_0$ and fields are instance-private, we must have that there is no $\mu'\, l'\texttt{.}f \in e$. Moreover, by our well-formedness rules on method bodies, and the CALL/CALL MUTATOR rules, the only monitor that could be in e is a monitor on l_0, thus we can't have made l' *repMutating*.

(d) Every *reachable* l' that was *repMutating* or not *repConfined* is *headNotObservable*: as in the UPDATE case above, by the inductive hypothesis, l' must have been *headNotObservable*, as we haven't removed any monitor expressions or field accesses, l' is still *headNotObservable*.

5. (TRY ERROR) $\sigma|\mathcal{E}_v[\mathtt{try}^{\sigma'}\mathtt{\{}e\mathtt{\}}\ \mathtt{catch}\ \mathtt{\{}e'\mathtt{\}}] \rightarrow \sigma|\mathcal{E}_v[e']$, where $error(\sigma, e)$

(a) Every *reachable* l is not *repCircular*: as in the CALL/CALL MUTATOR case above, since this rule doesn't mutate memory, by the inductive hypothesis, every *reachable* l is still not *repCircular*.

(b) Every *reachable* l that was *repConfined* and not *repMutating* still is: by Mut Consistency and the fact that we haven't modified memory, l must still be *repConfined*. Since we haven't introduced any monitor expressions or field accesses, l cannot now be *repMutating*.

(c) If l is still *reachable*, and was *repMutating* or not *repConfined* then it is now *repConfined* and not *repMutating*:

- By definition of *error*, we have $e = \mathcal{E}_v{}'[\mathtt{M(}l\mathtt{;}\ v\mathtt{;}\ v'\mathtt{)}]$.
- If the monitor was introduced by NEW or UPDATE, then $v = \mathtt{mut}\, l$. And so *headNotObservable* can't have held for l since $l = l'$, and v was not the receiver of a field access. Thus by the inductive hypothesis, l must have been *repConfined* and not *repMutating*, a contradiction.
- By definition of *validState* and our well-formedness rules on method bodies, we must have that monitor must introduced by CALL MUTATOR, due to a call to a rep mutator on l.[33]
- From our reduction rules, it follows that we were previously in a state $\sigma_i|e_i$, where $i \in [1, m-1]$, e_i is of form $\mathcal{E}_v{}''[e'']$, and the next state was obtained by said application of the CALL MUTATOR rule to e''.
- Moreover, it follows that $\mathcal{E}_v{}'' = \mathcal{E}_v[\mathtt{try}^{\sigma'}\mathtt{\{}\mathcal{E}_v{}'\mathtt{\}}\ \mathtt{catch}\ \mathtt{\{}e'\mathtt{\}}]$, as no reduction rules modify the $\mathcal{E}_v$.
- We must not have had that l was *headNotObservable*, since e'' would contain l as the receiver of a method call. Thus, by our inductive hypothesis, in state i, l was *repConfined* and not *repMutating*.
- By Strong Exception Safety and No Dangling, every l' *reachable* from $\mathcal{E}_v[e']$ has not been mutated, i.e. $\sigma(l') = \sigma_i(l') = \sigma'(l)$.
- Since nothing *reachable* has been mutated, it follows that l is still *repConfined*.
- By *validState* and our well-formedness rules on method bodies, it follows that e' contains no monitor expressions.
- Moreover, since l was not *repMutating* in $\mathcal{E}_v[\mathtt{try}^{\sigma'}\mathtt{\{}\mathcal{E}_v{}'[e'']\mathtt{\}}\ \mathtt{catch}\ \mathtt{\{}e'\mathtt{\}}]$, and e' contains no monitors, l it follows that l is not *repMutating* in $\mathcal{E}_v[e']$.

(d) Every *reachable* $l'' \neq l$ that was *repMutating* or not *repConfined* is *headNotObservable*: as in the above case for UPDATE, by the inductive hypothesis, l'' must have been *headNotObservable*, as we haven't removed any monitor expressions on l'', or any field accesses, l'' is still *headNotObservable*.

6. (MONITOR EXIT) $\sigma|\mathcal{E}_v[\mathtt{M(}l\mathtt{;}\ \mu\, l'\mathtt{;}\ \_\mathtt{)}] \rightarrow \sigma|\mathcal{E}_v[\mu\, l']$

(a) Every *reachable* l'' is not *repCircular*: as in the CALL/CALL MUTATOR case above, since this rule doesn't mutate memory, by the inductive hypothesis, every *reachable* l'' is still not *repCircular*.

(b) Every *reachable* l'' that was *repConfined* and not *repMutating* still is: as in the TRY ERROR case above, by Mut Consistency and the fact that we haven't modified memory, l'' must still be *repConfined*. Since we haven't introduced any monitor expressions or field accesses, l'' cannot now be *repMutating*.

(c) If l is still *reachable*, and l was *repMutating* or not *repConfined* then it is now *repConfined* and not *repMutating*:

- If the monitor was introduced by NEW or UPDATE, then $\mu\, l' = \mathtt{mut}\, l$. And so *headNotObservable* can't have held for l since $l = l'$, and v was not the receiver of a field access. Thus by the inductive hypothesis, l must have been *repConfined* and not *repMutating*, a contradiction.

[33] A type-system will likely prevent this case from happening, as this would require calling a **mut** method on l, but l is *reachable* outside the **try** block. However, if the typesystem can prove that said **mut** method will not actually mutate l, this would not violate our requirements. Thus we still need to ensure that Rep Field Soundness holds in this case.

- By definition of *validState* and our well-formedness rules on method bodies, we must have that monitor must introduced by CALL MUTATOR, due to a call to a rep mutator on l.
- From our reduction rules, it follows that we were previously in a state $\sigma_i|e_i$, where $i \in [1, m-1]$, e_i is of form $\mathcal{E}_v{}'[e']$, and the next state was obtained by said application of the CALL MUTATOR rule to e'.
- Moreover, it follows that $\mathcal{E}_v{}' = \mathcal{E}_v$, as no reduction rules modify the $\mathcal{E}_v$.
- We must not have had that l was *headNotObservable*, since e' would contain l as the receiver of a method call. Thus, by our inductive hypothesis, in state i, l was *repConfined* and not *repMutating*.
- As with the above case for try error, by the inductive hypothesis, l must have been *headNotObservable*, and so the monitor must have been introduced by CALL MUTATOR.
- Thus, we were previously in a state $\sigma_i|e_i$ where $i \in [1, m-1]$, e_i is of form $\mathcal{E}_v[e']$, and the next state was obtained by said application of the CALL MUTATOR rule to e'.
- Thus, by the inductive hypothesis, in state i, l must have been *repConfined* and not *repMutating*.
- Because l was not *repMutating* in $\sigma_i|\mathcal{E}_v[e']$, and $\mu\, l'$ contains no monitors, l is not *repMutating* in $\mathcal{E}_v[\mu\, l']$.
- Since a rep mutator cannot have any **mut** parameters, by Type Consistency and Non Mutating, the body of the method can only modify things *mutatable* through l, or a **capsule** parameter.
- By Type Consistency, and Capsule Consistency, every capsule parameter is *encapsulated*, and so anything mutated through such a parameter must have been un*reachable* outside the call.
- Thus, forall $l' \in dom(\sigma_i)$, if $reachable(\sigma_i, \mathcal{E}_v, l')$ and $l' \notin mrog(\sigma_i, l)$, then $\sigma(l) = \sigma_i(l)$.
- If $\mu =$ **capsule**, then by Capsule Consistency, not part of the *mrog* of any **rep** field of l can be in the *rog* of l' (or else l would have to be un*reachable*), so we can't have made such a field *mutatable*.
- If $\mu \neq$ **capsule**, then since a rep mutator cannot have a **mut** return type, and our CALL MUTATOR rule wraps the method body in a **as** expression, we must have that $\mu \not\leq$ **mut**. Thus $\mu \in \{$**read**, **imm**$\}$, and so by l is not *mutatable* through $\mu\, l'$.
- As l was *repConfined* in $\sigma_i|\mathcal{E}_v[e']$, and we haven't modified anything *reachable* through $\sigma \setminus l$, nor have we made the *rog* of l *mutatable* through $\mu\, l'$, it follows that l is also *repConfined* in $\mathcal{E}_v[\mu\, l']$.

(d) Every *reachable* $l'' \neq l$ that was *repMutating* or not *repConfined* is *headNotObservable*: as in the UPDATE case above, by the inductive hypothesis, l'' must have been *headNotObservable*, as we haven't removed any monitor expressions on l'', or any field accesses, l'' is still *headNotObservable*.

7. (AS, TRY ENTER, and TRY OK) these are trivial, since as in the above cases:
   (a) Every *reachable* l is not *repCircular*: as in the CALL/CALL MUTATOR case above, since these rules don't mutate memory, by the inductive hypothesis, every *reachable* l is still not *repCircular*.
   (b) Every *reachable* l that was *repConfined* and not *repMutating* still is: as in the TRY ERROR case above, by Mut Consistency and the fact that these rules don't modified memory, l must still be *repConfined*. Since this rules don't introduce any monitor expressions or field accesses, l cannot now be *repMutating*.
   (c) Every *reachable* l that was *repMutating* or not *repConfined* is *headNotObservable*: as in the UPDATE case above, by the inductive hypothesis, l must have been *headNotObservable*, as these rules don't remove any monitor expressions or field accesses, l'' is still *headNotObservable*.

Stronger Soundness

It is hard to prove Soundness directly, so we first define a stronger property, called Stronger Soundness.

We say that an object is *monitored* if execution is currently inside of a monitor for that object, and the monitored expression e_1 does not contain a reference to l as a *proper* sub-expression:

$monitored(e, l)$ iff $e = \mathcal{E}_v[\mathtt{M}(l; e'; \_)]$ and $l \in e'$ only if $e' = \_\, l$.

A monitored object is associated with an expression that cannot observe it, but may reference its internal representation directly. In this way, we can safely modify its representation before checking its invariant. The idea is that at the start the object will be valid and e' will reference l; but during reduction, l will be used to modify the object, but not observe it; only after that moment, the object may become invalid.

Stronger Soundness says that starting from a well-typed and well-formed $\sigma_0|e_0$, and performing any number of reductions, every *reachable* object is either *valid* or *monitored*:

Theorem 3 (Stronger Soundness)**.**

If $validState(\sigma, e)$ then $\forall l$, if $reachable(\sigma, e, l)$, then $valid(\sigma, l)$ or $monitored(e, l)$.

Proof. As with the above proof of Rep Field Soundness, we will prove this inductively on the number of reductions. By *validState* we have $c \mapsto \texttt{Cap}\{\}|e_0 \rightarrow^m \sigma|e$, The base case when $m = 0$ is trivial, from our requirements for the `Cap` class, $\sigma|\texttt{read}\, c\texttt{.invariant()} \rightarrow \sigma|\texttt{new True()} \rightarrow \sigma, l \mapsto \texttt{True}\{\}|l$, for some l, thus by Determinism, it follows that c (the only thing in the memory) is *valid*.

In the inductive case, where $m > 0$, we have $\sigma_0|e_0 \rightarrow .. \rightarrow \sigma_{m-1}|e_{m-1} \rightarrow \sigma|e$, for some $\sigma_0, .., \sigma_{m-1}$ and $e_0, .., e_{m-1}$, where $\sigma_0|e_0$ is a valid initial memory and expression. Our inductive hypothesis is then that that everything *reachable* from the previous *validState* is *valid* or *monitored*. We then proceed by cases on the reduction rule that gets us to $\sigma|e$:

1. (NEW) $\sigma'|\mathcal{E}_v[\texttt{new}\ C\texttt{(}\_l_1, .., \_l_n\texttt{)}] \rightarrow \sigma', l_0 \mapsto C\{l_1, .., l_n\}|\mathcal{E}_v[\texttt{M(}l_0\texttt{;}\ \texttt{mut}\ l_0\texttt{;}\ \texttt{read}\ l_0\texttt{.invariant())}]$:
   - Clearly the newly created object, l, is *monitored*.
   - This rule does not modify pre-existing memory, introduce pre-existing ls into the main expression, nor remove monitors on other ls, by the inductive hypothesis, every $l' \neq l_0$ is still *valid* (due to Determinism), or *monitored*.
2. (NEW TRUE) $\sigma'|\mathcal{E}_v[\texttt{new True()}] \rightarrow \sigma', l_0 \mapsto \texttt{True}\{\}|\mathcal{E}_v[\texttt{mut}\ l_0]$:
   - The `True` class is required to have an invariant of `new True()`, so as with c in the base case above, we have that l_0 is *valid*.
   - As in the above case for NEW, since we didn't modify pre-existing memory, introduce pre-existing ls into the main expression, nor remove monitors, by the inductive hypothesis, every $l' \neq l_0$ is still *valid* or *monitored*.
3. (UPDATE) $\sigma'|\mathcal{E}_v[\mu\, l\texttt{.}f\texttt{=}v] \rightarrow \sigma|\mathcal{E}_v[e']$, where $e' = \texttt{M(}l\texttt{;}\ \texttt{mut}\ l\texttt{;}\ \texttt{read}\ l\texttt{.invariant())}$:
   - Clearly l is now *monitored*.
   - Consider any other l', where $l \in rog(\sigma', l')$ and l' was *valid*; now suppose we just made l' in*valid*. By our well-formedness criteria, `invariant()` can only accesses `imm` and `rep` fields, thus by Non Mutating, and Determinism, we must have that l was in the *rog* of $\sigma'[l'.f']$, for some $f' \in repFields(\sigma', l')$.
     Since $l \neq l'$, l' can't have been *repConfined*. Thus, by Rep Field Soundness, l' was *headNotObservable*, and so $\mathcal{E}_v[\mu\, l\texttt{.}f\texttt{=}v]$ is of form $\mathcal{E}_v'[\texttt{M(}l'\texttt{;}\ e''\texttt{;}\ e'''\texttt{)}]$:
     - As the *rog* of l' has just been mutated, and since e''' must have started off as $\texttt{read}\ l'''\texttt{.invariant()}$, if follows from Determinism, that we cannot currently be inside e'''.
     - Thus, $\mathcal{E}_v = \mathcal{E}_v'[\texttt{M(}l'\texttt{;}\ \mathcal{E}_v''\texttt{;}\ e'''\texttt{)}]$, where $\mathcal{E}_v''[\mu\, l\texttt{.}f\texttt{=}v] = e''$.
     - Suppose that l' was not *reachable* in e'', then clearly $l' \notin e''$, since $l' \neq l$, it follows that $l' \notin \mathcal{E}_v''[e']$, and so l' is *monitored*.
     - Otherwise, by definition of *headNotObservable*, we have that $e'' = \mathcal{E}[\texttt{mut}\ l'\texttt{.}f'']$ for some $f'' \in repFields(\sigma', l')$, and where l' is not *reachable* in $\mathcal{E}$.
     - By the proof for the TRY ERROR case of Rep Field Soundness, the monitor must have come from a call to a rep mutator, in a state where l' was *repConfined*. Thus, we were previously in a state $\sigma_i|e_i$, for some $i \in [0, m-1]$, immediately after a CALL MUTATOR; moreover, e_i is of form $\mathcal{E}_v'[\texttt{M(}l'\texttt{;}\ e'_i\texttt{;}\ \_\texttt{)}]$, immediately after a CALL MUTATOR, where e'_i is of form $\mathcal{E}'[\texttt{mut}\ l'\texttt{.}f''']$.
     - By Rep Field Soundness, l' is not *reachable* through $\sigma'[l'.f''']$,. By the proof for the CALL/CALL MUTATOR case of Rep Field Soundness, we have that l' is not *reachable* through $\mathcal{E}'$. Thus, by Lost Forever, once $\texttt{mut}\ l'\texttt{.}f'''$ has been reduced, l' must be un*reachable*, and it follows that $\texttt{mut}\ l'\texttt{.}f'' = \texttt{mut}\ l'\texttt{.}f'''$

  - By Mut Update, l is *mutatable* in the current state, thus by Mut Consistency, we have that it was also *mutatable* when CALL MUTATOR rule was applied. But we have that l' was *repConfined*, so since $l \in \mathit{rog}(\sigma', \sigma'[l'.f'])$, we have that l can only be *mutatable* through l'.
  - By Lost Forever, the only way we could have obtain a reference to l was by reducing $\texttt{mut}\, l'\texttt{.}f''$, but we haven't done that yet, a contradiction.
- Every other *valid* l', where $l \notin \mathit{rog}(\sigma', l')$ is still *valid* by Determinism.
- As in the above case, since we don't remove any monitors, any other l' that was *monitored*, is still *monitored*.

4. (TRY ERROR) $\sigma|\mathcal{E}_v[\texttt{try}^{\sigma'}\texttt{\{}e\texttt{\}}\ \texttt{catch}\ \texttt{\{}e'\texttt{\}}] \to \sigma|\mathcal{E}_v[e']$, where $\mathit{error}(\sigma, e) = \mathcal{E}_v{}'[\texttt{M(}l\texttt{;}\ \_\texttt{;}\ \_\texttt{)}]$:
   - As with the case for TRY ERROR in the proof of Rep Field Soundnes, we were previously in a state $\sigma_i|e_i$, where $e_i = \mathcal{E}_v[\texttt{try}^{\sigma'}\texttt{\{\_\}}\ \texttt{catch}\ \texttt{\{\_\}}]$, and $\sigma_i = \sigma'$.
   - By definition of *error*, we have that l is not *valid* in σ. [Isaac: Because monitors always start of ass invariant calls]
   - Suppose l is still *reachable* in $\sigma|\mathcal{E}_v[e']$, by Strong Exception Safety, we have $l \in \mathit{dom}(\sigma')$. Thus by the inductive hypothesis, we have that l was *valid* or *monitored* in the state $\sigma'|e_i$.
   - If l was *monitored*, then by *validState* and our well-formedness rules on method bodies, said monitor must have been introduced by the NEW, UPDATE, or CALL MUTATOR reduction rules.
   - The NEW and UPDATE rules monitor a value, which cannot reduce to a $\texttt{try}\text{—}\texttt{catch}$, so the monitor must have been introduced by CALL MUTATOR.[Isaac: No new bullet point]
   - But by our well-formedness rules on rep mutators, the body of the called method cannot mention l except to read a field, as shown in the case for UPDATE above, l will be un*reachable* once the field access has been reduced, which by Lost Forever is a contradiction, as l is *reachable* through e.
   - Thus, l can't have been *monitored* in $\sigma'|e_i$, so it must have been *valid*.
   - Also by Strong Exception Safety, we have that nothing reachable from l could have been modified, that is $\forall l' \in \mathit{rog}(\sigma', l)$, we have $\sigma'(l') = \sigma(l')$. By Lost Forever, and our reduction rules, any memory location not *reachable* from a call $\texttt{read}\, l\texttt{.invariant()}$ cannot affect its reduction.
   - Thus, by Determinism, and the fact that l was valid in σ', we have that l is still *valid*, a contradiction.
   - Thus, l cannot be *reachable*, so the fact that it is in*valid* is irrelevant.
   - As in the above case for NEW, since we didn't modify any memory, or remove any other monitors, by the inductive hypothesis every $l' \neq l$ is still *valid* or *monitored*.

5. (MONITOR EXIT) $\sigma|\mathcal{E}_v[\texttt{M(}l\texttt{;}\ v\texttt{;}\ \texttt{imm}\, l'\texttt{)}] \to \sigma|\mathcal{E}_v[v]$, where $\mathrm{C}^{\sigma}_{l'} = \texttt{True}$:
   - By *validState* and our well-formedness requirements on method bodies, the monitor expression must have been introduced by UPDATE, CALL MUTATOR, or NEW. In each case the third expression started off as $\texttt{read}\, l\texttt{.invariant()}$, and it has now (eventually) been reduced to $\texttt{imm}\, l'$, thus by Determinism l is *valid*.
   - As in the above case for NEW, since we didn't modify any memory, or remove any other monitors, by the inductive hypothesis every *reachable* $l' \neq l$ is still *valid* or *monitored*.

6. (ACCESS, CALL/CALL MUTATOR, AS, TRY ENTER, and TRY OK) these are trivial:
   - As in the above case for NEW, since these rules don't modify memory or remove monitors, by the inductive hypothesis, every *reachable* l is still *valid* or *monitored*.

Proof of Soundness

[Isaac: I haven't checked the proofs here for correctness yet, I need a break...] First we need to prove that an object is not reachable from one of its `imm` fields; if it were, `invariant()` could access such a field and observe a potentially broken object:

Lemma 5 (Imm Not Circular)**.**

If $\mathit{validState}(\sigma, e)$, $\forall f, l$, if $\mathit{reachable}(\sigma, e, l)$, $\mathrm{C}^{\sigma}_{l}.f = \mathtt{imm}\,\_\,f$, then $l \notin \mathit{rog}(\sigma, \sigma[l.f])$.

Proof. The proof is by induction; obviously the property holds in the initial $\sigma|e$, since $\sigma = c \mapsto \mathtt{Cap}\{\}$. Now suppose it holds in a $\mathit{validState}(\sigma', e')$ where $\sigma'|e' \rightarrow \sigma|e$:

1. Consider any pre-existing *reachable* l and f with $\mathrm{C}^{\sigma'}_{l}.f = \mathtt{imm}\,\_\,f$, by Imm Consistency and Non Mutating, the only way $\mathit{rog}(\sigma, \sigma[l.f])$ could have changed is if $e' = \mathcal{E}_v[\mu\, l\mathtt{.}f = \mu'\, l']$, where $\mu \leq \mathtt{mut}$, i.e. we just applied the UPDATE rule. By Type Consistency, $\mu' \leq \mathtt{imm}$, so by Imm Consistency, $l \notin \mathit{rog}(\sigma, l')$. Since $l' = \sigma[l.f]$, we now have $l \notin \mathit{rog}(\sigma, \sigma[l.f])$.
2. The only rules that make an l *reachable* are NEW/NEW TRUE. So consider $e = \mathcal{E}_v[\mathtt{new}\ C\mathtt{(}\_l_1, .., \_l_n\mathtt{)}]$, and each i with $C.i = \mathtt{imm}\,\_\,f$. But each of $l_1, .., l_n$ existed in the previous state and $l \notin \mathit{dom}(\sigma')$; so by *validState* and our reduction rules, $l \notin \mathit{rog}(\sigma', l_i) = \mathit{rog}(\sigma, \sigma[l.f])$.

Note that the above only applies to `imm` *fields*: `imm` *references* to cyclic objects can be created by promoting a `mut` reference, however the cycle must pass through a *field* declared as `read` or `mut`, but such fields cannot be referenced in the invariant method.

We can now finally prove the soundness of our invariant protocol:

Theorem 4 (Soundness)**.** If $\mathit{validState}(\sigma, \mathcal{E}_r[\_l])$, then either $\mathit{valid}(\sigma, l)$ or $\mathit{trusted}(\mathcal{E}_r, l)$.

Proof. Suppose $\mathit{validState}(\sigma, e)$, and $e = \mathcal{E}_r[\_l]$. Suppose l is not *valid*; since l is *reachable*, by Stronger Soundness, $\mathit{monitored}(e, l)$, $e = \mathcal{E}[\mathtt{M(}l\mathtt{;}\, e_1\mathtt{;}\, e_2\mathtt{)}]$, and either:

- $\mathcal{E}_r = \mathcal{E}[\mathtt{M(}l\mathtt{;}\, \mathcal{E}'\mathtt{;}\, e_2\mathtt{)}]$, that is l was found inside of e_1, but by definition of $\mathcal{E}_r$, we can't have $e_1 = \mu\, l$, this contradicts the definition of *monitored*, or
- $\mathcal{E}_r = \mathcal{E}[\mathtt{M(}l\mathtt{;}\, e_1\mathtt{;}\, \mathcal{E}'\mathtt{)}]$, and thus l was found inside e_2. By our reduction rules, all monitor expressions start with $e_2 = \mathtt{read}\, l\mathtt{.invariant()}$; if this has yet to be reduced, then $\mathcal{E}' = \mathcal{E}''[\square\mathtt{.invariant()}]$, thus $\mathit{trusted}(\mathcal{E}_r, l)$. By our well-formedness rules for `invariant()`, the next reduction step will be a CALL, e_2 will only contain l as the receiver of a field access; so if we just performed said CALL, $\mathcal{E}' = \mathcal{E}''[\square\mathtt{.}f]$: hence $\mathit{trusted}(\mathcal{E}_r, l)$. Otherwise, by Imm Not Circular, Rep Field Soundness, and *repCircular*, no further reductions of e_2 could have introduced an occurrence of l, so we must have that l was introduced by the CALL to `invariant()`, and so $\mathit{trusted}(\mathcal{E}_r, l)$.

Thus either l is *valid* or $\mathit{trusted}(\mathcal{E}_r, l)$.

Appendix B. Typesystem and Proof of Requirements

Appendix C. Safe Parallelism in 42, without destructive reads

This section discuss the relation between our work and the 42 support for safe unobservable parallelism.

From its inception, the work on 42 tried to build a system supporting flexible, elegant and soundly unobservable parallelism without the need of relying on destructive reads. 42 uses the same kind of reference and object capabilities used by Pony and Gordon, but while they allow for *true* isolated fields with destructive reads, 42 avoids them. That is: in Pony/Gordon we can easy define a class boxing a capsule references as follows

```
class Box{ //pseudocode for clarity
  iso Foo foo;//isolated field
  Box(iso Foo foo){this.foo=foo;}//initialized with an iso reference
  mut method iso Foo getFoo(){return this.foo;}//destructive read here
  }//the getter mutates the box (destructive read) and returns the stored iso reference
//usage
iso Foo myFoo = ..
mut Box box = new Box(myFoo);//the type system ensures this is the only usage of 'myFoo'
//box is now a mut object and can be freely passed around and aliased
iso Foo foo1 = box.getFoo();//this foo1 can now be used for parallelism
iso Foo foo2 = box.getFoo();//either foo2==null or exception is thrown here
```

42 does not support destructive reads. Thus, there is no way to declare method `getFoo()`. This also means that after putting encapsulated data into a field of any kind, it is not possible to extract the data back as encapsulated. Indeed, if 42 where to somehow allow the initialization of `foo1` we would then have two ways to reach the 'encapsulated' data, thus breaking the encapsulation guarantee. Thus, the 42 research has explored various ways to access those fields, that are initialized with encapsulated data but can not soundly release the data as capsule. From a research perspective, it was interesting to discover many different access patterns that allowed to preserve some encapsulation properties but not others. On the other side, it is worth of notice that exploring those different kind of encapsulated data does not impact the way capsule references are threated when passed around as method parameters or saved in local variables, nor their parallemism properties.

Indeed, in 42, as in Pony or Gordon, we can make fork-join where the parallel branches only use `capsule` variables. The difference is that in 42 those variables will not be able to come from reading encapsulated fields of `mut` objects. Note that this is because, as a research question, the 42 developers are trying to understand how far they can go without resorting to destructive reads. They could easely add a primitive datatype working as a *consumable* mutable box storing a true capsule. Such a primitive data type would behave exactly like the `Box` type above. Then, a 42 user could chose to use those boxes to recover all the parallelism expressive power (and risks) of destructive reads. Doing so would however make it much harder to claim that 42 supports expressive automatic parallelism without the need of destructive reads, since destructive reads would now be avaible on demand.

Parallel patterns in 42

The current version of 42 relies on the annotation @`Cache.ForkJoin` and the `Data` decorator to activate varius forms of (unobservable) automatic parallelism. We show those forms below.

Non-Mutable computation

This is the simplest 42 parallel pattern. Consider the following code in 42:

```
Example = Data:{
  @Cache.ForkJoin class method
  capsule D foo(capsule A a, capsule B b, imm C c, read D d) = (
    mut A a0=a.op(d)
    mut B b0=b.op(c)
    mut C c0=c.op(d)
    a0.and(b0).and(c0)
    )
  }
```

The initialization expressions for `a0`, `b0`, and `c0` are run in parallel, and the final expression is run only when all of the initialization expressions are completed. The method itself can take any kind of parameters, and they can all be used in the final expression, but the initialization expressions need to fit one of the recognized safe parallel patterns. In *non-mutable computation* only `read`,`capsule` and `imm` parameters can be used in the initialization expressions. The name *non-mutable computation* comes from the fact that, even if the capsules can indeed be mutated, nothing that is visible outside of the fork-join can be mutated while the forkjoin is open; thus parallelism is unobservable.

More in general, @`Cache.ForkJoin` works only on methods whose body is exactly a round parenthesis block, with some local variable initialization expressions and a conclusive expression. Thus, fork-join methods will follow this specific syntactic pattern: [Isaac: TODO!!! The following code won't compile anymore!] where the varius expressions $e_0...e_k$ are executed in parallel, and the final expression e is executed after all of $e_0...e_k$. Different forms of parallelism impose different requirements on the free variables that expressions $e_0...e_k$ can use.

Some readers find suprising that in the *non-mutable computation* pattern `read` references can be freely used, since there can be `mut` references to those same objects. However, those `mut` references are all unreachable from inside our fork-join. This is because the whole mutROG of all the `capsule` rerefences is encapsulated and other `mut` references are not allowed. How the mutROG of `capsule` fields can be shared is not important here because in 42 there is no way to go from a `capsule` field back to a `capsule` reference.

This form of parallelism is the only one proposed by Gordon; it is very expressive in their setting with destructive reads, but it is quite limited in 42. However 42 offers other forms of parallelism, as shown below.

Single-Mutable computation

In this pattern, a single initialization expression can use any kind of parameter, while the other ones can not use `mut`, `lent` (a variation of `mut` present in 42) or `read` parameters. This pattern allows the single initialization expression that can use `mut` to recursively explore a complex mutable data structure and to update immutable elements arbitrarily nested inside of it. Consider for example this code computing in parallel new immutable string values for all of the entries in a mutable list:

```
UpdateList = Data:{
  class method S map(S that) = that++that//could be any user defined code
  class method Void of(mut S.List that) = this.of(current=0I,data=that)
  class method Void of(I current, mut S.List data) = (
    if current<data.size()
      this.of(current=current,elem=data.val(current),data=data)
    )
  @Cache.ForkJoin class method Void of(I current, S elem, mut S.List data) =(
    S newElem=this.map(elem)
    this.of(current=current+1I,data=data)
    data.set(current,val=newElem)
    )
  }
//usage
mut S.List data = S.List[S"a";S"b";S"c";S"d";S"e";]
UpdateList.of(data)
Debug(data)//["aa"; "bb"; "cc"; "dd"; "ee"]
```

As you can see, we do not need to ever copy the whole list. We can update the elements in place one by one. If the operation `map(that)` is complex enough, running it in parallel could be beneficial. You can equivalently read this code either as a fork join or as sending the computation `this.map(elem)` to be run on a separate worker, while the computation `this.of(current=current+1I,data=data)` is executed on the current thread. Indeed, to implement a fork join it is always more effienct to run the last branch in the current thread. As you can see, it is trivial to adapt that code to explore other kinds of collections, like for example a binary tree. The visit of the tree will be performed recursivelly but sequentially in the current thread, and workers will be spowned at all recursive layers and their results will be composed at the end of the recursion.

Those two forms of parallelism where already possible on the 42 model before our work on invariants and our new form of `capsule` fields. We think that it is pretty impressive that this kind of parallelism can be obtained without destructive reads. Building on top of our new form of `capsule` fields and on the concept of *capsule mutators*, a new form of parallel fork-join computation was recently added: *This-Mutable computation*.

This-Mutable computation

In this pattern, the `this` variable is considered specially. The method must be declared `mut`, and the initialization expressions can not use `mut`, `lent` or `read` parameters. However, the `mut` parameter `this` can be used to directly call capsule mutator methods (marked by `@Cache.Clear` in the 42 syntax). Since a capsule mutator can mutate the ROG of a capsule field, and the mutROG from different capsule fields is disjoint, different initialization expressions must use capsule mutators updating different `capsule` fields. In this way, 42 can express parallel computation processing arbitrary complex mutable objects inside well encapsulated data structures. Consider the following example, where instances of `Foo` could be arbitrarily complex; containing complex (possibly circular) graphs of mutable objects.

```
Foo=Data:{.. /*mut method Void op(I a, S b)*/ ..}

Tree={interface [HasToS]    mut method Void op(I a, S b) }
```

```
Node = Data:{[Tree]
  capsule Tree left, capsule Tree right
  @Cache.ForkJoin mut method Void op(I a, S b) = (
    unused1=this.leftOp(a=a,b=b)
    unused2=this.rightOp(a=a,b=b)
    void
    )
    @Cache.Clear class method Void leftOp(mut Tree left,I a, S b) = left.op(a=a,b=b)
    @Cache.Clear class method Void rightOp(mut Tree right,I a, S b) = right.op(a=a,b=b)
    }
Leaf = Data:{[Tree]
  capsule Foo label
  @Cache.Clear class method Void op(mut Foo label,I a, S b) = label.op(a=a,b=b)
  }
//usage
mut Tree top = Node(
  left=Node(
    left=Leaf(label=..)
    right=Leaf(label=..)
    )
  right=Node(
    left=Leaf(label=..)
    right=Leaf(label=..)
    )
  )
top.op(a=15I b=S"hello")
```

This pattern relies on the fact that using **capsule** fields we can define arbitrary complex data structures composed of disjointed mutable object graphs. Note that **read** aliases to parts of the data structure can be visible outside. This is safe since we can not access them when the forkjoin is open. The declarations can not use **read** parameters.

Non fork-join parallelism in 42

42 also supports eager caching using the annotation `@Cache.Eager`. This form of parallelism is limited to start only from fully immutable data. This annotation can be used only on no args methods of objects that are born immutable. Parallel workers are used to eagerly compute the result of those methods and cache the result in the object. This form of parallelism allows to express computation in a very declarative style, but it does not interact with our capsule fields or capsule mutators, so an in-dept discussion of `@Cache.Eager` is out of scope.

Parallelism in older versions of the 42 type system

42 has been undergoing many changes across the years. The earlier version of the 42 type system was based on [??], where you could see many more modifiers, including **fresh** and **baloon**. In that version **fresh** is similar to the current 42 **capsule** references and **baloon** is similar to one of the various kinds of available 42 **capsule** fields. That work was then summarized in a short 6 pages paper [??], that refers to both **fresh** and **baloon** as **baloon**; using the different context (reference, local variable, field) to make change the meaning of the single keword **baloon**, to cover both roles of **fresh** and **baloon** in [..]. Even in those earlier works there was no way to recover a **fresh** from a **baloon**, and a **baloon** was basically a kind of encapsulated reference that allowed other restricted kinds of references (**external** and **external readonly**) to point inside of it. Looking back to those earlier work it is clear to us that the current 42 type system is much more minimal and elegant. Those works suggested interesting forms of parallelism where the type system could cooperate with a few efficent run time pointer equality checks to decide what to run in parallel. Such a direction has not been explored further and it is not currently present in modern versions of 42.