Using Capabilities for Strict Runtime Invariant Checking

ISAAC OSCAR GARIANO, Victoria University of Wellington, New Zealand
MARCO SERVETTO, Victoria University of Wellington, New Zealand
ALEX POTANIN, Victoria University of Wellington, New Zealand

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 much lower annotation burden compared to Spec#, and performs orders of magnitude less runtime invariant checks compared to the widely used 'visible state semantics' protocols of D and Eiffel.

CCS Concepts: • **Theory of computation → Invariants**; **Program verification**; • **Software and its engineering → Object oriented languages**; *Dynamic analysis*; *Constraints.*

Additional Key Words and Phrases: reference capabilities, object capabilities, runtime verification, class invariants

ACM Reference Format:
Isaac Oscar Gariano, Marco Servetto, and Alex Potanin. 2020. Using Capabilities for Strict Runtime Invariant Checking. *ACM Trans. Program. Lang. Syst.* 1, 1, Article 1 (January 2020), 45 pages.

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 protocols are the *visible state semantics* [Meyer 1988] and the *Pack-Unpack/Boogie methodology* [Barnett et al. 2004a]. In the 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

[1]We do this (as in Dafny [Leino 2012]) to minimise the special treatment of invariants, whereas other approaches often treat invariants as a special annotation with its own syntax.

Authors' addresses: Isaac Oscar Gariano, School of Engineering and Computer Science, Victoria University of Wellington, Wellington, New Zealand, isaac@ecs.vuw.ac.nz; Marco Servetto, School of Engineering and Computer Science, Victoria University of Wellington, Wellington, New Zealand, marco.servetto@ecs.vuw.ac.nz; Alex Potanin, School of Engineering and Computer Science, Victoria University of Wellington, Wellington, New Zealand, alex@ecs.vuw.ac.nz.

2020. 0164-0925/2020/1-ART1 $15.00
https://doi.org/

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 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` cannot 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 ≥ `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 since `this` is not in the ROG of `this.box`, `min`, or `max`. The call to `BoxRange.set(min,max)` works in a context where the enclosing **Range** object is unreachable, and thus not involved in execution. 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

[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, but with a different goal. While in very specific situations the

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.

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. [Gopinathan and Rajamani 2008] On the other hand, approaches such as *pack/unpack* [Barnett et al. 2004a] encumber the type system and the syntax with features whose only purpose is to distinguish objects with broken invariants, while (at least in the case of Spec#) still not soundly supporting I/O and exceptions. The core insight behind our work is that we can design a general purpose language as presented in this paper that does not require any *invariant-specific* language mechanisms but instead we show how a clever use of capabilities and small number of decorator-like design patterns is sufficient to capture invariant guarantees that are comparable to the state of the art object-oriented verification systems.

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 [Giannini et al. 2016; Lagorio and Servetto 2011; Servetto et al. 2013; Servetto and Zucca 2015], Pony [Clebsch et al. 2015, 2017], and the language of Gordon *et al.* [Gordon et al. 2012] use RCs (Reference Capabilities)[4] and OCs (Object Capabilities) to statically ensure deterministic parallelism and the absence of dataraces. 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.

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. [Bolz et al. 2011]

[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.

Reference Capabilities

RCs, as used in this paper, are a type system feature that allows reasoning about aliasing and mutation. Recently a new design for them has emerged that radically improves their usability; three different research languages are being independently developed relying on this new design: the language of Gordon *et al.*, 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 [Clebsch et al. 2015, 2017; Gordon et al. 2012; Servetto et al. 2013].

Reference capabilities are a well known mechanism [Birka and Ernst 2004; Clebsch et al. 2015; Giannini et al. 2016; Gordon et al. 2012; Östlund et al. 2008; Tschantz and Ernst 2005] that allow statically reasoning about the mutability and aliasing properties of objects. Here we refer to the interpretation of [Gordon et al. 2012], that introduced the concept of recovery/promotion. This concept is the basis for L42, Pony, and Gordon *et al.*'s type systems [Clebsch et al. 2015, 2017; Gordon et al. 2012; Servetto et al. 2013; Servetto and Zucca 2015]. With slightly different names and semantics, those languages all support the following RCs for object references:

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

In L42, a `capsule` local variable always holds a `capsule` reference: this is ensured by allowing them to be used only once (similar to linear and affine types [Boyland 2001]). Pony and Gordon *et al.* follow a more complicated approach: `capsule` variables can be accessed multiple times, however the result will not be a `capsule` reference and can only be used in limited ways. Pony and Gordon also provide destructive reads, where the variable's old value is returned as `capsule`. Later on, we discuss `capsule` fields, which behave differently to `capsule` local variables.

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]

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
```

[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`.

```
 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 [Potanin et al. 2013; Zibin et al. 2010]:

- 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 **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.
- No casting or promotion from **read** to **mut** is allowed.

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 and L42 do not guarantee that **capsule** fields contain a **capsule** reference, as they provides non-destructive reads [Giannini et al. 2019; Servetto et al. 2013]. In particular, Pony and L42's pre-existing **capsule** fields do not prevent representation exposure. In Section 3 we present a novel kind of **capsule** field that does not have these problems; 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 [Clarke and Wrigstad 2003; Gordon et al. 2012; Haller and Odersky 2010; Servetto and Zucca 2015; Zibin et al. 2010]. 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 [Clebsch et al. 2015; Gordon et al. 2012; Servetto and Zucca 2015] 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** [Clebsch et al. 2015; Giannini et al. 2019], 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 [Clebsch et al. 2015, 2017; Gordon et al. 2012; Servetto et al. 2013; Servetto and Zucca 2015]. From a usability perspective, this improvement means that these RCs are opt-in: a programmer can write large sections of code simply using **mut** types and be free to have rampant aliasing. Then, at a later stage, another programmer may still be able to encapsulate those data structures into an **imm** or **capsule** reference.

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* [Abrahams 2000; Lagorio and Servetto 2011] 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[6] in the following way:[7]

- Code inside a **try** block that captures unchecked exceptions is typed as if all **mut** variables declared outside of the block are **read**.
- Only **imm** objects may be thrown as unchecked exceptions.

This strategy does not restrict 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.[8]

Object Capabilities

OCs, which L42, Pony, and Gordon *et al.*'s work have, are a widely used [Karger 1988; Miller et al. 2003; Noble et al. 2016] programming technique where access rights to resources are encoded as references to objects. When this style is respected, code that does not possess an alias to such an object cannot use its associated resource. Here, as in Gordon *et al.*'s work, we 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 [Finifter et al. 2008].

OCs are typically not part of the type system nor do they require runtime checks or special support beyond that provided by a memory safe language. However, since L42 allows user code to perform foreign calls without going through a predefined standard library, the OC pattern is enforced by the type system:

- Foreign methods (which have not been whitelisted as deterministic) and methods whose names start with `#$` are *capability operations*.
- Constructors of *capability classes* are also *capability operations*.

[6]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.
[7]Formal proof that these restriction are sufficient is in the work of Lagorio [Lagorio and Servetto 2011].
[8]Transactions are another way of enforcing strong exception safety, but they require specialised and costly run time support.

- 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.

L42 capability operations are mostly used internally by capability classes, whereas user code would call normal methods on already existing OCs.

For the purposes of invariant checking, we only care about the effects that methods could have on the running program and heap. As such, *output* methods (such as a `print` method) could be white listed as 'deterministic', provided they do not affect execution, such as by non-deterministically throwing I/O errors.

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, calling `invariant()` on it is guaranteed to return **true** in a finite number of steps.

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 receivers **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**[9] 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 [Barnett and Naumann 2004] 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

L42s 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[10]; (2) if a capability object is in the ROG of

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

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

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

As we discussed before, while most approaches agree on the exact properties of a capsule reference, capsule fields are handled in different ways in different approaches. Here we present a novel kind of **capsule** field, designed to prevent representation exposure.[11] [12]

Our **capsule** fields enforce the following key properties: 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 above the current stack frame, (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.

This means the ROG of $o.f$ can only be mutated under the control of a **mut** method of o, and during such mutation, o itself cannot be seen.

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*. Capsule mutators are the only methods that are able to mutate the ROG of capsule fields. Note that a field *update* of a **capsule** field (instead of a field access) does not make a method a capsule mutator.

The following rules define our novel **capsule** fields:

- A **capsule** field can only be initialised/updated with a **capsule** expression.
- 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,
  - otherwise, the same kind of references as a **mut** field access. [14]
- 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[15].

The above rules ensure 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. When a capsule mutator is not executing, no object in the ROG of $o.f$ can be seen as **mut** or **capsule**. If execution is (indirectly) in a capsule mutator, then 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

[11] Under the similar concept of owner-as-modifier [Cunningham et al. 2008; Dietl and Müller 2005], we could consider an object to be the 'owner' of all the mutable objects in the ROG of its **capsule** fields.

[12] In L42, our new kind of **capsule** can coexist with other kinds of **capsule** fields, such as those designed for safe parallelism [Clebsch et al. 2015; Giannini et al. 2019; Gordon et al. 2012].

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

[14] Thus, an **imm** if the receiver is **imm**, a **read** if the receiver is **read**, or a **capsule** if the receiver is **capsule**. This last case is safe since a **capsule** receiver object will then be unreachable, so do not need to preserve its invariant.

[15] 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.

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 *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 **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.
- After a field update, on the receiver.
- After a capsule mutator method returns, on the receiver of the method[16].

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 [Feldman et al. 2006], 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

[16]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.

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.

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() {//Creates a new
    return new Random().bool(); }}//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 capability objects, and restricts their creation, the above example is prevented.

Allowing Internal Mutation Through Back Doors?
Rust [Matsakis and Klock II 2014] and Javari [Tschantz and Ernst 2005] 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
  method Int incr(){/*return 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 [Gordon et al. 2012] 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 action. Since we represent invariant failures by throwing unchecked exceptions, programs can recover from them with a conventional **try–catch**. Due to the guarantees of strong exception safety, any object that has been mutated during a **try** block is now unreachable, as happens in alias burying [Boyland 2001]. 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**, it is guaranteed to be garbage collectable after the **try** is completed.

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

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 [Bloch 2008] 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 RCs 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 would require a paper of its own, and indeed many such papers exist in the literature [Clebsch et al. 2015; Gordon et al. 2012; Lagorio and Servetto 2011; Servetto et al. 2013; Servetto and Zucca 2015]. Thus we are assuming that we already have an expressive and sound type system enforcing the properties we need, and instead focus on invariant checking. We clearly list in Appendix A the assumptions we make on such a type system, so that any language satisfying them, such as L42, can soundly support our invariant protocol. To keep our small step semantics as conventional as possible, we follow Pierce [Pierce 2002] and Featherweight Java [Igarashi et al. 2001]; 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. Constructors are all of the form $C(T_1 x_1, \ldots, T_n x_n)\{\mathbf{this}.f_1{=}x_1; \ldots; \mathbf{this}.f_n{=}x_n;\}$, where the fields of C are $T_1 f_1; \ldots; T_n f_n;$. We do not model custom constructors and traditional subclassing since this would make the proof more involved without adding any additional insight.

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

Fig. 1. Grammar

We additionally assume the following:

- 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, $\sigma : l \rightarrow C\{\overline{v}\}$, is a finite map from locations, l, to annotated tuples, $C\{\overline{v}\}$, representing objects; where C is the class name and $\overline{v}$ are the field values. We use the notation $\sigma[l.f = v]$ to update a field of l, $\sigma[l.f]$ to access one, and $\sigma \setminus l$ to delete l.
- The main expression is reduced in the context of a memory and program.
- A typing relation, $\Sigma; \Gamma; \mathcal{E} \vdash e : T$, where the expression e can contain locations and free variables. The types of locations are encoded in a memory environment, $\Sigma : l \rightarrow C$, while the types of free variables are encoded in a variable environment, $\Gamma : x \rightarrow T$. $\mathcal{E}$ encodes the location, relative to the top-level expression we are typing, where e was found; this is needed so that ls can be typed with different reference capabilities when in different positions.
- We use Σ^{σ} to trivially extract the corresponding Σ from a σ.

To encode object capabilities and I/O, we assume a special location c of class **Cap**. This location 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 '**true**', and
- all other methods in the **Cap** class must require a **mut** receiver; such methods will have a non-deterministic body, i.e. calls to them may have multiple possible reductions.

For simplicity, we do not formalise actual exception objects, rather we have *errors*, which correspond to expressions which are currently 'throwing' an 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 proof.

Grammar

The grammar is defined in Figure 1. Most of our expressions are standard. *Monitor expressions* are the syntactic representation of our injected invariant checks. They are of the form $\mathtt{M}(l; e_1; e_2)$, they are runtime expressions and thus are not present in method bodies, rather they are generated by our reduction rules inside the main expression. Here, l refers to the object being monitored, e_1 is the expression which is being monitored, and e_2 denotes the evaluation of $l.\mathtt{invariant}()$; e_1 will be

(UPDATE)

$$\frac{}{\sigma|l.f\ \mathtt{=}\ v \longrightarrow \sigma[l.f = v]|\mathtt{M(}l\mathtt{;}\,l\mathtt{;}\,l.\mathtt{invariant())}}$$

(NEW)

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

(MCALL)

$$\frac{}{\sigma|l.m\mathtt{(}v_1, ..., v_n\mathtt{)} \longrightarrow \sigma|e'[\mathtt{this} \coloneqq l, x_1 \coloneqq v_1, ..., x_n \coloneqq v_n]}$$

$\sigma(l) = C\{\_\}$
$C.m = \mu\ \mathtt{method}\ T\ m\mathtt{(}T_1\ x_1...T_n x_n\mathtt{)}\ e$
if $\mu = \mathtt{mut}$ and $\exists f$ such that
  $C.f = \mathtt{capsule}\ \_$ and $e = \mathcal{E}[\mathtt{this}.f]$
then $e' = \mathtt{M(}l\mathtt{;}\,e\mathtt{;}\,l.\mathtt{invariant())}$
otherwise $e' = e$

(MONITOR EXIT)

$$\frac{}{\sigma|\mathtt{M(}l\mathtt{;}\,v\mathtt{;}\,\mathtt{true)} \longrightarrow \sigma|v}$$

(CTXV)

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

(TRY ENTER)

$$\frac{}{\sigma|\mathtt{try}\ \{e_1\}\ \mathtt{catch}\ \{e_2\} \longrightarrow \sigma|\mathtt{try}^{\sigma}\{e_1\}\ \mathtt{catch}\ \{e_2\}}$$

(TRY OK)

$$\frac{}{\sigma, \sigma'|\mathtt{try}^{\sigma}\{v\}\ \mathtt{catch}\ \{\_\} \longrightarrow \sigma, \sigma'|v}$$

(TRY ERROR)

$$\frac{}{\sigma, \sigma'|\mathtt{try}^{\sigma}\{\mathit{error}\}\ \mathtt{catch}\ \{e\} \longrightarrow \sigma, \sigma'|e}$$

(ACCESS)

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

Fig. 2. Reduction rules

evaluated to a value, and the e_2 will be further evaluated, if e_2 evaluated to **false** or an *error*, then l's invariant failed to hold; such a monitor expression corresponds to the throwing of an unchecked exception. In addition, our reduction rules will annotate **try** expressions with the original state of memory. This is used in our type-system assumptions (see appendix A) to model the guarantee of strong exception safety, that is, the annotated memory will not be mutated by executing the body of the **try**.

Well-Formedness Criteria and Reduction Rules

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

- `invariant()`s and capsule mutators follow the requirements of Section 3.
- In methods, field accesses/updates are of form $\mathtt{this}.f$ / $\mathtt{this}.f\ \mathtt{=}\ e$.
- In the main expression, field accesses/updates are of form $l.f$ / $l.f\ \mathtt{=}\ e$.
- Method bodies do not contain runtime expressions (i.e. l, $\mathtt{M}$, or $\mathtt{try}^{\sigma}$).

Our reduction rules are defined in Figure 2. They are standard, except for our handling of monitor expressions. Monitor expressions are added after all field updates, **new** expressions, and calls to capsule mutators. Monitor expressions are only a proof device, they need not be implemented directly as presented. For example, in L42 we implement them by statically injecting calls to `invariant()` at the end of setters, factory methods, and capsule mutators; this works as L42 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.

The failure of a monitor expression, $\mathtt{M(}l\mathtt{;}\,e_1\mathtt{;}\,e_2\mathtt{)}$, will be caught by our TRY ERROR rule, as will any other uncaught monitor failure in e_1 or e_2.

Statement of Soundness

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

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

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

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

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

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

- $r_l = l.\mathtt{invariant()}$ and $\mathcal{E}_v = \mathcal{E}_v'[\mathtt{M(}l\mathtt{;}v\mathtt{;}\square\mathtt{)}]$, or
- $r_l = l.f$ and $\mathcal{E}_v = \mathcal{E}_v'[\mathtt{M(}l\mathtt{;}v\mathtt{;}\mathcal{E}_v''\mathtt{)}]$.

We define a *validState* as one that was obtained by any number of reductions from a well typed initial expression and memory, containing no monitors and with only the c memory location available:

$validState(\sigma, e)$ iff $c \mapsto \mathtt{Cap\{\}}|e_0 \rightarrow^{+} \sigma|e$, for some e_0 with:

$c : \mathtt{Cap}; \emptyset; \square \vdash e_0 : T$, $\mathtt{M(\_;\_;\_)} \notin e_0$, and if $l \in e_0$ then $l = c$.

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

Theorem 1 (Soundness). If $validState(\sigma, \mathcal{E}_v[r_l])$, then either $valid(\sigma, l)$ or $trusted(\mathcal{E}_v, r_l)$.

Except for the injected invariant checks (and their field accesses), any redex in the execution of a well typed program takes in input only valid objects.

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 interact 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 **Widgets**' size and position as well as children (a list of **Widgets**). Both **SafeMovable**s and **Button**s implement **Widget**. Crucially, since the children of **SafeMovable** are stored in a list of **Widgets** 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[17], 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 Widgets**.

```
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); }}
```

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

```
  read method Bool invariant() {..}
  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.

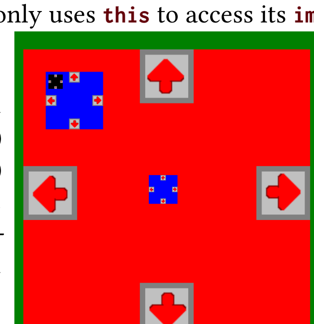
Our example shows how to encode interactive GUI programs, where widgets may circularly reference other widgets. In order to perform this case study we had to first implement a simple GUI Library in L42. This library uses object capabilities to draw the widgets on screen, as well as fetch and dispatch 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:

```
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 to the right, 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 [Alexandrescu 2010; Meyer 1992], where the invariant of the receiver is instead checked at the start and end of *every* public (in D) and qualified[18] (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' [Meyer 1992], 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; in comparison our invariant protocol only performed 77 calls. The number of checks is exponential in the depth of the GUI: the invariant of a **`SafeMovable`** will call the `width()`, `height()`, `left()`, and `top()` methods of its children, which may themselves be **`SafeMovable`**s, and hence such calls may invoke further invariant checks. Note that `width()` and `height()` are simply getters for fields, whereas the other two are non-trivial *methods*. Concluding, we 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#[19]; 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 [Barnett et al. 2005a]), which only gave us 2 warnings[20]: that 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 [Fähndrich et al. 2010], 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#.

---
[18] That is, the receiver is not **`this`**.
[19] We compiled Spec# using the latest available source (from 19/9/2014). The verifier available online at rise4fun.com/SpecSharp behalves differently.
[20] 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.

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	18	18
Characters (with minimal whitespace)	619	207	74	60

To encode the GUI example in L42, the only annotations we needed were the 3 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#. In the table we present 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);
```

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.

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) {
    return that instanceof Point &&
      this.x == ((Point)that).x && this.y == ((Point)that).y; }}
class Hamster {Point pos; //pos is imm by default
  Hamster(Point pos) {..}}
class Cage {
  capsule Hamster h;
  List<Point> path; //path is imm by default
  Cage(capsule Hamster h, List<Point> path) {..}
  read method Bool invariant() {
    return this.path.contains(this.h.pos); }
```

```
  mut method Void move() {
    Int index = 1 + this.path.indexOf(this.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. This contrasts with Spec#: all code involved in verification needs to be designed with verification in mind [Barnett et al. 2011].

Comparison with Spec#

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

```
// 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.[21] 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 our custom **Equal**(x,y) method on **Point**: this is needed since we can't overload the usual **Object**.**Equals**(other) method because 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.

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# looses 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 were 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.

[21] There is a paper [Leino et al. 2008] 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.

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++) { // Run for 1 year
  fam.processDay(day);
}
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[22], 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[23] 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

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

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 worse than the code following our approach that we showed above, which has 14 annotations and 14 tokens.

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 mains ones that introduced their methodology and extensions:

- *Verification of Object-Oriented Programs with Invariants:* [Barnett et al. 2004a] 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[24]. 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 essentially copy and paste the methods from **FileList**, and they will automatically enforce the invariant of **DirFileList**.
- *Object Invariants in Dynamic Contexts:* [Leino and Müller 2004] 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[25] (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:* [Barnett and Naumann 2004] this paper shows how one can verify invariants over interacting objects, where neither

[24]Our encodings are in the artifact.
[25]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.

owns/contains the other. They have multiple examples which utilise the 'subject/observer' pattern, where a 'subject' has some state that an 'observer' wants to keep track of. In their **Subject**/**View** example, **View**s are created with references to **Subject**s, and copies of their state. When a **Subject**'s state is modified, it calls a method on its attached **View**s, notifying them of this update. The invariant is that a **View**'s copy of its **Subject**'s state is up to date. Their **Master**/**Clock** example is similar, a **Clock** contains a reference to a **Master**, and saves a copy of the **Master**'s time. The **Master** has a **Tick** method that increases its time, but unlike the **Subject**/**View** example, the **Clock** is not notified. The invariant is that the **Clock**'s time is never ahead of its **Master**'s. Our protocol is unable to verify these interactions, because the interacting objects are not immutable or encapsulated by each other.

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 need not 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 resemble 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()){        //Check that the subInvariant of
      if(!wInvariant(wi)){return false;}//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 **Widget**s where 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 **Widget**s 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> { 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(Transformer<Widgets> t);
}
class SafeMovable { ...
  // A well typed capsule mutator
  mut method Void transform(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,scale); }});
  w.width(w.width() / 2); ...; w.top(w.top() / 2); }
```

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.

The **Node**s 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 customized by the library user, and arbitrary transformations can be performed on the **Graph**s. This is a generalization of the example proposed by [Summers et al. 2009](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.transform(ns->{
    mut MyNode n1=this;//single usage of capsule 'this'
    ns.add(n1);
    return 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 in the lambda in `connectWith(other,g)`, we can neither see the **read this** nor 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.

8 RELATED WORK

Reference Capabilities

We rely on a combination of RCs supported by at least 3 languages/lines of research: L42 [Giannini et al. 2016; Lagorio and Servetto 2011; Servetto et al. 2013; Servetto and Zucca 2015], Pony [Clebsch et al. 2015, 2017], and Gordon *et al.* [Gordon et al. 2012]. They all support full/deep interpretation (see page 5), without back doors. Former work [Aiken et al. 2003; Boyland 2003, 2010; Hogg 1991; Smith et al. 2000] (which eventually enabled the work of Gordon *et al.*) does not consider promotion and infers uniqueness/isolation/immutability only when starting from references that have been tracked with restrictive annotations along their whole lifetime. Other approaches like Javari [Boyland 2006; Tschantz and Ernst 2005] and Rust [Matsakis and Klock II 2014] provide back doors, which are not easily verifiable as being used properly.

Ownership [Clarke et al. 2013; Dietl et al. 2007; Zibin et al. 2010] is a popular form of aliasing control often used as a building block for static verification [Barnett et al. 2011; Müller 2002]. 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 [Miller 2006], in conjunction with reference capabilities, are able to enforce purity of code in a modular way, without requiring the use of monads. L42 and Gordon use OCs simply to reason about I/O and non-determinism. This approach is best exemplified by Joe-E [Finifter et al. 2008], 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* 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 [Drossopoulou et al. 2008; Leino and Müller 2004; Meyer 2016].

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 [Gopinathan and Rajamani 2008]. 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 [Burdy et al. 2005; Drossopoulou et al. 2008]. Some approaches ensure the invariant of the receiver of the *calling* method, rather than the *called* method [Müller et al. 2006]. JML [Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Muller, Joseph Kiniry, Patrice Chalin, Daniel M. Zimmerman, Werner Dietl 2013] 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 [Meyer 2016].
- 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' [Barnett et al. 2004a].

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 [Takikawa et al. 2012; Wrigstad et al. 2010], 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 formal proofs show that every class that compiles/type checks is soundly handled by our protocol, independently of the behaviour of code that uses such class or any other surrounding code.

Our approach works both in a library setting and with the open world assumption. Consider for example the work of Parkinson [Parkinson 2007]: 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 [Polikarpova et al. 2014] 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 [Leino 2012] is a new 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# [Barnett et al. 2005b] 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 [Naumann and Barnett 2006] 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 [Barnett et al. 2004a], where an invariant only depends on objects it owns; or *visibility* based [Barnett and Naumann 2004; Leino and Müller 2004], 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 us, does perform runtime checks for invariants and throws unchecked exceptions on failure. However Spec# does not allow soundly recovering from an invariant failure, since catching unchecked exceptions in Spec# is intentionally unsound. [Leino and Schulte 2004]

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 [Chalin 2007] discovered that developers expect specification languages to follow the semantics of the underling language, including short-circuit semantics and arithmetic exceptions; thus for example `1/0 || 2>1` should not hold, while `2>1 || 1/0` should, thanks to short circuiting. This study was influential enough to convince JML to change its interpretation of logical expressions accordingly [Chalin and Rioux 2008]. Dafny [Leino 2012] 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 [Barnett et al. 2004b] '*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 [Barnett and Schulte 2003]. 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 [Flanagan 2006]. 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.* [Voigt et al. 2013] and the extensive MOP project [Meredith et al. 2012], it seems that most runtime verification tools (RV) empower users to implement the kind of monitoring they see fit for their specific problem at hand. This means that users are responsible for deciding, designing, and encoding both the logical properties and the instrumentation criteria [Meredith et al. 2012]. 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 [Gorbovitski et al. 2008] 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 [Burdy et al. 2005] and Microsoft Code Contracts [Fähndrich et al. 2010].

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 [Findler and Felleisen 2001]. 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. [Gopinathan and Rajamani 2008] approach keeps a similar level of control: relying on powerful aspect-oriented support, they detect any field update in the whole ROG of any object, and check all the invariants that such update may have violated. We agree with their criticism of visible state semantics, where methods still have to assume that any object may be broken; in such case calling any public method would trigger an error, but while the object is just passed around (and for example stored in collections), the broken state will not be detected; Gopinathan *et al.* says "*there are many instances where o's invariant is violated by the programmer inadvertently changing the state of p when o is in a steady state. Typically, o and p are objects exposed by the API, and the programmer (who is the user of the API), unaware of the dependency between o and p, calls a method of p in such a way that o's invariant is violated. The fact that the violation occurred is detected much later, when a method of o is called again, and it is difficult to determine exactly where such violations occur.*"

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

Their approach is very computationally intensive, but we think it is powerful enough that it could even be used to roll back the very field update that caused the invariant to fail, making the object valid again. We considered a 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 [Strickland et al. 2012] lifts the techniques of gradual typing [Takikawa et al. 2015, 2012; Wrigstad et al. 2010] 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.

9 CONCLUSIONS AND FUTURE WORK

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 be sound and guarantees 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* [Stephens 2015] 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 variation of these. Crucially, any language already designed with such support can also support our invariant protocol with minimal added complexity.

For an implementation of our work to be sound, catching exceptions like stack overflows or out of memory cannot be allowed in `invariant()` methods, since they are not deterministically thrown. Currently L42 never allows catching them, however we could also write a (native) capability operation (which thus can't be used inside an invariant) that enables catching them.

REFERENCES

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

Alexander Aiken, Jeffrey S. Foster, John Kodumal, and Tachio Terauchi. 2003. 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.* 129–140. https://doi.org/10.1145/781131.781146

Andrei Alexandrescu. 2010. *The D Programming Language* (1st ed.). Addison-Wesley Professional.

Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005a. 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.* 364–387. https://doi.org/10.1007/11804192_17

Michael Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. 2004a. Verification of Object-Oriented Programs with Invariants. *Journal of Object Technology* 3, 6 (2004), 27–56. https://doi.org/10.5381/jot.2004.3.6.a2

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

Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. 2005b. The Spec# Programming System: An Overview. In *Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices* (Marseille, France) *(CASSIS'04)*. Springer-Verlag, Berlin, Heidelberg, 49–69. https://doi.org/10.1007/978-3-540-30569-9_3

Michael Barnett and David A. Naumann. 2004. 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.* 54–84. https://doi.org/10.1007/978-3-540-27764-4_5

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

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

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

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

Carl Friedrich Bolz, Antonio Cuni, Maciej FijaBkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. 2011. Allocation Removal by Partial Evaluation in a Tracing JIT. In *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation* *and Program Manipulation* (Austin, Texas, USA) *(PEPM '11)*. ACM, New York, NY, USA, 43–52. https://doi.org/10.1145/1929501.1929508

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

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

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

John Boyland. 2010. Semantics of Fractional Permissions with Nesting. *ACM Transactions on Programming Languages and Systems* 32, 6 (2010). https://doi.org/10.1145/1749608.1749611

Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. 2005. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer* 7, 3 (01 Jun 2005), 212–232. https://doi.org/10.1007/s10009-004-0167-4

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

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

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

David Clarke and Tobias Wrigstad. 2003. External Uniqueness is Unique Enough. In *ECOOP'03 - Object-Oriented Programming (Lecture Notes in Computer Science)*, Vol. 2473. Springer, 176–200. https://doi.org/10.1007/978-3-540-45070-2_9

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

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

Dave Cunningham, Werner Dietl, Sophia Drossopoulou, Adrian Francalanza, Peter Müller, and Alexander J. Summers. 2008. Universe Types for Topology and Encapsulation. In *Formal Methods for Components and Objects*. 72–112.

Werner Dietl, Sophia Drossopoulou, and Peter Müller. 2007. Generic Universe Types. In *ECOOP'07 - Object-Oriented Programming (Lecture Notes in Computer Science)*, Vol. 4609. Springer, 28–53. https://doi.org/10.1007/978-3-540-73589-2_3

Werner Dietl and Peter Müller. 2005. Universes: Lightweight Ownership for JML. *JOURNAL OF OBJECT TECHNOLOGY* 4, 8 (2005), 5–32.

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

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

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

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

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

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

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

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

Paola Giannini, Marco Servetto, and Elena Zucca. 2016. Types for Immutability and Aliasing Control. In *ICTCS'16 - Italian Conf. on Theoretical Computer Science (CEUR Workshop Proceedings)*, Vol. 1720. CEUR-WS.org, 62–74. http://ceur-ws.org/Vol-1720/full5.pdf

Paola Giannini, Marco Servetto, Elena Zucca, and James Cone. 2019. Flexible recovery of uniqueness and immutability. *Theoretical Computer Science* 764 (2019), 145 – 172. https://doi.org/10.1016/j.tcs.2018.09.001

Madhu Gopinathan and Sriram K. Rajamani. 2008. Runtime Verification. Springer-Verlag, Berlin, Heidelberg, Chapter Runtime Monitoring of Object Invariants with Guarantee, 158–172. https://doi.org/10.1007/978-3-540-89247-2_10

Michael Gorbovitski, Tom Rothamel, Yanhong A. Liu, and Scott D. Stoller. 2008. Efficient Runtime Invariant Checking: A Framework and Case Study. In *Proceedings of the 6th International Workshop on Dynamic Analysis (WODA 2008)*. ACM Press. https://doi.org/10.1145/1401827.1401837

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

Philipp Haller and Martin Odersky. 2010. Capabilities for uniqueness and borrowing. In *ECOOP'10 - Object-Oriented Programming (Lecture Notes in Computer Science)*, Theo D'Hondt (Ed.), Vol. 6183. Springer, 354–378. https://doi.org/10.1007/978-3-642-14107-2_17

John Hogg. 1991. Islands: Aliasing Protection in Object-oriented Languages. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1991*. ACM Press, 271–285.

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

Paul Ashley Karger. 1988. *Improving security and performance for capability systems*. Ph.D. Dissertation. Citeseer.

Giovanni Lagorio and Marco Servetto. 2011. Strong exception-safety for checked and unchecked exceptions. *Journal of Object Technology* 10 (2011), 1:1–20. https://doi.org/10.5381/jot.2011.10.1.a1

K. Rustan M. Leino. 2012. 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*. 9–10. https://doi.org/10.1145/2402676.2402682

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

K. Rustan M. Leino and Peter Müller. 2004. Object Invariants in Dynamic Contexts. In *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings*. 491–516. https://doi.org/10.1007/978-3-540-24851-4_22

K. Rustan M. Leino, Peter Müller, and Angela Wallenburg. 2008. Flexible Immutability with Frozen Objects. In *Verified Software: Theories, Tools, Experiments, Second International Conference, VSTTE 2008, Toronto, Canada, October 6-9, 2008. Proceedings*. 192–208. https://doi.org/10.1007/978-3-540-87873-5_17

K. Rustan M. Leino and Wolfram Schulte. 2004. Exception safety for C#. *Proceedings of the Second International Conference on Software Engineering and Formal Methods, 2004. SEFM 2004.* (2004), 218–227.

Nicholas D Matsakis and Felix S Klock II. 2014. The rust language. In *ACM SIGAda Ada Letters*, Vol. 34. ACM, 103–104. https://doi.org/10.1145/2663171.2663188

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

Bertrand Meyer. 1988. *Object-Oriented Software Construction* (1st ed.). Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

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

Bertrand Meyer. 2016. Class Invariants: Concepts, Problems, Solutions. *arXiv preprint arXiv:1608.07637* (2016).

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

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

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

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

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

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

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

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

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

Nadia Polikarpova, Julian Tschannen, Carlo A. Furia, and Bertrand Meyer. 2014. Flexible Invariants through Semantic Collaboration. In *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings.* 514–530. https://doi.org/10.1007/978-3-319-06410-9_35

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

Marco Servetto, David J. Pearce, Lindsay Groves, and Alex Potanin. 2013. Balloon Types for Safe Parallelisation over Arbitrary Object Graphs. In *WODET 2014 - Workshop on Determinism and Correctness in Parallel Programming.* https://doi.org/doi=10.1.1.353.2449

Marco Servetto and Elena Zucca. 2015. Aliasing Control in an Imperative Pure Calculus. In *Programming Languages and Systems - 13th Asian Symposium (APLAS) (Lecture Notes in Computer Science)*, Xinyu Feng and Sungwoo Park (Eds.), Vol. 9458. Springer, 208–228. https://doi.org/10.1007/978-3-319-26529-2_12

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

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

T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. 2012. 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.* 943–962. https://doi.org/10.1145/2384616.2384685

Alexander J. Summers, Sophia Drossopoulou, and Peter Müller. 2009. The Need for Flexible Object Invariants. In *International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming* (Genova, Italy) *(IWACO '09).* ACM, New York, NY, USA, Article 6, 9 pages. https://doi.org/10.1145/1562154.1562160

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

Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. 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.* 793–810. https://doi.org/10.1145/2384616.2384674

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

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

Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. 2010. 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.* 377–388. https://doi.org/10.1145/1706299.1706343

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

A PROOF AND AXIOMS

As previously discussed, instead of providing a concrete set of typing rules, we provide a set of properties that the type system needs to ensure. We will express such properties using type judgements of the form $\Sigma;\Gamma;\mathcal{E} \vdash e : T$. This judgement form allows an l to be typed with different types based on how it is used, e.g. we might have $\Sigma;\Gamma;\square.m(l) \vdash l : \texttt{mut}\ C$ and $\Sigma;\Gamma;l.m(\square) \nvdash l : \texttt{mut}\ C$, where m is a **mut** method taking a **read** parameter. Importantly, we allow types to change during reduction (such as to model promotions), but do not allow the types inside methods to change when they are called (see the Method Consistency assumption below).

Auxiliary Definitions

To express our type system assumptions, we first need some auxiliary definitions. 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 $l \in rog(\sigma, l')$,

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

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:

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

- $e = \mathcal{E}[l']$, $\Sigma^\sigma; \emptyset; \mathcal{E} \vdash l' : \texttt{imm}\_$, and $l \in rog(\sigma, l')$, or
- $reachable(\sigma, e, l')$, $\Sigma^\sigma(l').f = \texttt{imm}\_$, and $l \in rog(\sigma, \sigma[l'.f])$.

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

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

- $l' = l$ or
- $\exists f$ such that $\Sigma^\sigma(l).f \in \{\texttt{capsule}\_, \texttt{mut}\_\}$, and $l' \in mrog(\sigma, \sigma[l.f])$

Now we can define what it means for an l to be *mutatable*[26] by a sub-expression e, found in $\mathcal{E}$: something in l is reachable from a **mut** reference in e, by passing through any number of **mut** and **capsule** fields:

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

- $e = \mathcal{E}'[l']$, $\Sigma^\sigma; \emptyset; \mathcal{E}[\mathcal{E}'] \vdash l' : \texttt{mut}\_$, and
- $mrog(\sigma, l')$ not disjoint $rog(\sigma, l)$.

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

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

Axiomatic Type Properties

Here we assume a slight variation of the usual Subject Reduction: a (sub) expression obtained using any number of reductions, from a well-typed and well-formed initial $\sigma_0|e_0$, is also well-typed:

Assumption 1 (Subject Reduction). If $validState(\sigma, \mathcal{E}[e])$, then $\Sigma^\sigma; \emptyset; \mathcal{E} \vdash e : T$.

As we do not have a concrete type system, we need to assume some properties about its derivations. First we require that **new** expressions only have field initialisers with the appropriate type, fields are only updated with expressions of the appropriate type, methods are only called on receivers with the appropriate RC, method parameters have the appropriate type, and method calls are typed with the return type of the method:

Assumption 2 (Type Consistency).

(1) If $C.i = T_i\_$, then $\Sigma; \Gamma; \mathcal{E}[\texttt{new}\ C(e_1, ..., e_{i-1}, \square, e_{i+1}, ..., e_n)] \vdash e_i : T_i$.

(2) If $\Sigma; \Gamma; \mathcal{E}[\square.f = e'] \vdash e : \_C$ and $C.f = T'\ f$, then $\Sigma; \Gamma; \mathcal{E}[e.f = \square] \vdash e' : T'$.

(3) If $\Sigma; \Gamma; \mathcal{E}[\square.m(e_1, ..., e_n)] \vdash e : \_C$ and $C.m = \mu\ \texttt{method}\ T\ m(T_1\ x_1, ..., T_n x_n)\ \_$, then:

  (a) $\Sigma; \Gamma; \mathcal{E}[\square.m(e_1, ..., e_n)] \vdash e : \mu\, C$,

  (b) $\Sigma; \Gamma; \mathcal{E}[e.m(e_1, ..., e_{i-1}, \square, e_{i+1}, ..., e_n)] \vdash e_i : T_i$, and

  (c) $\Sigma; \Gamma; \mathcal{E} \vdash e.m(e_1, ..., e_n) : T$.

We also assume that any expression inside a method body can be typed with the same reference capabilities as when it is expanded by our MCALL rule:

Assumption 3 (Method Consistency). If $validState(\sigma, \mathcal{E}_v[l.m(v_1, ..., v_n)])$ where:

- $\Sigma^\sigma; \emptyset; \mathcal{E}_v[\square.m(v_1, ..., v_n)] \vdash l : \_C$, $C.m = \_\texttt{method}\_\ m(T_1\ x_1, ...T_n\ x_n)\ \mathcal{E}[e]$,
- $\mathcal{E}' = \texttt{M}(l; \mathcal{E}; l.\texttt{invariant}())$ if $C.m$ is a capsule mutator, otherwise $\mathcal{E}' = \mathcal{E}$,
- $\Gamma = \texttt{this} : \mu\, C, x_1 : T_1, ..., x_n : T_n$, and $e' = e[\texttt{this} := l, x_1 := v_1, ..., x_n := v_n]$,

then $\emptyset; \Gamma; \mathcal{E} \vdash e : \mu\_$ iff $\Sigma^\sigma; \emptyset; \mathcal{E}_v[\mathcal{E}'[\texttt{this} := l, x_1 := v_1, ..., x_n := v_n]] \vdash e' : \mu\_$.

[26] We use the term *mutatable* to distinguish from *immutable*: an object might be neither *mutatable* nor *immutable*, e.g. if there are only **read** references to it.

Now we define formal properties about our RCs, thus giving them meaning. First we require that an *immutable* object not also be *mutatable*: i.e. an object reachable from an **imm** reference/field cannot also be reached from a **mut**/**capsule** reference and through **mut**/**capsule** fields:

Assumption 4 (Imm Consistency).
If $validState(\sigma, e)$ and $immutable(\sigma, e, l)$, then not $mutatable(\sigma, \Box, e, l)$.

Note that this does not prevent *promotion* from a **mut** to an **imm**: a reduction step may change the type of an l from **mut** to **imm**, provided that in the new state, the above assumption holds.

We require that if something was not *mutatable*, that it remains that way; this prevents, for example, runtime promotions from **read** to **mut**, as well as field accesses returning a **mut** from a receiver that was not **mut**:

Assumption 5 (Mut Consistency). If $validState(\sigma, \mathcal{E}_v[e])$,
not $mutatable(\sigma, \mathcal{E}_v, e, l)$, and $\sigma|\mathcal{E}_v[e] \rightarrow^+ \sigma'|\mathcal{E}_v[e']$, then not $mutatable(\sigma', \mathcal{E}_v, e', l)$.

We require that a **capsule** reference be *encapsulated*; and require that **capsule** is a subtype of **mut**:

Assumption 6 (Capsule Consistency).
(1) If $\Sigma^\sigma; \emptyset; \mathcal{E} \vdash l : \mathtt{capsule}\ \_$, then $encapsulated(\sigma, \mathcal{E}, l)$.
(2) If $\Sigma; \Gamma; \mathcal{E} \vdash e : \mathtt{capsule}\ C$, then $\Sigma; \Gamma; \mathcal{E} \vdash e : \mathtt{mut}\ C$.

We require that field updates only be performed on **mut** receivers:

Assumption 7 (Mut Update). If $\Sigma; \Gamma; \mathcal{E} \vdash e.f = e' : T$, then $\Sigma; \Gamma; \mathcal{E}[\Box.f = e'] \vdash e : \mathtt{mut}\ \_$.

We additionally require that field accesses only be typed as **mut**, if their receiver is also **mut**:

Assumption 8 (Mut Access). If $\Sigma; \Gamma; \mathcal{E} \vdash e.f : \mathtt{mut}\ \_$, then $\Sigma; \Gamma; \mathcal{E}[\Box.f] \vdash e : \mathtt{mut}\ \_$.

Finally, we require that a **read** variable or method result not be typeable as **mut**; in conjunction with Mut Consistency, Mut Update, and Method Consistency, this allows one to safely pass or return a **read** without it being used to modify the object's *rog*:

Assumption 9 (Read Consistency).
(1) If $\Gamma(x) = \mathtt{read}\ \_$, then $\Sigma; \Gamma; \mathcal{E} \nvdash x : \mathtt{mut}\ \_$.
(2) If $\Sigma; \Gamma; \mathcal{E}[\Box.m(\overline{e})] \vdash e : \_\ C$ and $C.m = \_\ \mathtt{method\ read}\ C'\ \_$, then $\Sigma; \Gamma; \mathcal{E} \nvdash e.m(\overline{e}) : \mathtt{mut}\ \_$.

Note that Mut Consistency prevents an access to a **read** field from being typed as **mut**

Strong Exception Safety

Finally we assume strong exception safety: the memory preserved by each **try**–**catch** execution is not *mutatable* within the **try**:

Assumption 10 (Strong Exception Safety). If $validState(\sigma', \mathcal{E}[\mathtt{try}^{\sigma_0}\{e_0\}\ \mathtt{catch}\ \{e_1\}])$, then $\forall l \in dom(\sigma_0)$, not $mutatable(\sigma, \mathcal{E}[\mathtt{try}^{\sigma_0}\{\Box\}\ \mathtt{catch}\ \{e_1\}], e_0, l)$.

We use strong exception safety to prove that locations preserved by **try** blocks are never monitored (this is important as it means that a **catch** that catches a monitor failure will not be able to see the responsible object):

Lemma 1 (Unmonitored Try). If $validState(\sigma, e)$, $\forall \mathcal{E}$, if $e = \mathcal{E}[\mathtt{try}^{\sigma_0}\{\mathcal{E}[\mathtt{M}(l;\_;\_)]\}\ \_]$, then $l \notin \sigma_0$

Proof. The proof is by induction: after 0 reduction steps, e cannot contain a monitor expression by the definition of *validState*. If this property holds for $validState(\sigma, e)$ but not for $\sigma'|e'$ with $\sigma|e \rightarrow \sigma'|e'$, we must have applied the update, mcall, or new rules; since our well-formedness rules on method bodies prevent any other reduction step from introducing a monitor expression. If the reduction was a new, l will be fresh, so it could not have been in σ_0. If the reduction was an update, by Mut Update, l must have been **mut**, similarly mcall will only introduce a monitor over a call to a **mut** method, so by Type Consistency, l was **mut**; either way we have that l was *mutatable*, since our reductions never change the σ_0 annotation, by Strong Exception Safety, we have that $l \notin \sigma_0$.

Determinism

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 2 (Determinism). If $validState(\sigma, \mathcal{E}_v[l.\texttt{invariant()}])$ and
$\sigma|\mathcal{E}_v[l.\texttt{invariant()}] \rightarrow \sigma'|\mathcal{E}_v[e'] \rightarrow^+ \sigma''|\mathcal{E}_v[e'']$,
then $\sigma'' = \sigma, \_, \sigma|\mathcal{E}_v[l.\texttt{invariant()}] \Rightarrow^+ \sigma''|\mathcal{E}_v[e'']$, and $\forall l' \in dom(\sigma)$, not $mutatable(\sigma'', \mathcal{E}_v, e'', l)$.

Proof. The proof will proceed by induction.

Base case: If $\sigma|\mathcal{E}_v[l.\texttt{invariant()}] \rightarrow \sigma'|\mathcal{E}_v[e']$, then the reduction was performed by MCALL. By our well-formedness rules, the `invariant()` method takes a **`read this`**, so by Method Consistency and Read Consistency, we have that l is not *mutatable* in e'. By our well-formedness rules on method bodies and MCALL, we have that no other l' was introduced in e', thus nothing is *mutatable* in e'.

The only non-deterministic single reduction steps are for calls to **`mut`** methods on a **`Cap`**; however `invariant()` is a **`read`** method, so even if $l = c$, we have $\sigma|\mathcal{E}_v[l.\texttt{invariant()}] \Rightarrow \sigma'|\mathcal{E}_v[e']$. In addition, since MCALL does not mutate σ' with have $\sigma' = \sigma$.

Inductive case: Consider $\sigma|\mathcal{E}_v[l.\texttt{invariant()}] \Rightarrow^+ \sigma'|\mathcal{E}_v[e'] \rightarrow \sigma''|\mathcal{E}_v[e'']$. We inductively assume that $\forall l' \in dom(\sigma)$, not $mutatable(\sigma', \mathcal{E}_v, e', l)$; thus by Mut Consistency, each such l' is not *mutatable* in e'. We also inductively assume that $\sigma' = \sigma, \_$, since nothing in σ was *mutatable*: by Mut Update, our reduction can't have modified anything in σ, i.e. $\sigma'' = \sigma, \_$. As our reduction rules never remove things from memory, $c \in dom(\sigma)$, so it can't by *mutatable* in e'. By definition of **`Cap`**, no other instances of **`Cap`** exist, thus by Type Consistency, no **`mut`** methods of **`Cap`** can be called; since calling such a method is the only way to get a non-deterministic reduction, we have $\sigma'|\mathcal{E}_v[e'] \Rightarrow \sigma''|\mathcal{E}_v[e'']$.

Capsule Field Soundness

Now we define and prove important properties about our novel **`capsule`** fields. We first start with a few core auxiliary definitions. We define a notation to easily get the **`capsule`** field declarations for an l:

$f \in capsuleFields(\sigma, l)$ iff $\Sigma^\sigma(l).f = \texttt{capsule}\ \_$.

An l is *capsuleNotCircular* if it is not reachable from its **`capsule`** fields:

$capsuleNotCircular(\sigma, l)$ iff $\forall f \in capsuleFields(\sigma, l), l \notin rog(\sigma, \sigma[l.f])$.

We say that an l is *wellEncapsulated* if none of its **`capsule`** fields is *mutatable* without passing through l:

$wellEncapsulated(\sigma, e, l)$ iff $\forall f \in capsuleFields(\sigma, l)$, not $mutatable(\sigma \setminus l, \square, e, \sigma[l.f])$.

We say that an l is *notCapsuleMutating* if we aren't in a monitor for l which must have been introduced by MCALL, and we don't access any of it's **`capsule`** fields as **`mut`**:

$notCapsuleMutating(\sigma, e, l)$ iff $\forall \mathcal{E}$:

- if $e = \mathcal{E}[\texttt{M(}l\texttt{;}e'\texttt{;}\_\texttt{)}]$, then $e' = l$, and
- if $e = \mathcal{E}[l.f]$, $f \in capsuleFields(\sigma, l)$, and $\Sigma^\sigma; \emptyset; \mathcal{E}[\square.f] \nvdash l : \texttt{capsule}\ \_$, then $\Sigma^\sigma; \emptyset; \mathcal{E} \nvdash l.f : \texttt{mut}\ \_$.

Finally we say that l is *headNotObservable* if we are in a monitor introduced for a call to a capsule mutator, and l is not reachable from inside this monitor, except perhaps through a single **`capsule`** field access.

$headNotObservable(\sigma, e, l)$ iff $e = \mathcal{E}_v[\texttt{M(}l\texttt{;}e'\texttt{;}\_\texttt{)}]$, and either:

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

Now we formally state the core properties of our **`capsule`** fields (informally described in 3):

Theorem 2 (Capsule Field Soundness). If $validState(\sigma, e)$ then $\forall l$, if $reachable(\sigma, e, l)$, then: $capsuleNotCircular(\sigma, l)$ and either:

- *wellEncapsulated*(σ, e, l) and *notCapsuleMutating*(σ, e, l), or
- *headNotObservable*(σ, e, l).

Proof. This trivially holds in the base case when $\sigma = c \mapsto$ `Cap{}`, since **`Cap`** has no **`capsule`** fields and the initial main expression cannot have monitors. Now we suppose it holds for a *validState* and prove it for the next *validState*.

Note that any single reduction step can be obtained by exactly one application of the CTXV rule and one other rule. We will first proceed by cases on the property we need to prove, and then by the non-CTXV reduction rules that could violate or ensure it:

(1) *capsuleNotCircular*:
  (a) (NEW) $\sigma|\mathcal{E}_v[\texttt{new } C\texttt{(}v_1, ..., v_n\texttt{)}] \rightarrow \sigma'|\mathcal{E}_v[\texttt{M(}l\texttt{;}l\texttt{;}l\texttt{.invariant())}]$, where $\sigma' = \sigma, l \mapsto C\{v_1, ..., v_n\}$:
    - This reduction step doesn't modify any pre-existing l', so we can't have broken *capsuleNotCircular* for them.
    - Since the pre-existing σ was not modified, by *validState*, $l \notin rog(\sigma, v_i) = rog(\sigma', \sigma'[l.f])$; thus *capsuleNotCircular* holds for l.
  (b) (UPDATE) $\sigma|\mathcal{E}_v[l.f \texttt{ = } v] \rightarrow \sigma[l.f = v]|\mathcal{E}_v[\texttt{M(}l\texttt{;}l\texttt{;}l\texttt{.invariant())}]$:
    - If $f \in capsuleFields(\sigma, l)$: by Mut Update, we have that l is *mutatable*, so by Type Consistency and Capsule Consistency, *encapsulated*$(\sigma, \mathcal{E}_v[l.f = \square], v)$, hence l is not *reachable* from v, and so after the update, *capsuleNotCircular* still holds for l.
    - Now consider any l' and $f' \in capsuleFields(\sigma, l')$, with $l'.f' \neq l.f$:
      – If l' was *wellEncapsulated*, by Mut Update, l is **`mut`**. By *wellEncapsulated*, the *rog* of $l'.f'$ is not *mutatable* (except through a field *access* on l'), thus we have that $l \notin rog(\sigma, \sigma[l'.f'])$, in addition, since $l'.f' \neq l.f$, we can't have modified the *rog* of $l'.f'$, hence l' is still *capsuleNotCircular*.
      – Otherwise, l' was *headNotObservable*, and so $l' \notin rog(\sigma, v)$, so we can't have added l' to the *rog* of anything, thus *capsuleNotCircular* still holds.
  (c) No other reduction rule modifies memory, so they trivially preserve *capsuleNotCircular* for all ls.

(2) *headNotObservable*:
  (a) (ACCESS) $\sigma|\mathcal{E}_v[l.f] \rightarrow \sigma|\mathcal{E}_v[\sigma[l.f]]$:
    - Suppose l was *headNotObservable*, then $\mathcal{E}_v = \mathcal{E}_v'[\texttt{M(}l\texttt{;}\mathcal{E}[l.f]\texttt{;\_)}]$, with l not *reachable* from $\mathcal{E}$, and $l.f$ is an access to a **`capsule`** field. By *capsuleNotCircular*, l is not in the *rog* of $\sigma[l.f]$, and so l is not *reachable* from $\mathcal{E}[\sigma[l.f]]$, and so *headNotObservable* still holds.
    - Clearly this reduction cannot have made any l' *reachable* in a sub-expression where it wasn't already *reachable*, so we can't have violated *headNotObservable* for any other l'.
  (b) (MONITOR EXIT) $\sigma|\mathcal{E}_v[\texttt{M(}l\texttt{;}v\texttt{;true)}] \rightarrow \sigma|\mathcal{E}_v[v]$:
    - As with the above case, we can't have violated *headNotObservable* for any $l' \neq l$.
    - If this monitor was introduced by NEW or UPDATE, then $v = l$. And so *headNotObservable* can't have held for l since $l = v$, and v was not the receiver of a field access.
    - Otherwise, this monitor was introduce by MCALL, due to a call to a capsule mutator on l. Consider the state $\sigma_0|\mathcal{E}_v[e_0]$ immediately before that MCALL:
      – We must not have had that l was *headNotObservable*, since e_0 would contain l as the receiver of a method call. Thus, by induction, l was originally *wellEncapsulated* and *notCapsuleMutating*.
      – Because *notCapsuleMutating* held in $s_0|\mathcal{E}_v[e_0]$, and v contains no field accesses or monitor, it also holds in $\mathcal{E}_v[v]$.
      – Since a capsule mutator cannot have any **`mut`** parameters, by Type Consistency, Mut Consistency, and Mut Update, the body of the method can't have modified σ_0: thus $\sigma = \sigma_0, \_$. Since no pre-existing memory has changed since the MCALL, and a capsule

mutator cannot have a **mut** return type, by Type Consistency, we must have $\Sigma^\sigma; \emptyset; \mathcal{E}_v \vdash v : \mu\_$ where $\mu \neq$ mut:

* If $\mu =$ capsule, by Capsule Consistency, the value of any **capsule** field of l can't be in the *rog* of v (unless l is no longer *reachable*), so we haven't made such a field *mutatable*.
* Otherwise, $\mu \in \{$read, imm$\}$, by Read Consistency, Imm Consistency, and Mut Consistency, we have that v is not *mutatable*.

Either way, the MONITOR EXIT reduction has restored *wellEncapsulated*$(\sigma_0, \mathcal{E}_v[e_0], l)$.

(c) (TRY ERROR) $\sigma|\mathcal{E}_v[$try$^{\sigma_0}$ {$error$} catch {e}$] \rightarrow \sigma|\mathcal{E}_v[e]$, where $error = \mathcal{E}_v'[$M(l;_;_)$]$:
By our reduction rules, we were previously in state $\sigma_0|\mathcal{E}_v[$try {e_0} catch {e}$]$. By Unmonitored Try, $l \notin dom(\sigma_0)$, and so l was not *reachable* from $\mathcal{E}_v[$try {e_0} catch {e}$]$. By Strong Exception Safety, we have that nothing in σ_0 has changed, so we must still have that l is not *reachable* from $\mathcal{E}_v[e]$: thus it doesn't matter that l is no longer *headNotObservable*.

(d) No other rules remove monitors or field accesses, or make something *reachable* that wasn't before; thus they preserve *headNotObservable* for all ls.

(3) *notCapsuleMutating*:

(a) (MCALL) $\sigma|\mathcal{E}_v[l.m(v_1, ..., v_n)] \rightarrow \sigma|\mathcal{E}_v[e]$:

- Suppose m is not a capsule mutator, by our well-formedness rules for method bodies, e doesn't contain a monitor.
  - Since m is not a capsule mutator, if $e = \mathcal{E}[l.f]$, for some $f \in capsuleFields(\sigma, l)$, we must have that m was not a **mut** method. So by Mut Access and Method Consistency, we have that $\Sigma^\sigma; \emptyset; \mathcal{E}_v[\mathcal{E}] \nvdash l.f$: mut_ only if m was a **capsule** method, which by Method Consistency, would mean that $\Sigma^\sigma; \emptyset; \mathcal{E}_v[\mathcal{E}[\square.f]] \vdash l$: capsule_. So regardless of what fields e accesses on l, we can't have broken *notCapsuleMutating* for l.
  - Consider $l' \neq l$, since fields are instance-private, and by our well-formedness rules on method bodies, $l' \notin e$, thus we can't have introduced any field accesses on l. As e doesn't contain monitors either, we haven't broken *notCapsuleMutating* for l'.
- Otherwise, $e =$ M(l;e';l.invariant()). By our rules for capsule mutators, m must be a **mut** method with only **imm** and **capsule** parameters, thus by Type Consistency, l must have been **mut**, and each v_i must be **imm** or **capsule**. By Imm Consistency and Capsule Consistency, l can't be reachable from any v_i. Since capsule mutators use **this** only once, to access a **capsule** field, $e' = \mathcal{E}[l.f]$, for some $f \in capsuleFields(\sigma, l)$. Since l is not *reachable* from any v_i, $l \notin \mathcal{E}$, and by our well-formedness rules for method bodies, l is not *reachable* from any $l' \in \mathcal{E}$, thus *headNotObservable* now holds for l.

(b) Since no other rule can introduce a monitor expression over an $e \neq l$, nor introduce field access, by Mut Consistency and Mut Access, we can't have broken *notCapsuleMutating* for any l.

(4) *wellEncapsulated*:

(a) (NEW) $\sigma|\mathcal{E}_v[$new $C(v_1, ..., v_n)] \rightarrow \sigma, l \mapsto C\{v_1, ..., v_n\}|\mathcal{E}_v[$M($l$;$l$;$l$.invariant())$]$:

- Consider any pre-existing l'. Suppose we broke *wellEncapsulated* for l' by making some $f' \in capsuleFields(\sigma, l)$ *mutatable*. Since the *rog* of l' can't have been modified, nor could the *rog* of any other pre-existing l'', we must have that $\sigma[l'.f]$ is now *mutatable* through some $l.f$. This requires that a v_i be an initialiser for a **mut** or **capsule** field, which by Type Consistency and Capsule Consistency, means that v_i must also be typeable as **mut**. But then the $\sigma[l'.f']$ was already *mutatable* through v_i, so l' can't have already been *wellEncapsulated*, a contradiction.
- Now consider each i with $C.i =$ capsule _f. By Type Consistency and Capsule Consistency, v_i was *encapsulated* and $rog(\sigma, v_i)$ is not *mutatable* from $\mathcal{E}_v$, and so v_i is not

mutatable$(\sigma' \setminus l, \square, \mathcal{E}_v[\texttt{M(}l\texttt{;}l\texttt{;}l\texttt{.invariant())}], v_i)$; thus *wellEncapsulated* holds for l and each of its **capsule** fields.

(b) (UPDATE) $\sigma|\mathcal{E}_v[l.f = v] \rightarrow \sigma[l.f = v]|\mathcal{E}_v[\texttt{M(}l\texttt{;}l\texttt{;}l\texttt{.invariant())}]$:

- If l was *wellEncapsulated* and $f \in$ *capsuleFields*(σ, l), by Type Consistency and Capsule Consistency, v is *encapsulated*, thus v is not *mutatable* from $\mathcal{E}_v$, and l is not *reachable* from v, thus v is still *encapsulated* and *wellEncapsulated* still holds for l and f.
- Now consider any *wellEncapsulated* l' and $f' \in$ *capsuleFields*(σ, l'), with $l'.f' \neq l.f$; by the above UPDATE case for *capsuleNotCircular*, $l \notin$ *rog*$(\sigma, \sigma[l'.f'])$. If f was a **mut** or **capsule** field, by Type Consistency and Capsule Consistency, v was **mut**, so by *wellEncapsulated*, $v \notin$ *rog*$(\sigma, \sigma[l'.f'])$; thus we can't have made *rog*$(\sigma, \sigma[l'.f'])$ *mutatable* through $l.f$; so $l'.f'$ can't now be *mutatable* through l. By Mut Consitency, we couldn't have have made $l'.f'$ *mutatable* some other way, so l' is still *wellEncapsulated*.

(c) (ACCESS) $\sigma|\mathcal{E}_v[l.f] \rightarrow \sigma|\mathcal{E}_v[\sigma[l.f]]$:

- Suppose l was *wellEncapsulated* and *notCapsuleMutating*, and $f \in$ *capsuleFields*(σ, l), by Mut Access, either $\Sigma^\sigma; \emptyset; \mathcal{E}_v \nvdash \sigma[l.f] : \texttt{mut}\_$ or $\Sigma^\sigma; \emptyset; \mathcal{E}_v[\square.f] \vdash l : \texttt{capsule}\_$. If l was **capsule**, then by Capsule Consistency and *capsuleNotCircular*, l is not *reachable* from $\mathcal{E}_v[\sigma[l.f]]$, so it is irrelevant if l is no longer *wellEncapsulated*. Otherwise, if l was not **capsule**, $\sigma[l.f]$ will not be **mut**, so *wellEncapsulated* is preserved for l. Note that if l wasn't *notCapsuleMutating*, it was *headNotObservable*, so we don't need to preserve *wellEncapsulated*.
- Since this reduction doesn't modify memory, by Mut Consistency, there is no other way to make the *rog* of a **capsule** field f' of l' *mutatable* without going through l', so *wellEncapsulated* is preserved for l'.

(d) Since none of the other reduction rules modify memory, by Mut Consistency, they can't violate *wellEncapsulated*.

In each case above, for each l, *capsuleNotCircular* holds; and either *wellEncapsulated* and *notCapsuleMutating* holds, or *headNotObservable* holds.

Stronger Soundness

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

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

monitored(e, l) iff $e = \mathcal{E}_v[\texttt{M(}l\texttt{;}e_1\texttt{;}e_2\texttt{)}]$ and either $e_1 = l$ or $l \notin e_1$.

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_1 will reference l; but during reduction, l will be used to modify the object; 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*(σ, e) then $\forall l$, if *reachable*(σ, e, l) then *valid*(σ, l) or *monitored*(e, l).

Proof. We will prove this inductively, in a similar way to how we proved Capsule Field Soundness. In the base case, we have $\sigma = c \mapsto \texttt{Cap\{\}}$, since **Cap** is defined to have the trivial invariant, we have that c (the only thing in σ), is *valid*.

Now we assume that everything reachable from the previous *validState* was *valid* or *monitored*, and proceed by cases on the non-CTXV rule that gets us to the next *validState*.

(1) (UPDATE) $\sigma|\mathcal{E}_v[l.f = v] \rightarrow \sigma'|\mathcal{E}_v[e']$, where $e' = \texttt{M(}l\texttt{;}l\texttt{;}l\texttt{.invariant())}$:

- 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' not *valid*. By our well-formedness criteria, `invariant()` can only accesses **`imm`** and **`capsule`** fields, thus by Imm Consistency and Mut Update, we must have that l was in the *rog* of $l'.f'$, for some $f' \in capsuleFields(\sigma, l')$. Since $l \neq l'$, l' can't have been *wellEncapsulated*. Thus, by Capsule Field Soundness, l' was *headNotObservable*, and $\mathcal{E}_v = \mathcal{E}_v'[\texttt{M(}l'\texttt{;}\mathcal{E}_v''\texttt{;\_)}]$:
  - If $\mathcal{E}_v''[l.f = v] = \mathcal{E}[l'.f']$, then by *headNotObservable*, l' is not reachable from $\mathcal{E}$. The monitor must have been introduced by an MCALL, on a capsule mutator for l'. Since a capsule mutator can take only **`imm`** and **`capsule`** parameters, by Type Consistency, Imm Consistency, and Capsule Consistency, l cannot be in their *rogs* (since l was in the *rog* of l', and l is **`mut`**). Thus the only way for the body of the monitor to access l is by accessing $l'.f'$. Since capsule mutators can access **`this`** only once, and by the proof of Capsule Field Soundness, there is no other $l'.f'$ in $\mathcal{E}[l'.f']$, nor was there one in a previous stage of reduction: hence l is not *reachable* from $\mathcal{E}$. This is in contradiction with us having just updated l.
  - Thus, by *headNotObservable*, we must have $\mathcal{E}_v''[l.f = v] = e$, with l' not *reachable* from e; so l' was, and still is, *monitored*.
- Since we don't remove any monitors, we can't have violated *monitored*. In addition, if an l was not in the *rog* of a *valid* l', by Determinism, l is still *valid*.

(2) (MONITOR EXIT) $\sigma|\texttt{M(}l\texttt{;}v\texttt{;true)} \rightarrow \sigma|v$:
By our *validState* and our well-formedness requirements on method bodies, the monitor expression must have been introduced by UPDATE, MCALL, or NEW. In each case the 3rd expression started of as l`.invariant()`, and it has now (eventually) been reduced to `true`, thus by Determinism l is *valid*. This rule does not modify pre-existing memory, introduce pre-existing ls into the main expression, nor remove monitors on other ls, thus every other pre-existing l' is still *valid* (due to Determinism), or *monitored*.

(3) (NEW) $\sigma|\mathcal{E}_v[\texttt{new } C\texttt{(}\overline{v}\texttt{)}] \rightarrow \sigma, l \mapsto C\{\overline{v}\}|\mathcal{E}_v[\texttt{M(}l\texttt{;}l\texttt{;}l\texttt{.invariant())}]$:
Clearly the newly created object, l, is *monitored*. As with the case for MONITOR EXIT above, every other *reachable* l is still *valid* or *monitored*.

(4) (TRY ERROR) $\sigma|\mathcal{E}_v[\texttt{try}^{\sigma_0}\texttt{\{}error\texttt{\} catch \{}e\texttt{\}}] \rightarrow \sigma|\mathcal{E}_v[e]$, where $error = \mathcal{E}_v'[\texttt{M(}l\texttt{;\_;\_)}]$:
By the proof of Capsule Field Soundness, we must have that l is no longer *reachable*, it is ok that it is now no longer *valid* or *monitored*. As with the case for MONITOR EXIT above, every other *reachable* l is still *valid* or *monitored*.

None of the other reduction rules modify memory, the memory locations reachable inside of the main expression, or any pre-existing monitor expressions; thus regardless of the reduction performed, we have that each *reachable* l is *valid* or *monitored*.

Proof of Soundness

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 3 (IMM NOT CIRCULAR).
If $validState(\sigma, e)$, $\forall f, l$, if $reachable(\sigma, e, l)$, $\Sigma^{\sigma}(l).f = \texttt{imm}\_$, then $l \notin rog(\sigma, \sigma[l.f])$.

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

(1) Consider any pre-existing *reachable* l and f with $\Sigma^{\sigma}(l).f = \texttt{imm}\_$, by Imm Consistency and Mut Update, the only way $rog(\sigma, \sigma[l.f])$ could have changed is if $e = \mathcal{E}_v[l.f = v]$, i.e. we just applied the UPDATE rule. By Mut Update we must have that l was **`mut`**, by Type Consistency, v must have been **`imm`**, so by Imm Consistency, $l \notin rog(\sigma, v)$. Since $v = \sigma'[l.f]$, we now have $l \notin rog(\sigma', \sigma'[l.f])$.

(2) The only rule that makes an l *reachable* is NEW. So consider $e = \mathcal{E}_v[\texttt{new}\ C\texttt{(}v_1, ..., v_n\texttt{)}]$ and each i with $C.i = \texttt{imm}\ \_$. But v_i existed in the previous state and $l \notin dom(\sigma)$; so by *validState* and our reduction rules, $l \notin rog(\sigma, v_i) = rog(\sigma', \sigma'[l.f])$.

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

Theorem 1 (Soundness). If $validState(\sigma, \mathcal{E}_v[r_l])$, then either $valid(\sigma, l)$ or $trusted(\mathcal{E}_v, r_l)$.

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

- $\mathcal{E}_v = \mathcal{E}[\texttt{M(}l\texttt{;}\mathcal{E}'\texttt{;}e_2\texttt{)}]$, that is r_l (which by definition cannot equal l) was found inside of e_1, this contradicts the definition of *monitored*, or
- $\mathcal{E}_v = \mathcal{E}[\texttt{M(}l\texttt{;}e_1\texttt{;}\mathcal{E}'\texttt{)}]$, and thus r_l was found inside e_2. By our reduction rules, all monitor expressions start with $e_2 = l\texttt{.invariant()}$; if this has yet to be reduced, then $\mathcal{E}'[r_l] = l\texttt{.invariant()}$, thus r_l is *trusted*. The next execution step will be an MCALL, so by our well-formedness rules for `invariant()`, e_2 will only contain l as the receiver of a field access; so if we just performed said MCALL, $r_l = l.f$: hence r_l is trusted. Otherwise, by Imm Not Circular, Capsule Field Soundness, and *capsuleNotCircular*, no further reductions of e_2 could have introduced an occurrence of l, so we must have that r_l was introduced by the MCALL to `invariant()`, and so it is *trusted*.

Thus either l is *valid* or r_l is *trusted*.