Using Type Modifiers for Sound Runtime Invariant Checking

First Author[1][0000−1111−2222−3333], Second Author[2,3][1111−2222−3333−4444], and Third Author[3][2222−−3333−4444−5555]

[1] Princeton University, Princeton NJ 08544, USA
[2] Springer Heidelberg, Tiergartenstr. 17, 69121 Heidelberg, Germany
lncs@springer.com
http://www.springer.com/gp/computer-science/lncs
[3] ABC Institute, Rupert-Karls-University Heidelberg, Heidelberg, Germany
{abc,lncs}@uni-heidelberg.de

Abstract. In this paper we use pre-existing language support for type modifiers to enable a system for sound runtime verification of invariants. Our system guarantees that class invariants hold for all objects involved in execution. Invariants are specified simply as methods whose execution is statically guaranteed to be deterministic and not access any externally mutable state. We automatically call such invariant methods only when objects are created or the state they refer to may have been mutated. Our design restricts the range of expressible invariants, but improves upon the usability and performance of prior work. In addition, we soundly support mutation, dynamic dispatch, exceptions, and non-deterministic I/O, while requiring only a modest amount of annotation.
We present case studies showing that our system requires a lower annotation burden compared to Spec#, and performs orders of magnitude less runtime invariant checks compared to the widely used 'visible state semantics' protocols of D and Eiffel. We also formalise our approach and prove that such pre-existing type modifier support is sufficient to ensure its soundness.

Keywords: Type modifiers · Runtime verification · Class invariants.

1 Introduction

Class invariants are a useful concept when reasoning about software correctness in OO (object oriented) languages, and 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 boolean method called `invariant`. We say that an object's invariant holds when its `invariant` method would return `true`. We do this, like Dafny [?], to minimise special syntactic and type-system treatment of invariants.

Class invariants are designed to hold in most moments, but they can be (temporarily) broken and observed broken. The two main sound invariant protocols present in literature are *visible state semantics* [**?**] and the *Boogie/Pack-Unpack methodology* [**?**]. In the visible state semantics, they can be broken when a method on the object is active (that is, one of the object methods is currently in execution somewhere in the stack trace). Some interpretations of the visible state semantic are more permissive, requiring the invariants of receivers to hold before and after every public method call, and after constructors. In pack-unpack, objects can be either packed or unpacked, and only the invariant of unpacked objects can be broken.

In this paper we propose a much stricter invariant protocol: the invariant of every observable object must hold. We formally define *observable* later in the paper, but for example it requires that at the point of a method call, say `a.foo(b)`, the invariant of all the objects in the ROG of the receiver and all the arguments (`a` and `b`) must hold.

Note that this is much stronger than just saying that the invariant should hold every time an object is actually observed (for example, every time a field is accessed). This is still more flexible that Refinement types [**?**]: objects that are not visible in scope can be be broken; this of course includes objects ready for garbage collections.

This stricter invariant protocol would clearly support easier reasoning; however at a first glance it may look too restrictive, preventing us to express 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{
  Int min; Int max;//assumed getters and setters
  method Bool invariant(){return min<max;}
  method Void set(Int min; Int max){
    if(min>=max){return;}
    this.min(min);//setters for min/max
    this.max(max);
    }
  }
```

Under the visible state semantic, this code of `set(_,_)` is ok: `min(_)` may temporarily break the invariant, that is fixed the moment after by `max(_)`. It is ok to break the invariant in that point, since we are inside the method `set(_,_)` of Range; thus there is an active method. However, under our stricter approach, we consider this code to be wrong. The moment `this.max(max)` is called, the invariant of `this` may be broken, and the invariant of an observable object can never be broken.

However, we can easily adapt this code and provide a correct modified `Range` class with the desired client interface:

```
class BoxRange{//no invariant in BoxRange
  Int min; Int max;//assumed getters and setters
  BoxRange(Int min,Int max){this.min=min;this.max=max;}
```

```
  method Void set(Int min; Int max){
    if(min>=max){return;}
    this.min(min); this.max(max);
    }
  }
class Range{ BoxRange box;
  static mehtod Range of(Int min,Int max){
    return new Range(new BoxRange(min,max));
    } //factory of BoxRange
  method Bool invariant(){return min<max;}
  method Void set(Int min; Int max){
    return this.box().set(min,max);
    }
  }
```

Now, the code of `Range.set(_,_)` is correct: since `this` $\notin ROG($`this.box()`$)$, the call `BoxRange.set(_,_)` works in an environment where the `Range` object is not observable, thus its invariant can be temporarily broken.

The former example is an illustration of the *box pattern*.[4]

While in very specific situations the overhead of creating such additional box object may be unacceptable, we designed our work for environments where such fine performance differences are negligible.[5]

In the reminder of this work, we discuss how to combine runtime checks, object capabilities and reference capabilities to create a convenient language where our strict invariant protocol can be soundly enforced; even in the presence of mutations, I/O, non determinism and exceptions, all under the open world assumption, when we only need to assume that all code is well typed.

Summary of our contributions

We have fully implemented our protocol in L42[6], we used this implementation to implement and test an interactive GUI involving a class with an invariant. On a test case with 5 objects with an invariant, our protocol performed only 77 invariant checks, whereas the visible state semantic invariant protocols of D and Eiffel perform 53 and 14 million checks (respectively). See Section 7 for an expla-

[4] This pattern is obvious enough that we do not wish to claim it as a contribution of our work, but we are unable to find it referenced with a specific name in literature. Technically speaking, it is a simplification of the Decorator, but with a different goal in mind.

[5] Also, many VMs and compilers allows optimizing away wrapper objects in many circumstances.[**?**]

[6] Our implementation does not actually extend the core L42 language, but is implemented a meta-programming operation that checks that a given class conforms to our protocol, and injects invariant checks in the appropriate places. A suitably anonymised, experimental version of L42, supporting the protocol described in this paper, together with the full code of our case studies, is available at http://l42.is/InvariantArtifact.zip. We also believe it would be easy to implement our protocol in Pony and Gordon *et al.*'s language.

nation of these result. We also compared with Spec#, whose invariant protocol performs the same number of checks as ours, however the annotation burden was almost 4 times higher than ours. In pack/unpack, an object's invariant is checked only by the pack operation. In order for this to be sound, some form of aliasing and/or mutation control is necessary. For example, Spec# [**?**], which follows the pack/unpack methodology, uses a theorem prover, together with source code annotations. While Spec# can be used for full static verification, it conveniently allows invariant checks to be performed at runtime, whilst statically verifying aliasing, purity and other similar standard properties. This allows us to closely compare our approach with Spec#. Since a case study composed by a single program is not very compelling, in the appendix we present many more case studies.

In this paper we argue that our protocol is not only more succinct than the pack/unpack approach, but is also easier and safer to use. Moreover, our approach deals with more scenarios than most prior work: we allow sound catching of invariant failures and also carefully handle non-deterministic operations like I/O. Section 2 explains the pre-existing *type modifier* features we use for this work. Section 3 explains the details of our invariant protocol, and Section 4 formalises a language enforcing this protocol. Sections 5 and 6 explain and motivate how our protocol can handle invariants over immutable and encapsulated mutable data, respectively. Section 7 presents our GUI case study and compares it against visible state semantics and Spec#: they performed 5 orders of magnitude more invariant checks, and required 60% more annotations, respectively. Sections 8 and 9 provide related work and conclusions.

Appendix A provides a proof that our invariant protocol is sound. In Appendix B we explore in the detail another case study, and we explains exactly why the Spec# encoding of those examples is so verbose. Appendix C, we designed a worst case scenario for our invariant protocol, where Spec# performed four times less invariant checks, while D and Eiffel performed only twice as many. In Appendix C we also compare with examples from others work on Spec# [**?**,**?**,**?**]; we show why we cannot encode some of their examples: namely when state that an object's invariant depends on can be directly modified by other objects. At first glance, our approach may feel very restrictive; in Appendix D, we show programming patterns demonstrating that these restrictions do not significantly 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. In Appendix E, we discuss more related work on runtime verification.

2 Background on Type Modifiers

Reasoning about imperative object-oriented (OO) programs is a non trivial task, made particularly difficult by mutation, aliasing, dynamic dispatch, I/O, and exceptions. There are many ways to perform such reasoning; instead of using automated theorem proving, it is becoming more popular to verify aliasing and immutability using a type system. For example, three languages: L42 [?,?,?,?], Pony [?,?], and the language of Gordon *et al.* [?] use Type Modifiers (TMs)[7] to statically ensure deterministic parallelism and the absence of data-races. While studying those languages, we discovered an elegant way to enforce invariants: we use TMs to restrict how/when the result of invariant methods may change, this is done by restricting I/O, what state the invariant can refer to, and what can alias/mutate such state.

Type Modifiers (TMs)

TMs, as used in this paper, are a type system feature that allows reasoning about aliasing and mutation. Recently a new design for them has emerged that radically improves their usability; three different research languages are being independently developed relying on this new design: the language of Gordon *et al.* [?], 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 TMs of these languages are used to provide automatic and correct parallelism [?,?,?,?].

Type modifiers are a well known language mechanism [?,?,?,?,?,?] that allow static reasoning about mutability and aliasing properties of objects. Here we refer to the interpretation of [?], that introduced the concept of recovery/promotion. This concept is the basis for L42, Pony, and Gordon *et al.*'s type systems [?,?,?,?,?]. With slightly different names and semantics, those languages all support the following modifiers for object references (i.e. expressions and variables):

- Mutable (`mut`): the referenced object can be mutated, and freely shared/aliased, as in most imperative languages without modifiers. If all types are `mut`, there is no restriction on aliasing/mutation.
- Immutable (`imm`): the referenced object cannot mutate, not even through other aliases. We call an object referenced as `imm`, an *immutable object*. Note that an object may be mutated and then *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 still be observed. Readonly references can refer to both mutable and immutable objects, since `read` is a supertype of both `imm` and `mut`.
- Encapsulated (`capsule`): every non-immutable object in the reachable object graph (ROG) of a capsule reference (including itself) is only reachable through that reference. This means that if a capsule reference r is usable in the same expression as a reference r', then either r' does not refer to an object reachable from r, or r' refers to an immutable object. Note an encapsulated reference can be freely used as either mutable or immutable, since there could have been no other references to it.

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

> In L42, a capsule variable always holds a capsule reference: this is ensured by allowing them to be used only once, thus they are expressed using linear/affine types [?]. Pony and Gordon *et al.* follow a more complicated approach where capsule variables can be accessed multiple times, however the result (which will not be a capsule reference) 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.

TMs are different to field or variable modifiers like Java's final: TMs apply to references, whereas final applies to fields themselves. Unlike a variable/field of a read type, a final variable/field cannot be reassigned, it always refers to the same object, however the variable/field can still be used to mutate the referenced object. On the other hand, an object cannot be mutated through a read reference, however a read variable can still be reassigned.[8]

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

```
 mut Point mp = new Point(1, 2);
 mp.x = 3; // ok
 imm Point ip = new Point(1, 2);
 //ip.x = 3; // type error
read Point rp = mp;
//rp.x = 3; // type error
mp.x = 5; // ok, now we can observe rp.x == 5
ip = new Point(3, 5); // ok, ip is not final
```

There are several possible interpretations of the semantics of type modifiers when applied to fields. Here we assume the full/deep meaning [?,?]:

– 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. In order for access to a capsule fields to always 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 treats capsule fields the same as capsule variables: it does not guarantee that they contain a capsule reference, as it provides non-destructive reads. Pony's capsule fields are still useful for safe parallelism, as destructive reads of a capsule field return a capsule reference (which can then be sent to other actors), however the ROG of a capsule field can be mutated by the same actor, even within methods of unrelated objects. L42 supports a variation of capsule fields similar to Pony's, but does not support destructive reads [?,?].

[8] 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.

These forms of `capsule` fields are useful for safe parallelism but not invariant checking: Pony and L42's existing `capsule` fields do not prevent representation exposure; while Gordon *et al.*'s cannot be read non-destructively, thus they should not be accessible in a invariant method. 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 we believe they could be easily added to Pony and Gordon *et al.*'s language. We repeat here for more clarity: a capsule field is not the same concept of a capsule reference. In all approaches a capsule reference is required to initialize or update a capsule field. However, different languages have different behaviour when a capsule field is accessed, and not always a capsule reference is produced.

Promotion and Recovery

There are many different existing techniques and type systems that handle the modifiers above [?,?,?,?,?]. 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 [?,?,?] the type modifiers, adapting them to their use context. To see a glimpse of this flexibility, consider the following example:

```
    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 syntactically initialised with the same expression: `new Circle(..)`. All `new` expressions return a `mut` [?,?], so `mc` is obviously ok. The declarations of `cc` and `ic` are ok, 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 TMs in recent literature [?,?,?,?,?]. From a usability perspective, this improvement means that these TMs are opt-in: a programmer can write large sections of code mindlessly using `mut` types and be free to have rampant aliasing. Then, at a later stage, another programmer may still be able to encapsulate those data structures into an `imm` or `capsule` reference.

Exceptions

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

SES for unchecked exceptions.[9] L42 enforces SES using TMs in the following way:[10][11]

- 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*. SES allows us to throw invariant failures as unchecked exceptions: if an objects 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 SES guarantees that not object mutated within a try block is visible when it catches an unchecked exception. For the purposes of soundly catching invariant failures, it would be sufficient to enforce SES only when capturing exceptions caused by such failures.

Object Capabilities (OCs)

OCs, which L42, Pony, and Gordon *et al.*'s work have, are a widely used [?,?,?] programming technique where access to resources are encoded as 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 TMs 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 instance methods of classes whose instantiation is kept under control.

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

In contrast, if Java were to respect the object capability style, the `main` method could take a `System` parameter, as in `main(System s) {.. s.in.read() ..}`. Calling methods on that `System` instance would be the only way to perform I/O; moreover, the only `System` instance would be the one created by the runtime system before calling `main`. This design has been explored by Joe-E [?]. 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.

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

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

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

However, since L42 allows user code to perform foreign calls without going through a predefined standard library, its type system enforces the OC pattern over such calls:

- Foreign methods (which have not been whitelisted as deterministic) and methods whose names start with `#$` are *capability methods*.
- Constructors of classes declared as *capability classes* are also capability methods.
- Capability methods can only be called by other capability methods 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 methods, thus they can instantiate capability objects and pass them around to the rest of the program.

L42 expects capability methods to be used mostly internally by capability classes, whereas user code would call normal methods on already existing capability objects.

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) can be whitelisted as 'deterministic', provided they do not affect program execution, such as by non deterministically throwing I/O errors.

Purity

Our TM enforcement of 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. Such methods are pure because:

- the ROG of the parameters (including `this`) is only accessible as `read` (or `imm`), thus it cannot be mutated[12],
- if a capability object is in the ROG of any of the arguments (including the receiver), then it can only be accessed as `read`, preventing calling any non deterministic (capability) methods,
- no other preexisting objects are accessible (as L42 does not have global variables).[13]

We are unsure about the exact details of Gorodn *et al.*'s and Pony's OC style, and if they can be used to enforce purity.

[12] 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.

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

3 Our Invariant Protocol

Our invariant protocol guarantees that the whole ROG of any object involved in execution (formally, in a redex) is *valid*: if you can call methods on an object, calling `invariant` on it is guaranteed to return `true` in a finite number of steps. However, calls to `invariant` that are generated by our runtime monitoring (see below) can access the fields of a potentially invalid `this`. This is necessary to allow for the `invariant` method to do its job: namely distinguish between valid and invalid objects. However, as for calls to any other method, calls to `invariant` written explicitly by users are guaranteed to have a valid receiver.

For simplicity, in the following explanation and in our formalism we require receivers to always be specified explicitly, and require that the receivers of field accesses and updates are always `this`; that is, all fields are instance private. We also do not allow explicit constructor definitions, instead we assume constructors are of the standard form C`(`$T_1 x_1$`,`...`,`$T_n x_n$`) {this.`f_1`=`x_1`;`...`;this.`f_n`=`x_n`;}`, where the fields of C are $T_1 f_1$; ...; $T_n f_n$;. This ensures that partially uninitialised (and likely invalid) objects are not passed around or used. These restrictions only apply to our formalism; our code examples and the L42 implementation soundly relax these, see below for a discussion.

Capsule Fields

To allow invariants over complex (cyclic) mutable objects, we introduce a novel kind of `capsule` field[14], which can be accessed within invariants. To be able to easily detect when an objects invariant could be violated, we define the following rules on `capsule` fields:

- A `capsule` field can only be initialised/updated with a `capsule` expression.
- Access to a `capsule` field on a `mut` receiver will return a `mut`. Since fields are instance private, this access will be on `this` and within a `mut` method. We call such methods *capsule mutators*, they must:
  - use `this` exactly once in their body, namely to access the `capsule` field,
  - have no `mut` or `read` parameters (excluding the `mut` receiver),
  - not have a `mut` return type, and
  - be declared as not throwing any checked exception[15].
- Any other `capsule` field access behaves like a `mut` field access: if the receiver is `imm`, the field access will return `imm`, if the receiver is `read`, it will return `read`, if the receiver is `capsule`, it will return `mut`, which is then immediately promotable to `capsule`.

These restrictions ensure that for all objects o, and `capsule` field's f of that object[16]:

- o is not in the ROG of $o.f$.

[14] Our L42 implementation for our invariant protocol supports these fields by enforcing syntactic restrictions over constructors, getters, setters, and capsule mutators.

[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.

[16] See Appendix A for a proof of these properties.

– When we are not executing a capsule mutator on o that reads f, no object in the ROG of $o.f$ can be seen as `mut` or `capsule`, using any sequence of field accesses on a local variable. Since only a capsule mutator can see $o.f$ as `mut`, this means that the only way to mutate the ROG of $o.f$ is through a capsule mutator on o.
– If execution is (indirectly) in such 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 mutate within a capsule mutator, and only after the single use of o to access $o.f$; such mutation could invalidate the invariant of o, so we simply check it at the end of the method before o can be used again. Provided that the invariant is re-established before returning, no invariant failure will be thrown, even if the invariant was broken *during* the method call.

Rather than allowing the values of such fields to be shared between threads/actors, this new kind of `capsule` field prevents representation exposure, as does the very similar concept of owner-as-modifier [**?**,**?**], where we could consider an object to be the 'owner' of all the mutable objects in the ROG of its `capsule` fields. In particular, our new kind of `capsule` field is primarily intended to be used in invariants; for other uses, one should consider using normal `mut` fields or another kind of `capsule` field, such as those designed for safe parallelism [**?**,**?**,**?**].

Note that 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, unrestricted readonly access to `capsule` fields can be allowed by writing getters of the form `read method read C f() { return this.f; }`. Such getters are already a fundamental part of the L42 language [**?**]. Since `mut` is a subtype of `read`, such a method can be called on a `mut this`, without making the method a capsule mutator.

L42 also supports `capsule` methods: methods with a `capsule this`. They are not considered capsule mutators since `capsule` variables can only be used once. This means that L42 guarantees that `this` will not be reachable from anywhere else including the `capsule` field itself; thus immediately after the single use of `this` to read the `capsule` field, `this` will be garbage collectable.

Invariants

We require that all classes contain a `read method Bool invariant() {..}`, if no `invariant` method is present, a trivial one returning `true` will be assumed. Since `invariant` only takes a `read` parameter (the receiver), it is pure [17], as discussed in Section 2. The bodies of `invariant` methods are limited in their usage of `this`: `this` can only occur as the receiver of a field access to an `imm` or `capsule` field. This restriction ensures that an invalid `this` cannot be passed around. We prevent accessing `mut` fields since their ROG could be changed by unrelated code (see Section 5). Note that we do not require such fields to be

[17] If `invariant` were not pure, it would be nearly impossible to ensure that it would keep returning `true`.

`final`: when a field is updated, we simply check the invariant of the receiver of the update.

Monitoring

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

– After a constructor call, on the newly created object.
– After a field update, on the receiver.
– After a capsule mutator method returns, on the receiver of the method[18].

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

Relaxations

TODO: mix this in the former points, and push far non 42 extensions. Also, make clear that only final methods can be used in invariants. In L42, and our code examples, we allow a couple of sound relaxations:

– `invariant` methods can call instance methods that in turn only use `this` to read `imm` or `capsule` fields, or call other such instance methods. The semantics of such methods must then be reinterpreted in the context of `invariant`, where `this` may be invalid.
– All fields can be allowed to be public, provided that access to a `capsule` field on a `mut` receiver other than `this` is typed as `read`. However, even without this relaxation getters and setters could be used to simulate public fields.

If we were to extend L42 to support user written constructors or traditional sub-classing: In our examples, we allow user written constructors, provided that `this` is only used as the receiver of field initialisations. L42 itself does not support user-written constructors, instead one would just write a static factory method that behaves equivalently.

To apply our invariant protocol to a language with traditional sub-classing, such as Gordon*et al.*'s, invariant methods of a sub-class would implicitly start with a check that `super.invariant()` returns `true`. In addition, invariant methods of non-final classes should also be prevented from calling non-final methods on `this`, so that a subclass can't override such a method to access non `imm` or `capsule` fields. Note that invariant checks would not be performed at the end of `super(..)` constructor calls, but only at the end of `new` expressions, as happens in [**?**].

We do not allow the above relaxations in our formalism as they would make the proof more complicated, without making it more interesting.

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

4 Formal Language Model

In order to model our system, we need to formalise an imperative object-oriented language with exceptions, object capabilities, and rich type system support for TMs 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 literature [**?**,**?**,**?**,**?**,**?**]. Thus we are going to assume 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 [**?**] and Featherweight Java [**?**], and assume:

- An implicit program/class table; we use the notation $C.m$ to get the method declaration for m, within class C, similarly we use $C.f$ to get the declaration of field f, and $C.i$ to get the declaration of the i^{th} field.
- Memory, $\sigma : l \rightarrow C\{\overline{v}\}$, 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.
- A main expression that is reduced in the context of such 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 type-modifiers 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 *error*s, 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 exceptions,

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

Fig. 1. Grammar

formalising exceptions in this way would not cause any interesting variation of our proof.

Grammar

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

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

- `invariant` methods and capsule mutators satisfy the restrictions in Section 3.
- Field accesses and updates in methods are of the form $\texttt{this}.f$ or $\texttt{this}.f \texttt{ = } e$, respectively.
- Field accesses and updates in the main expression are of the form $l.f$ or $l.f \texttt{ = } e$, respectively.
- Method bodies do not contain any l or $\texttt{M(\_;\_;\_)}$ expressions.

Reduction rules

Our reduction rules are defined in Figure 2. They are pretty 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

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

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

(MCALL)
$$\frac{}{\sigma|l.m\mathtt{(}v_1, \ldots, v_n\mathtt{)} \to \sigma|e'[\mathtt{this} := l, x_1 := v_1, \ldots, x_n := v_n]}$$
$\sigma(l) = C\{\_\}$
$C.m = \mu\ \mathtt{method}\ T\ m\mathtt{(}T_1\ x_1 \ldots T_n\ x_n\mathtt{)}\ e$
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)
$$\frac{}{\sigma|\mathtt{M(}l\mathtt{;}v\mathtt{;true)} \to \sigma|v}$$

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

(TRY ENTER)
$$\frac{}{\sigma|\mathtt{try}\ \{e_1\}\ \mathtt{catch}\ \{e_2\} \to \sigma|\mathtt{try}^{\sigma}\{e_1\}\ \mathtt{catch}\ \{e_2\}}$$

(TRY OK)
$$\frac{}{\sigma'|\mathtt{try}^{\sigma}\{v\}\ \mathtt{catch}\ \{\_\} \to \sigma'|v}$$

(TRY ERROR)
$$\frac{}{\sigma'|\mathtt{try}^{\sigma}\{\mathit{error}\}\ \mathtt{catch}\ \{e\} \to \sigma'|e}$$

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

Fig. 2. Reduction rules

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 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 $\sigma_0|e_0 \to \sigma|e\}$

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

$\mathit{valid}(\sigma, l)$ 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 the call to `invariant` itself, and any field accesses inside its evaluation. Note that this only applies to single small step reductions, and not the entire evaluation of `invariant`.

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

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

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

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

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

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

5 Invariants Over Immutable State

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

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

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

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

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

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

Unrestricted Access To Capability Objects

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

```
class EvilString extends String {
  @Override read method Bool isEmpty() {
    // Create a new capability object out of thin air
    return new Random().bool(); }}
...
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, the above call to it is not. Obviously this breaks any reasoning and would make our protocol unsound. In particular, note how in the presence of dynamic class loading, we have no way of knowing what the type of `name` could be. Since our system allows non-determinism only through capability objects, and restricts their creation, the above example would be prevented.

Allowing Internal Mutation Through Back Doors

Suppose we relax our rules by allowing interior mutability as in Rust and Javari, allowing the ROG of an 'immutable' object to be mutated through back doors. Such back doors would allow the `invariant` method to store and read information about previous calls. For example `MagicCounter` breaks determinism by remotely breaking the invariant of `person` without any interaction with the `person` object itself:

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

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

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 [?]). In addition, since unchecked exceptions are immutable, they can not contain a `read` reference to any object (such as the `this` reference seen by `invariant` methods). These two properties ensure that an object whose invariant fails will be unreachable after the invariant failure has been captured. If instead we were to not enforce strong exception safety, an invalid object could be made reachable:

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

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

6 Invariants over encapsulated state

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

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

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

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

As you can see, it would be possible for external code with no knowledge of the `ShippingList` to mutate its items.[19]

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

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

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

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

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

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

Allowing `this` to be used more than once can also cause problems; if the following code were accepted, `this` may be reachable from `f`, thus `f.hi()` may observe an invalid object.

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

In order to ensure that a second reference to `this` is not reachable through the parameters, we only accept `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 imm Void addHeavy(read Foo f) {
  this.items.add(new HeavyItem())
  f.hi() } // Can 'this' be observed here?
...
mut ShippingList l = new ShippingList(new Items());
read Foo f = new Foo(l);
l.addHeavy(f); // We pass another reference to ‘l’ through f
```

7 GUI Case study

Here we show that we are able to verify classes with circular mutable object graphs, that interact with the real world using I/O. Our case study involves a GUI with containers (`SafeMovables`) and `Buttons`; the `SafeMovable` class has an invariant to ensure that its children are completely contained within it and do not overlap. The `Buttons` 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 `SafeMovables` and `Buttons` implement `Widget`. Crucially, since the children of `SafeMovable` are stored in a list of `Widgets` it can contain other `SafeMovables`, 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[20], 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); }}
  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 event) { this.outer.l += 1; }}
...
// main expression; #$ is a capability method 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 circular: since the `MoveAction`s inside `Button`s need a reference to the containing `Box` in order to move it. Even though the children of `SafeMovable`s are fully encapsulated, we can still easily dispatch events to them using `dispatch`. Once a `Button` receives an `Event` with a matching ID, it will call its `Action`'s `process` method.

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

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

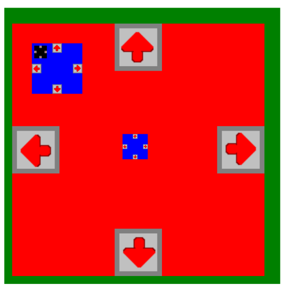
The Invariant

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

```
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` only uses `this` to access its `imm` and `capsule` fields.

Our Experiment



As shown in the figure to the left, counting both `SafeMovables` and `Buttons`, our main method creates 21 widgets: a top level (green) `SafeMovable` without buttons, containing 4 (red, blue, and black) `SafeMovables` 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 `Widgets` is 5. Our main method automatically presses each of the 16 buttons once. In L42, using the approach of this paper, this resulted in 77 calls to `SafeMovable`'s invariant.

Comparison With Visible State Semantics

As an experiment, we set our implementation to generate invariant checks following the visible state semantics approaches of D and Eiffel [?,?], where the invariant of the receiver is instead checked at the start and end of *every* public (in D) and qualified[21] (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', 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

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

SafeMovable will call the width, height, left, and top methods of its children, which may themselves be SafeMovables, and hence such calls may invoke further invariant checks. Note that width and height are simply getters for fields, whereas the other two are non trivial *methods*.

Spec# Comparison

We also encoded our example in Spec#[22], which like L42, statically verifies aliasing/ownership properties, as well as the admissibility of invariants. 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 [?]), which only gave us 2 warnings[23]: that the invariant of SafeMovable was not known to hold at the end of its constructor and dispatch method. Like our system however, Spec# checks the invariant at those two points at runtime. Thus the code is equivalently verified in both Spec# and L42; in particular it performed exactly the same number (77) of runtime invariant checks.[24]

We found it quite difficult to encode the GUI in Spec#, due to its unintuitive and rigid ownership discipline. In particular we needed to use many more annotations, which were larger and had greater variety. In the following table we summarise the annotation burden[25], for the *program* that defines and displays the SafeMovables and our GUI; as well as the *library* which defines Buttons, Widget, and event handling.[26]:

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

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

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

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

[23] 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.

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

[25] 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.

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

The Spec# code also required us to deviate from the style of code we showed in our simplified version: we could not write a usable `children` method in `Widget` that returns a list of children, instead we had to write `children_count()` and `children(int i)` methods; we also needed to create a trivial class with a `[Pure]` constructor (since `Object`'s one is not marked as such). In contrast, the only strange thing we had to 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.

The Box Pattern

We have found that using an inner `Box` object, is quite a useful pattern in static verification: where one encapsulates all relevant mutating state into an encapsulated subobject which is not exposed to users. Both our L42 and Spec# code required us to use the box pattern for our `SafeMovable`, due to the circular object graph caused by the `Action`s of `Button`s needing to change their enclosing `SafeMovable`'s position. In Appendices C and D, we show how the box pattern can also be used to pass the state of invalid objects around, and batch together complex mutations and multiple field updates, with only a single invariant check. Appendix D also shows a 'transformer' pattern, were we can allow the children of `Widgets` to be mutated by arbitrary code, albeit with restrictions.

8 Related Work[27]

Type Modifiers

We rely on a combination of modifiers that are supported by at least 3 languages/lines of research: L42 [?,?,?,?], Pony [?,?], and Gordon *et al.* [?]. Those approaches all support deep/strong interpretation, without back doors. Former work [?,?,?,?,?], which eventually enabled the work of Gordon *et al.*'s, 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 TMs approaches like Javari [?,?] and Rust [?] are unsuitable; they model weaker properties and provide back doors which are not easily verifiable as being used properly. Many approaches just try to preserve purity (as for example [?]), but here we also need aliasing control. Ownership [?,?,?] is another popular form of aliasing control that can be used as a building block for static verification [?,?].

Object Capabilities

In literature, OCs are used to provide a wide range of guarantees, and many variations are present. Object capabilities [?], in conjunction with type modifiers, are able to enforce purity of code in a modular way, without requiring the use of monads. L42 and Gordon use OCs simply to reason about I/O and non-

[27] See Appendix E for related work on runtime verification.

determinism. This approach is best exemplified by Joe-E [?], 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 with only final fields that refer to immutable classes. In Joe-E, every method that only takes instances of immutable classes is pure. Thus their model would not allow the verification of purity for invariant methods of mutable objects. In contrast our model has a more fine grained representation of mutability: it is *reference-based* instead of *class-based*. In our work, every method taking only `read` or `imm` *references* is pure, regardless of their class type; in particular, we allow the parameter of such a method to be mutated later on by other code.

Class invariant protocols

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

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

- The invariants of objects in a *steady* state are known to hold: that is when execution is not inside any of the objects' public methods [?]. 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 [?,?]. Some approaches ensure the invariant of the receiver of the *calling* method, rather than the *called* method [?]. JML [?] 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. non instance private) methods, and only for 'qualified' (i.e. not `this`) calls [?].
- 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' [?].
- Or, as in this work, the invariant of any object which could be *involved* in execution is assumed to hold. It is checked after every modification of the object or the ROG of its `capsule` fields.

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

Security and Scalability

Our approach allows verifying an object's invariant independently of the actual invariants of other objects. 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. This is less problematic with a type system, since its properties are more coarse grained, simpler and easier to check. Static verification has more flexible and fine-grained annotations and often relies on a fragile theorem prover as a backend.

To soundly verify code embedded in an untrusted environment, as in gradual typing [?,?], it is possible to consider a verified core and a runtime verified boundary. You can see our approach as an extremely modularized version of such system: every class is its own verified core, and the rest of the code could have Byzantine behaviour. Our formal proofs show that every class that compiles/type checks is soundly handled by our protocol, independently of the 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 [?]: in his short paper he verified a property of the `Subject/Observer` pattern. However, the proof relies on (any override of) the `Subject.register(Observer)` method respecting its contract. Such assumption is unrealistic in a real-world system with dynamic class loading, and could trivially be broken by a user-defined `EvilSubject`.

Static Verification

Spec# [?] 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 [?] 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 [?], where an invariant only depends on objects it owns; or *visibility* based [?,?], where an invariant may depend on objects it doesn't own, provided that the class of such objects know about this dependence. Unlike our approach, Spec# does not restrict the aliases that may exist for an object, rather it restricts object mutation: an object cannot be modified if the invariant of its owner is required to hold. This is more flexible than our approach as it also allows only part of an object's ROG to be owned/encapsulated. However as we showed in Section 7, it can become much more difficult to work with and requires significant annotation, since merely having an alias to an object is insufficient to modify it or call 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. [?]

Another system is AutoProof [?], 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 [?] 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.

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

9 Conclusions and Future Work

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

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

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

The language we presented here restricts the forms of `invariant` and capsule mutator methods; such strong restrictions allow for sound and efficient injection of invariant checks.

In order to obtain safety, simplicity, and efficiency we traded some expressive power: the `invariant` method can only refer to immutable and encapsulated state. This means that while we can easily verify that a doubly linked list of immutable elements is correctly linked up, we can not do the same for a doubly linked lists of mutable elements. Our approach does not prevent correctly implementing such data structures, but the `invariant` method would be unable to access the list's nodes, since they would contain `mut` references to shared objects. Our restrictions do not get in the way of writing invariants over immutable data, but the box pattern is required for verifying complex mutable data structures. We believe this pattern, although verbose, is simple and understandable. While it may be possible for a more complex and fragile type system to reduce the need for the pattern whilst still ensuring our desired semantics, we prioritize simplicity and generality.

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

Other directions that could be investigated to improve our work include the addition of syntax sugar to ease the burden of the box pattern, as well as type modifier inference.

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 $\Sigma; \Gamma; \blacksquare.m(l) \vdash l : \texttt{mut}\ C$ and $\Sigma; \Gamma; l.m(\blacksquare) \nvdash l : \texttt{mut}\ C$, 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:

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

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

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

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

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

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:

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

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

Now we can define what it means for an l to be *mutatable*[28] 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:

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

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

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

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

---

[28] 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.

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 $\mathit{validState}(\sigma, \mathcal{E}[e])$, 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 modifier, 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 $\Sigma; \Gamma; \mathcal{E}[\mathtt{new}\ C\mathtt{(}e_1, ..., e_{i-1}, \blacksquare, e_{i+1}, ..., e_n\mathtt{)}] \vdash e_i : T_i$.
2. If $\Sigma; \Gamma; \mathcal{E}[\blacksquare.f \mathtt{=} e'] \vdash e : \_C$ and $C.f = T'\ f$, then $\Sigma; \Gamma; \mathcal{E}[e.f \mathtt{=} \blacksquare] \vdash e' : T'$.
3. If $\Sigma; \Gamma; \mathcal{E}[\blacksquare.m\mathtt{(}e_1, ..., e_n\mathtt{)}] \vdash e : \_C$ and $C.m = \mu\,\mathtt{method}\ T\ m\mathtt{(}T_1\,x_1, ..., T_n x_n\mathtt{)}$ _, then:
   (a) $\Sigma; \Gamma; \mathcal{E}[\blacksquare.m\mathtt{(}e_1, ..., e_n\mathtt{)}] \vdash e : \mu\, C$,
   (b) $\Sigma; \Gamma; \mathcal{E}[e.m\mathtt{(}e_1, ..., e_{i-1}, \blacksquare, e_{i+1}, ..., e_n\mathtt{)}] \vdash e_i : T_i$, and
   (c) $\Sigma; \Gamma; \mathcal{E} \vdash e.m\mathtt{(}e_1, ..., e_n\mathtt{)} : T$.

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

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

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

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

Now we define formal properties about our TMs, 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 $\mathit{validState}(\sigma, e)$ and $\mathit{immutable}(\sigma, e, l)$, then not $\mathit{mutatable}(\sigma, \blacksquare, e, l)$.
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 $\mathit{validState}(\sigma, \mathcal{E}_v[e])$,
not $\mathit{mutatable}(\sigma, \mathcal{E}_v, e, l)$, and $\sigma|\mathcal{E}_v[e] \rightarrow^+ \sigma'|\mathcal{E}_v[e']$, then not $\mathit{mutatable}(\sigma', \mathcal{E}_v, e', l)$.

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

Assumption 6 (Capsule Consistency)

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

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

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

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

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

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 $\Gamma(x) = \texttt{read}\ \_$, then $\Sigma;\Gamma;\mathcal{E} \not\vdash x : \texttt{mut}\ \_$.
2. If $\Sigma;\Gamma;\mathcal{E}[\blacksquare.m(\overline{e})] \vdash e : \_C$ and $C.m = \_\texttt{method}\ \texttt{read}\,C'\ \_$, then $\Sigma;\Gamma;\mathcal{E} \not\vdash e.m(\overline{e}) : \texttt{mut}\ \_$.

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 $validState(\sigma', \mathcal{E}[\texttt{try}^{\sigma_0}\{e_0\}\ \texttt{catch}\ \{e_1\}])$, then

$\forall l \in dom(\sigma_0)$, not $mutatable(\sigma, \mathcal{E}[\texttt{try}^{\sigma_0}\{\blacksquare\}\ \texttt{catch}\ \{e_1\}], e_0, l)$.

We use SES 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 $e = \mathcal{E}[\texttt{try}^{\sigma_0}\{\mathcal{E}[\texttt{M}(l;\_;\_)]\}\ \_])$, 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 4) to prove that the `invariant` method is deterministic and does not mutate existing memory:

Lemma 2 (Determinism) If $\mathit{validState}(\sigma, \mathcal{E}_v[l\texttt{.invariant()}])$ 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 $\forall l' \in \mathit{dom}(\sigma)$, not $\mathit{mutatable}(\sigma'', \mathcal{E}_v, e'', l)$.

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 $\forall l' \in \mathit{dom}(\sigma)$, not $\mathit{mutatable}(\sigma', \mathcal{E}_v, e', l)$; 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, $c \in \mathit{dom}(\sigma)$, 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:

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

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

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

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

$\mathit{wellEncapsulated}(\sigma, e, l)$ iff $\forall f \in \mathit{capsuleFields}(\sigma, l)$, not $\mathit{mutatable}(\sigma \backslash l, \blacksquare, e, \sigma[l.f])$.

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`:

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

- if $e = \mathcal{E}[\texttt{M(}l\texttt{;}e'\texttt{;}\_\texttt{)}]$, then $e' = l$, and

- if $e = \mathcal{E}[l.f]$, $f \in \mathit{capsuleFields}(\sigma, l)$, and $\Sigma^\sigma; \emptyset; \mathcal{E}[\blacksquare.f] \not\vdash l : \texttt{capsule}\ \_$, then $\Sigma^\sigma; \emptyset; \mathcal{E} \not\vdash l.f : \texttt{mut}\ \_$.

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

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

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

Theorem 2 (Capsule Field Soundnes). If $\mathit{validState}(\sigma, e)$ then $\forall l$, if $\mathit{reachable}(\sigma, e, l)$, then:

capsuleNotCircular(σ, 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 $\sigma = c \mapsto \texttt{Cap}\{\}$, 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*, $l \notin \mathit{rog}(\sigma, v_i) = \mathit{rog}(\sigma', \sigma'[l.f])$; 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 $f \in \mathit{capsuleFields}(\sigma, l)$: by Mut Update, we have that l is *mutatable*, so by Type Consistency and Capsule Consistency, $\mathit{encapsulated}(\sigma, \mathcal{E}_v[l.f = \blacksquare], v)$, hence l is not *reachable* from v, and so after the update, *capsuleNotCircular* still holds for l.
      - Now consider any l' and $f' \in \mathit{capsuleFields}(\sigma, l')$, 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 $l \notin \mathit{rog}(\sigma, \sigma[l'.f'])$, 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 $l' \notin \mathit{rog}(\sigma, v)$, 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 $\mathcal{E}_v = \mathcal{E}_v{}'[\mathtt{M(}l\mathtt{;}\mathcal{E}[l.f]\mathtt{;}\_\mathtt{)}]$, 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 subexpression where it wasn't allready *reachable*, so we can't have violated *headNotObservable* for any other l'.

   (b) (MONITOR EXIT) $\sigma|\mathcal{E}_v[\mathtt{M(}l\mathtt{;}v\mathtt{;true)}] \rightarrow \sigma|\mathcal{E}_v[v]$:
   - 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 $\mu \neq \mathtt{mut}$:
       * If $\mu = \mathtt{capsule}$, 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, $\mu \in \{\mathtt{read}, \mathtt{imm}\}$, by Read Consistency, Imm Consistency, and Mut Consistency, we have that v is not *mutatable*.

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

   (c) (TRY ERROR) $\sigma|\mathcal{E}_v[\mathtt{try}^{\sigma_0}\{\mathit{error}\}\ \mathtt{catch}\ \{e\}] \rightarrow \sigma|\mathcal{E}_v[e]$, where $\mathit{error} = \mathcal{E}_v{}'[\mathtt{M(}l\mathtt{;}\_\mathtt{;}\_\mathtt{)}]$:
   By our reduction rules, we were previously in state $\sigma_0|\mathcal{E}_v[\mathtt{try}\ \{e_0\}\ \mathtt{catch}\ \{e\}]$. By Unmonitored Try, $l \notin \mathit{dom}(\sigma_0)$, and so l was not *reachable* from $\mathcal{E}_v[\mathtt{try}\ \{e_0\}\ \mathtt{catch}\ \{e\}]$. 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*(σ, 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 $\Sigma^\sigma; \emptyset; \mathcal{E}_v[\mathcal{E}[\blacksquare.f]] \vdash l$: `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*(σ, 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*(σ, 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*$(\sigma' \setminus l, \blacksquare, \mathcal{E}_v[$`M(`$l$`;`$l$`;`$l$`.invariant())`$], v_i)$; thus *wellEncapsulated* holds for l and each of its `capsule` fields.

   (b) (UPDATE) $\sigma|\mathcal{E}_v[l.f$ `=` $v] \rightarrow \sigma[l.f = v]|\mathcal{E}_v[$`M(`$l$`;`$l$`;`$l$`.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 $f' \in \mathit{capsuleFields}(\sigma, l')$, with $l'.f' \neq l.f$; by the above UPDATE case for *capsuleNotCircular*, $l \notin \mathit{rog}(\sigma, \sigma[l'.f'])$. If f was a `mut` or `capsule` field, by Type Consistency and Capsule Consistency, v was `mut`, so by *wellEncapsulated*, $v \notin \mathit{rog}(\sigma, \sigma[l'.f'])$; thus we can't have made $\mathit{rog}(\sigma, \sigma[l'.f'])$ *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 $f \in \mathit{capsuleFields}(\sigma, l)$, by Mut Access, either $\Sigma^\sigma; \emptyset; \mathcal{E}_v \nvdash \sigma[l.f] : \mathtt{mut}\ \_$ or $\Sigma^\sigma; \emptyset; \mathcal{E}_v[\blacksquare.f] \vdash l : \mathtt{capsule}\ \_$. 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* subexpression:

$\mathit{monitored}(e, l)$ iff $e = \mathcal{E}_v[\mathtt{M(}l\mathtt{;}e_1\mathtt{;}e_2\mathtt{)}]$ 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 $\mathit{reachable}(\sigma, e, l)$ then $\mathit{valid}(\sigma, l)$ or $\mathit{monitored}(e, l)$.

Proof. We will prove this inductively, in a similar way to how we proved Capsule Field Soundness. In the base case, we have $\sigma = c \mapsto \mathtt{Cap}\{\}$, 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 *rog*s (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) $\sigma|\mathcal{E}_v[\texttt{try}^{\sigma_0}\{error\}\texttt{ catch }\{e\}] \rightarrow \sigma|\mathcal{E}_v[e]$, 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 $reachable(\sigma, e, l)$, $\Sigma^\sigma(l).f = \mathtt{imm}\ \_$, then $l \notin rog(\sigma, \sigma[l.f])$.

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

1. Consider any pre-existing *reachable* l and f with $\Sigma^\sigma(l).f = \mathtt{imm}\ \_$, 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 $e = \mathcal{E}_v[\mathtt{new}\ C(v_1, ..., v_n)]$ and each i with $C.i = \mathtt{imm}\ \_$. 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, $monitored(e, l)$, $e = \mathcal{E}[\mathtt{M}(l;e_1;e_2)]$, and either:

- $\mathcal{E}_v = \mathcal{E}[\mathtt{M}(l;\mathcal{E}';e_2)]$, that is r_l (which by definition cannot equal l) was found inside of e_1, this contradicts the definition of *monitored*, or
- $\mathcal{E}_v = \mathcal{E}[\mathtt{M}(l;e_1;\mathcal{E}')]$, and thus r_l was found inside e_2. By our reduction rules, all monitor expressions start with $e_2 = l.\mathtt{invariant}()$; if this has yet to be reduced, then $\mathcal{E}'[r_l] = l.\mathtt{invariant}()$, 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*.

B The Hamster Example in Spec#

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

The `Point` Class

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

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

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

The `Hamster` Class

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

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

Though this is legal in Spec#, it is practically useless. Spec# has no way of knowing whether `pos` is *valid* or *consistent*. If `pos` is not known to be valid, one will be unable to pass it to almost any method, since by default methods implicitly require their receivers and arguments to be valid (compare this with our invariant protocol, which guarantees that any reachable object is valid). If `pos` is not known to be consistent, one will be unable to mutate it, by updating one of its fields or by passing it as an argument (or receiver) to a non-`Pure` method. Though we don't want `pos` to ever mutate, Spec# currently has no way of enforcing that an *instance* of a non-immutable class is itself immutable[29], as such we will simply refrain from mutating it.

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

To simplify instantiation of `Hamster`s, their constructors will take unowned `Point`s; Spec# will then automatically make such `Point` a peer. This is achieved by taking a `[Captured] Point` in the constructor (note how similar this is to taking a `capsule Point`). Note that unlike our system, this prevents multiple

[29] There is a paper [**?**] 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 TM system we use, which allows an initially mutable object to be promoted to immutable.

Hamsters from sharing the same Point, unless both Hamsters have the same owner, if Point were immutable, there would be no such restriction.

With the aforementioned modifications, the Hamster becomes:

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

If however, we did want Point to be an immutable/value type, the original unannotated version would not have any problems.

The Cage Class

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

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

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

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

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

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

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

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

Defining Equality of `Points`

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

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

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

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

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

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

C More Case Studies

Family

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

```
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`, after all the `parents` and `children` have been updated. Thus `Family.processDay` is atomic: it updates all its contained `Persons` together. Had we instead made the `parents` and `children` `capsule` fields of `Family`, the invariant would be required to also hold between modifying the two lists. This could cause problems if, for example, a child was updated before their parent.

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

```
// 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 `Persons`.

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[30] 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` and `addChild` call (for both `Person` and `Family`).

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

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

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

Spec# relies on the absence of arithmetic overflow, and performs runtime checks to ensure this[31], as such the verifier concludes that the field increment in `processDay` cannot break the invariant. Spec# is able to avoid some invariant checks in this case by relying on all arithmetic operations performing runtime overflow checks; whereas integer arithmetic in L42 has the common wrap around semantics.

The annotations we had to add in the Spec# version[32] were similar to our previous examples, however since the fields of `Person` all have immutable classes/types, we only needed to add the invariant itself. The `Family` class was similar to our `Cage` example (see Section 1), however in order to implement the `addChild` method we were forced to do a shallow clone of the new child (this also caused a couple of extra runtime invariant checks). Unlike L42 however, we did not need to create a box to hold the `parents` and `children` fields, instead we wrapped the body of the `Family.processDay` method in an `expose (this)` block. In total we needed 16 annotations, worth a total of 45 tokens, this is worse than the code following our approach that we showed above, which has 14 annotations and 14 tokens.

Spec# Papers

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

- *Verification of Object-Oriented Programs with Invariants:* [**?**] this paper introduces their methodology. In their examples section (pages 41–47), they show how their methodology would work in a class heirarchy with `Reader` and `ArrayReader` classes. The former represents something that reads characters, whereas the latter is a concrete implementation that reads from an owned array. They extend this further with a `Lexer` that owns a `Reader`, which it uses to read characters and parse them into tokens. They also show an example of a `FileList` class that owns an array of filenames, and a `DirFileList` class that extends it with a stronger invariant. All of these examples can be represented in L42[33]. 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,

[31] Runtime checks are enabled by a compilation option; when they fail, unchecked exceptions are thrown.

[32] The Spec# code is in the artifact.

[33] Our encodings are in the artifact.

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:* [?] 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[34] (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:* [?] this paper shows how one can verify invariants over interacting objects, where neither owns/contains the other. They have multiple examples which utilise the ‘subject/observer’ pattern, where a ‘subject’ has some state that an ‘observer’ wants to keep track of. In their `Subject/View` example, `Views` are created with references to `Subjects`, and copies of their state. When a `Subject`’s state is modified, it calls a method on its attached `Views`, 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,

[34] 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.

because the interacting objects are not immutable or encapsulated by each other.

D Patterns

In Section 7 and Appendix C 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` TM (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 {
  // Here ImmList<T> represents a list of immutable Ts.
  mut method ImmList<Transfer> compute(); }

class Transfer { Int money;
  // An 'AccountBox' is like an 'invalid Account':
  //   'that' need not have income > expenses
  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; }
class Account {
  capsule AccountBox box; mut Person holder;
  read method Bool invariant() {
    return this.box.income > this.box.expenses; }

  // 'h' could be aliased elsewehere 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. 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.
- Perform multiple state updates with only a single invariant check: the loop in `transferInner` 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 its invariant: if you have an `Account`, you can be sure that its `income > expenses`, but not if you have an `AccountBox`.
- Wrap normal methods over capsule mutators: `transfer` is not a capsule mutator, so it can use `this` multiple times and take a `mut` parameter.

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.

The Transform Pattern

Recall the GUI case study in Section 7, 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` 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` method in `Widget`, and a `Transformer` interface like so:[35]

```
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` 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); }
```

E Related Work on Runtime Verification Tools

By looking to a survey by Voigt *et al.* [?] and the extensive MOP project [?], 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 [?]. 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 [?] lets you write Python conditions that are verified on a set of Python objects, but the programmer needs to be able to predict which objects are in need of being checked and to use a simpler domain specific language to target them. Hence if a programmer makes a mistake while using this domain specific language, invariant checking will not be triggered. Some

[35] A more general transformer could return a generic `read R`.

tools are intentionally unsound and just perform invariant checking following some heuristic that is expected to catch most failures: such as jmlrac [?] and Microsoft Code Contracts [?].

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

Our invariant protocol is much stronger than visible state semantics, and keeps the invariant under tight control. Gopinathan *et al.*'s. [?] 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 TMs to enforce strong exception safety is a much simpler alternative, providing the same level of safety, albeit being more restrictive (namely that if the operation did succeed it is still effectively rolled back).

Chaperones and impersonators [?] lifts the techniques of gradual typing [?,?,?] 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. Aspect oriented systems like Jose [?], similarly wrap invariant checks around method bodies.