Using Capabilities for Strict Runtime Invariant Checking

Isaac Oscar Gariano, Marco Servetto*

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

Alex Potanin

The Australian National University, Canberra, 2600, ACT, Australia

Abstract

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

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

1. Introduction

Representation invariants (sometimes called class invariants, object invariants, or type refinements) are a useful concept when reasoning about software correctness, particularly with Object Oriented (OO) languages. Such invariants are predicates on the state of an object and its reachable object graph (ROG). They can be presented as documentation, checked as part of static verification, or, as we do in this paper, monitored for violations using runtime verification. In our system, a class specifies its invariant by defining a method called `invariant()` that returns a Boolean. We say that an object's invariant holds when its `invariant()` method would return **`true`**.[1] In a purely functional setting, the programmer only needs to write the code for the invariant check itself, then the runtime needs to call this code each time a value/object is created (or in the case of refinement types, converted to such a type).

In an impure setting, like most OO languages, operations on data structures are often implemented as complex sequences of mutations, where the invariant is temporarily broken. To support this behaviour, most invariant protocols present in the literature allow invariants to be broken and observed broken. The two main forms of invariant protocols are *visible state semantics* [2] and the *Pack-Unpack/Boogie methodology* [3]. In visible state semantics, invariants can be broken when a method on the object is active (that is, currently executing). Some interpretations of the visible state are more permissive, requiring the invariants of receivers to hold only before and after every public method call, and after constructors. In the pack-unpack approach,

*Corresponding Author

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

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

objects are either in a 'packed' or 'unpacked' state, the invariant of 'packed' objects must hold, whereas unpacked objects can be broken. To complicate matters further, OO languages often permit rampant aliasing of mutable state, thus any mutation may inadvertently break the invariant of an arbitrary object.

In this paper we propose a much stricter invariant protocol: at all times, the invariant of every object involved in execution must hold; thus they can be broken when the object is not (currently) involved in execution. An object is *involved in execution* when it is in the reachable object graph of any of the objects mentioned in the method call, field access, or field update that is about to be reduced; we state this more formally later in the paper.

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

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

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

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

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

The code of `Range.set(min,max)` does not violate our protocol. The call to `BoxRange.set(min,max)` works in a context where the `Range` object is unreachable, and thus not involved in execution. That is, the `Range` object is not in the reachable object graph of the receiver or the parameters of `BoxRange.set(min,max)`. Thus `Range.set(min,max)` can temporarily break the `Range`'s invariant. By using the `box` field as an extra level of indirection, we restrict the set of objects involved in execution while the state of the object `Range` is modified. [2]

[2] Due to its simplicity and versatility, we do not claim this pattern to be a contribution of our work, as we expect others

With appropriate type annotations (simple keywords attached to fields, method receivers, paramaters, and return types), the code of `Range` and `BoxRange` is accepted as correct by our system: no matter how `Range` objects are used, a broken `Range` object will never be involved in execution. In particular, our system ensures that the `Range.set` method cannot pass `this`, or an alias to `this`, to the `box.set` method.[3]

Contributions

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

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

We formalise and prove our approach sound, and have fully implemented our protocol in L42[4], and used it to run our various case studies. It is important to note that unlike most prior work, we soundly handle catching of invariant failures and I/O.

The remainder of this paper proceeds as follows:

- Section 2 explains background information necessary to understand our approach.
- Section 3 fully explains our novel invariant protocol, and our novel field kind for mutable data.
- Section 4 demonstrates why the soundness of our protocol depends on the properties of the type system of L42, and similar languages.
- Section 5 formalises our runtime invariant checking, and what it means to soundly enforce our invariant protocol.
- Section 6 contains many case studies, showing that our protocol is more succinct than the pack/unpack approach and much more efficient then the visible state semantic.
- Section 7 shows how our approach does not hamper expressiveness, by showing programming patterns that can be used to perform batch mutation operations with a single invariant check, and how the state of a 'broken' object can be safely passed around.
- Section 8 summarises how we have implemented our protocol in L42.
- Section 9 presents related work, and Section 10 concludes.

---

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

[3] Note that our system does not require that the `min` and `max` fields use primitive immutable number types, in fact, they could store complex (and possibly cyclic) mutable data; our system will ensure that this data can only be mutated within the `Range.set` method (or other similar methods within the enclosing `Range`). We would like to keep the language as minimal as possible, and so we have opted to not add a separate concept of immutable *classes*.

[4] Our implementation works by checking that a given class conforms to our protocol, and injecting invariant checks in the appropriate places. An anonymised version of L42, supporting the protocol described in this paper, together with the full code of our case studies, is available at `http://l42.is/InvariantArtifact.zip`.

- Appendix A formally specifies the properties a type system needs to guarantee, and proves the formalism in Section 5 sound.
- Appendix B presents a simple L42-inspired type system and proves that it satisfies the requirements in Appendix A.

2. Background on Reference and Object Capabilities

Reasoning about imperative OO programs is a non-trivial task, made particularly difficult by mutation, aliasing, dynamic dispatch, I/O, and exceptions. There are many ways to perform such reasoning; instead of using automated theorem proving, it is becoming more popular to verify aliasing and immutability properties using a type system. For example, three languages: L42 [6, 7, 8, 9], Pony [10, 11], and the language of Gordon *et al.* [12] use *reference capabilities*[5] and *object capabilities* to statically ensure deterministic parallelism and the absence of data races. While studying those languages, we discovered an elegant way to enforce invariants: we use capabilities to restrict how/when the result of invariant methods changes; this is done by restricting I/O, and how mutation through aliases can affect the state seen by invariants.

That is, our work shows that reference and object capabilities are useful also outside of the context of safe parallelism.

Reference Capabilities

Reference capabilities, as used in this paper, are a type system feature that allows reasoning about aliasing and mutation. A more recent design for them has emerged that radically improves their usability; three different research languages are being independently developed relying on this new design: the language of Gordon *et al.*, Pony, and L42. These projects are quite large: several million lines of code are written in Gordon *et al.*'s language and are used by a large private Microsoft project; Pony and L42 have large libraries and are active open source projects. In particular the reference capabilities of these languages are used to provide automatic and correct parallelism [12, 10, 11, 7].

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

- Mutable (`mut`): the referenced object can be mutated and shared/aliased without restriction; as in most imperative languages without reference capabilities.
- Immutable (`imm`): the referenced object cannot be mutated, not even through other aliases. An object with any `imm` aliases is an *immutable object*. Any other object is a *mutable object*. All objects are born mutable and may later become immutable. Thus, an object can be classified as *mutable* even if it has no fields that can be updated or mutated.
- Readonly (`read`): the referenced object cannot be mutated by such references, but there may also be mutable aliases to the same object, thus mutation can be observed. Readonly references can refer to both mutable and immutable objects, as `read` types are supertypes of both their `imm` and `mut` variants.
- Encapsulated (`capsule`): every mutable object in the reachable object graph of a capsule reference (including itself) is only reachable through that reference. Immutable objects in the reachable object graph of a capsule reference are not constrained, and can be freely referred to without passing through that reference.

[5] reference capabilities are called *Type Modifiers* in former works on L42.

There are only two kinds of objects: mutable and immutable, but there are more kinds of reference capabilities. In L42 only **mut** and **imm** references can be saved on the heap: **capsule** and **read** references only exists on the stack.

Reference capabilities are different to field or variable qualifiers like Java's **final**: reference capabilities 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.[6]

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

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
```

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

- A **mut** field accessed from a **read** reference produces a **read** reference; thus a **read** reference cannot be used to mutate the reachable object graph of the referenced object.
- Any field accessed from an **imm** reference produces an **imm** reference; thus all the objects in the reachable object graph of an immutable object are also immutable.

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

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

In L42, a **capsule** local variable always holds a **capsule** reference: this is ensured by allowing them to be read only once (similar to linear and affine types [18]). For example:

```
method mut Point foo(mut Point mp) {
  mp.x += 3; // mp is mut, so it can be used twice
  mp.y -= 3;
}

capsule Point cp = new Point(1, 2);
//cp.x = 3; cp.y = 3; // type error: cannot use 'cp' more than once
capsule Point cp2 = foo(cp); // ok, since foo(cp) only uses capsule variables
foo(cp2); // ok, 'cp2' is used only once
```

[6] 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**.

Pony and Gordon *et al.* follow a more complicated approach: **capsule** variables can be accessed multiple times, however in those cases the result will not be a **capsule** reference but another kind of reference, that can be promoted to **capsule**, but only under certain conditions. Pony and Gordon also provide destructive reads, where the variable's old value is returned as **capsule**.

Like **capsule** variables, how **capsule** fields are handled differs widely in the literature, however they must always be initialised and updated with **capsule** references. In order for access to a **capsule** field to safely produce a **capsule** reference, Gordon *et al.* only allows them to be read destructively (i.e. by replacing the field's old value with a new one, such as **null**). In contrast, Pony does not guarantee that **capsule** fields contain a **capsule** reference at all times, as it also provides non-destructive reads.

The formal model of L42 [19] does not contain **capsule** fields. The L42 concrete language interprets the syntax for capsule fields as private **mut** fields with some extra restrictions, including being initialised and updated only with **capsule** references. Those *encapsulated* fields (which do not support destructive reads) facilitate parallelism and can model various forms of ownership.[7] In Section 3 we present a novel kind of "**rep**" field. These, like **capsule** fields, can only be initialised/updated with **capsule** references, however alias to it can be created in restricted ways. Unlike **capsule** fields, which are usually designed for safe parallelism, these **rep** fields are specifically useful for invariant checking; we added support for them to L42, and believe they could be easily added to Pony and Gordon *et al.*'s language.

Promotion and Recovery

Many different techniques and type systems handle the reference capabilities above [16, 20, 21, 12, 6]. The main progress in the last few years is with the flexibility of such type systems: where the programmer should use **imm** when representing immutable data and **mut** nearly everywhere else. The system will be able to transparently promote/recover [12, 10, 6] the reference capability, adapting them to their use context. To see a glimpse of this flexibility, consider the following:

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

Here `mc`, `cc`, and `ic` are all syntactically initialised with the same exact expression. All **new** expressions return a **mut** [10, 19], so `mc` is well typed. The declarations of `cc` and `ic` are also well typed, since any expression (not just **new** expressions) of a **mut** type that has no **mut** or **read** free variables can be implicitly promoted to **capsule** or **imm**. This requires the absence of **read** and **mut** *global/static* variables, as in L42, Pony, and Gordon *et al.*'s language. L42 also allows such expression to use **read** free variables as well as **mut** variables as if they were **read**. For this to be sound, L42 does not allow **read** fields.

This is the main improvement on the flexibility of reference capabilities in recent literature [7, 6, 12, 10, 11]. From a usability perspective, this improvement means that programmers can write many classes simply using **mut** types and be free to have rampant aliasing. Then, at a later stage, another programmer may still be able to encapsulate instances of those data structures into an **imm** or **capsule** reference.

For example, imagine a program where most objects belong to classes designed without worrying about ownership, aliasing and encapsulation and with most methods requiring mutation. Thanks to the flexibility discussed above, those objects can still take advantage of our invariant protocol; we just need to apply our Box pattern around those.

Exceptions

In most languages exceptions may be thrown at any point. Combined with mutation this complicates reasoning about the state of programs after exceptions are caught: if an exception was thrown while mutating an object, what state is that object in? Does its invariant hold? The concept of *strong exception safety* [22, 8] simplifies reasoning: if a **try–catch** block caught an exception, the state visible before execution of the **try** block is unchanged, and the exception object does not expose any object that was being mutated; this prevents exposing objects whose invariant was left broken in the middle of mutations.

[7] It may seem surprising that those weaker forms of encapsulation are still sufficient to ensure safe unobservable parallelism. The detailed way L42 parallelism works is unrelated to the presented work. Please see the tutorial on `Forty2.is` (specifically, section 5 and 6) for more information on parallelism in L42.

L42 enforces strong exception safety for unchecked exceptions using reference capabilities[8] in the following way:[9]

- Only **imm** objects may be thrown as unchecked exceptions.
- Code inside a **try** block that captures unchecked exceptions is typed as if all variables declared outside of the block are **final** and all those of a **mut** type were **read**. With such restrictions those **try-catch**es can not rely on side effects to produce a result. In L42 **try-catch** is an expression, so the **try** can produce a result without the need of updating local variables. In a language where the **try-catch** is a statement, the **try** can still produce a result; for example using the **return** keyword.

This strategy does not restrict when exceptions can be *thrown*, but only restricts when unchecked exceptions can be *caught*. Strong exception safety allows us to throw invariant failures as unchecked exceptions: if an object's reachable object graph was mutated into a broken state within a **try**, when the invariant failure is caught, the mutated object will be unreachable/garbage-collectable. This works since strong exception safety guarantees that no object mutated within a **try** is visible when it catches an unchecked exception.[10]

For example:

```
// unchecked and checked exception types
class MyUnchecked extends RuntimeException { }
class MyChecked extends Exception { }

class Point {
  imm Int x; imm Int y;
  mut method Void add(imm Int d) throws MyChecked {
    this.x += d; this.y += d;
    if (...) { throw new MyUnchecked(); }//Always ok throwing unchecked exceptions
    if (...) { throw new MyChecked(); }//Ok: 'MyChecked' is in the 'throws'
  }}
...
try {
  mut Point p = new Point(1, 2)
  try {
    p.add(someNumber); // could throw a MyChecked, or any unchecked exception
    if (...) { p = new Point(...); } // update a local variable
  }
  catch (MyChecked e) { ... }// Ok
  // Adding the following 'catch' would be a type error,
  // as the body of the try needs 'p' to be non-final and mut:
  // catch (MyUnchecked e) { ... }
  ...
  try { if (p.x != p.y) { throw new MyUnchecked(); } }
  catch (MyUnchecked e) { // Ok:
    // the try part can be typed where 'p' is seen as final and read
    // we can be sure the value of 'p' has not changed
    // but we can change it here
    p = new Point(...) // or p.x+1;
  }
}
catch (MyUnchecked e) { ... } // ok, 'p' is guaranteed to be unreachable
```

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

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

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

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

Object Capabilities

Object capabilities, which L42, Pony, and Gordon *et al.*'s work have, are a widely used [23, 24, 25, 26] programming technique where access rights to resources are encoded as references to objects. When this style is respected, code unable to reach a reference to such an object cannot use its associated resource. Here, as in Gordon *et al.*'s work, we enforce the object capabilities pattern with reference capabilities in order to reason about determinism and I/O. To properly enforce this, the object capabilities style needs to be respected while implementing the primitives of the standard library, and when performing foreign function calls that could be non-deterministic, such as operations that read from files or generate random numbers. Such operations would not be provided by static methods, but instead by instance methods of classes whose instantiation is kept under control by carefully designing their implementation.

For example, in Java, `System.in` is a *capability object* that provides access to the standard input resource. However, since it is globally accessible it completely prevents reasoning about determinism. In contrast, if Java were to respect the object capability style, the `main` method could take a `System` parameter, as in

```
public static void main(System s){... s.in().read() ...}
```

Calling methods on that `System` instance would be the only way to perform I/O; moreover, the only `System` instance would be the one created by the runtime system before calling `main(s)`. This design has been explored by Joe-E [27].

Object capabilities are typically not part of the type system nor do they require runtime checks or special support beyond that provided by a memory safe language.

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

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

3. Our Invariant Protocol

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

Our protocol guarantees that the whole reachable object graph of any object involved in execution (formally, in a redex) is *valid*: if you can use an object, *manually* calling `invariant()` on it is guaranteed to return `true` in a finite number of steps.[11]

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

We restrict `invariant()` methods so that they represent a predicate over the receiver's `imm` and `rep` fields. To ensure that `invariant()` methods do not expose a potentially broken `this` to the other objects, we require that all occurrences of `this`[12]in the `invariant()`'s body are the receiver of a field access (`this.f`) of an `imm`/`rep`

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

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

field, or the receivers of a method call (`this.m(..)`) of a final (non-virtual) method that in turn satisfies these restrictions. No other uses of `this` are allowed, such as as the right hand side of a variable declaration, or an argument to a method. An equivalent alternative design could instead rely on static `invariant(..)` methods taking each `imm`/`rep` field as an `imm`/`read` parameter.

Invariants can only refer to immutable and encapsulated state. Thus while we can easily verify that a doubly linked list of immutable elements is correctly linked up, we can not do the same for a doubly linked lists of unencapsulated mutable elements. We do not make it harder to correctly implement lists of mutable elements, we only limit what invariants can be expressed in our protocol. In particular, as the nodes of the list must be mutable (since they reference the mutable elements), for these to be referenced in the `invariant` method, they must be reachable from a `rep` field, but then the elements of the list cannot be made accessible as `mut` from outside the list.[13] Note that we could use the transform pattern presented in Section 7 to mutate the elements of such a list, in a way that does not allow aliases to be saved outside.

There is a line of work [28] striving to allow invariants over other forms of state. We have not tried to integrate such solutions into our work, as we believe it would make our system more complex and ad hoc, probably requiring numerous specialised kinds of reference capabilities. Thus we have traded some expressive power in order to preserve safety and simplicity.

Purity

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

Rep Fields

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

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

Here we present a novel kind of encapsulated field, that we call a `rep` field. As for the various kinds of L42 `capsule` fields, our new kind of field is also just a private `mut` fields with extra restrictions, enforcing the following key property: the reachable object graph of a `rep` field $o.f$ can only be mutated under the control of a `mut` method of o, and during such mutation, o itself cannot be seen. This is similar to owner-as-modifier [29, 30], where we could consider an object to be the 'owner' of all the mutable objects in the reachable object graph of its `rep` fields, but with the extra restriction that the owner is unobservable during mutation of those objects.

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

[13] If our protocol were extended to support polymorphic reference capabilities (as in Gordon *et al.*'s work [12]), we could allow a reference with a polymorphic reference capability to be reachable from a `rep` field, provided that the state reachable from such a reference cannot be read from the `invariant` method. This could be done by supporting a reference capability that prevents reading state (such as the `tag` capability of Pony [10]), and requiring that such a polymorphic reference capability be type checked as if it were `tag`. Outside the list however, if the polymorphic reference capability is known to be `mut`, the elements could then be freely accessed and aliased, as the `invariant` would be guaranteed to not depend on them.

To show how our **rep** fields ensure these properties, we first define some terminology: `x.f` is a *field access*, `x.f=e` is a *field update*,[14] a **mut** method with a field access on a **rep** field of **this** is a *rep mutator*. [15] Note that a method performing a field *update* of a **rep** field (instead of a field access) is not called a rep mutator, but it is just a normal method performing a field update. Rep mutators handle the more subtle case where the fields of an object with invariant are not updated, but a mutation deep within their reachable object graph may potentially break the invariant.

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

- A **rep** field can only be initialised/updated using the result of an expression with **capsule** type.
- A **rep** field access will return a:
  - **mut** reference, when accessed on **this** within a rep mutator,
  - **read** reference, when accessed on any other **mut** receiver,
  - **imm** if the receiver is **imm**, **read** if the receiver is **read**, or **capsule** if the receiver is **capsule**. This last case is safe since a **capsule** receiver object will then be garbage collectable, so we do not need to preserve its invariant.
- A rep mutator must:
  - use **this** exactly once: to access the **rep** field,
  - have no **mut** or **read** parameters (except the **mut** receiver),
  - not have a **mut** return type,
  - not throw any checked exceptions[16].

The above rules ensure that rep mutators control the mutation of the reachable object graph of **rep** fields, and ensures our points (1), (2), and (3): o will not be in the reachable object graph of $o.f$ and only a rep mutator on o can see $o.f$ as **mut**; this means that the only way to mutate the reachable object graph of $o.f$ is through such methods. The restriction on the parameter types of a rep mutator ensures that o will not be reachable from any of the method's arguments, nor can these arguments be made reachable through $o.f$, which would violate our point (3). If execution is (indirectly) in a rep mutator, then o is only used as the receiver of the `this.f` expression in the rep mutator. Thus we can be sure that the reachable object graph of $o.f$ will only be mutated within a rep mutator, and only after the single use of o to access $o.f$. Since such mutation could invalidate the invariant of o, we call the `invariant()` method at the end of the rep mutator body; before o can be used again. Provided that the invariant is re-established before a rep mutator returns, no invariant failure will be thrown, even if the invariant was temporarily broken *during* the body of the method.[17]

The following example illustrates these properties of rep mutators:

```
class Foo {
  rep Point p;
  read method Bool invariant() {
    return this.p.x < this.p.y;
  }
```

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

[15]We could relax our protocol, so that a **mut** method that reads a **rep** field is not considered a rep mutator if the method only needs to use the field's value as **read**. This relaxation would merely be for convenience; it does not change expressivity as one can write a getter of form `read method read C m(){return this.f;}` for a `rep C f` field, and then call `this.m()` on a `mut this`.

[16]To allow rep 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.

[17]That is, rep mutators pretend that **this** is linear by requiring it to be used exactly once to read the **rep** field. By burying the only current access point to the **rep** field we can read it as **mut** and mutate it. The restrictions on parameter types and return types ensure that when such reference goes out of scope, the only remaining reference allowing mutation is in the **rep** field again. This is similar to the ideas of the Focus operation [3] and View-point adaptation [31].

```
  read method read Point getP() {
    return this.p; // ok, not a rep mutator
  }
  mut method read Point baz(capsule Qux q) { // a rep mutator
    mut Qux my_q = q; // Ok, q is used exactly once
    mut Point my_p = this.p; // ok, single use of 'this'
    my_p.x = my_q.compute_x(my_p); // may break the invariant
    // it is ok if the invariant does not hold, as 'this' cannot be reachable
    // from 'my_q' or 'my_p', this holds since 'q' is capsule, and so cannot
    // alias 'this'; if 'q' were instead mut or read, this would not be guaranteed.
    my_p.y = my_q.compute_y(my_p);
    return my_p; // invariant check here; type checks as return type is read
  }
}
```

In contrast, L42's pre-existing **capsule** fields do not have our rep mutator restrictions, in particular, other objects can mutate them, although storing references to them on the heap is highly restricted. These properties are also *weaker* than those of **capsule** *references*: we do not need to prevent arbitrary **read** aliases to the reachable object graph of a **rep** field, and we do allow arbitrary **mut** aliases to exist during the execution of a rep mutator. In particular, our rules allow unrestricted read only access to our **rep** fields.

Runtime Monitoring

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

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

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

Traditional Constructors and Subclassing

L42 constructors directly initialise all the fields using the parameters, and L42 does not provide traditional subclassing. L42 does however provide subtyping similar to Java 7's interfaces. This works naturally with our invariant protocol. We can support traditional constructors as in Pony and Gordon *et al.*'s language, by requiring that constructors only use **this** as the receiver of a field initialisation. Subclassing can be supported by forcing that a subclass invariant method implicitly starts with a check that **super**.invariant() returns **true**. We would also perform invariant checks at the end of **new** expressions, as happens in [32], and not at the end of **super**(..) constructor calls.

4. Essential Language Features

Our invariant protocol relies on many different features and requirements. In this section we will show examples of using our system, and how relaxing any of our requirements would break the soundness of our protocol. In our examples and in L42, the reference capability **imm** is the default, and so it can be omitted. Many verification approaches take advantage of the separation between primitive/value types and objects, since the former are immutable and do not support reference equality. However, our approach works in a pure OO setting without such a distinction. Hence we write all type names in **BoldTitleCase** to emphasise this. To save space, we omit the bodies of constructors that simply initialise fields with the values of the constructor's parameters, but we show their signature in order to show any annotations.

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

[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.

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

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

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

  Person(String name) {
    this.name = name; // check at end of constructor
    if (!this.invariant()) { throw new Error(...); }
  }
}
```

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

Unrestricted Access to Capability Objects?

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

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

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

Allowing Internal Mutation Through Back Doors?

Rust [33] and Javari [13] allow interior mutability: the reachable object graph of an 'immutable' object can be mutated through back doors. Such back doors would allow `invariant()` methods to store and read information about previous calls. The example class **MagicCounter** breaks determinism by remotely breaking the invariant of `person` without any interaction with the `person` object itself:

```
class MagicCounter {//INVALID EXAMPLE
  Int counter = 0;
  method Int incr(){return unsafe{counter++};} //using a backdoor
}
class NastyS extends String {..
  MagicCounter c = new MagicCounter(0);//can be 'imm' since it is 'unsafe'
  @Override read method Bool isEmpty(){return this.c.incr()!=2;}
}
...
NastyS name = new NastyS(); //the type system believes name's ROG is immutable
Person person = new Person(name); // person is valid, counter=1
name.incr(); // counter == 2, person is now broken
person.invariant(); // returns false, counter == 3
person.invariant(); // returns false, counter == 4
```

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

No Strong Exception Safety?

The ability to catch and recover from invariant failures allows programs to take corrective actions. Since we represent invariant failures by throwing unchecked exceptions, programs can recover from them with a conventional `try`–`catch`. Due to the guarantees of strong exception safety, any object that has been mutated during a `try` block is now unreachable, as happens in alias burying [18]. This property ensures that an object whose invariant fails will be unreachable after the invariant failure has been captured. If instead we were to not enforce strong exception safety, an invalid object could be made reachable. The following code is ill-typed since we try to mutate `bob` in a `try-catch` block that captures all unchecked exceptions; thus also including invariant failures:

```
mut Person bob = new Person("Bob");//INVALID EXAMPLE
// Catch and ignore invariant failure:
try { bob.name(""); } catch (Error t) { }// bob mutated
assert bob.invariant(); // fails!
```

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

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

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

Relaxing Restrictions on Rep Fields?

Rep fields allow expressing invariants over mutable object graphs. Consider managing the shipment of items, where there is a maximum combined weight:

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

We inject calls to `invariant()` at the end of the constructor and the `addItem(item)` method. This is safe since the `items` field is declared **rep**. Relaxing our system to allow a **mut** reference capability for the `items` field and the corresponding constructor parameter would make the above checks insufficient: it would be possible for external code with no knowledge of the **ShippingList** to mutate its items. In order to write correct library code in mainstream languages like Java and C++, defensive cloning [34, 35] is needed. For performance reasons, this is hardly done in practice and is a continuous source of bugs and unexpected behaviour.

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

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

```
//INVALID EXAMPLE: l.items can be exposed as mut
mut ShippingList l = new ShippingList(new Items()); // l is ok
mut Items evilAlias = l.items; // here l loses control
evilAlias.addItem(new HeavyItem()); // now l is invalid!
```

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

```
//INVALID EXAMPLE: rep mutator expose(c) return type is mut
mut method mut Items expose(C c) {return c.foo(this.items);}
```

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

Allowing **this** to be used more than once would allow the following code, where **this** may be reachable from `f`, thus `f.hi()` may observe an object that does not satisfying its invariant:

```
mut method Void multiThis(C c) {//INVALID EXAMPLE: two 'this'
  read Foo f = c.foo(this);
  this.items.add(new HeavyItem());
  f.hi(); }//'this' could be observed here if it is in ROG(f)
```

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

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

5. Formal Language Model

To model our system we need to formalise an imperative OO language with exceptions, non determinism (modelling I/O), object capabilities, and type system support for reference capabilities and strong exception safety. Formal models of the runtime semantics of such languages are simple, but defining and proving the correctness of such a type system is quite complex, and indeed many such papers exist that have already done this [7, 6, 12, 10, 8]. Thus we parameterise our language formalism, and assume we already have an expressive and sound type system enforcing the properties we need, so that we can separate our novel invariant protocol, from the non-novel reference capabilities. We clearly list in Appendix A the requirements we make on such a type system, so that any language satisfying them can soundly support our invariant protocol. In Appendix B we show an example type system, a restricted subset of L42, and prove that it satisfies our requirements. Conceptually our approach can parametrically be applied to any type system supporting these requirements, for example you could extend our type system with additional promotions or generics. To keep our small step reduction semantics as conventional as possible, we base our formalism on Featherweight Java [36] [37, Chapter 19], which is a Turing-complete [38] minimalistic subset of Java. As such, we model an OO language where receivers are always specified explicitly, and the receivers of field accesses and updates in method bodies are always `this`; that is, all fields are instance-private. Constructor declarations are not present explicitly, instead we assume they are all of the form C`(`$T_1\,x_1, \ldots, T_n\,x_n$`){this.`$f_1$`=`$x_1$`;`$\ldots$`;this.`$f_n$`=`$x_n$`}`, for appropriate types T_1, ..., T_n. Note that we do not model variable updates or traditional subclassing, since this would make the proofs more involved without adding any additional insight.

Notational Conventions

We use the following notational conventions:

- Class, method, parameter, and field names are denoted by C, m, x, and f, respectively.
- We use "vs" and "ls" as metavariables denoting a sequence of form $v_1, \ldots, v_n$ and $l_1, \ldots, l_n$, similarly with other metavariables ending in "s".
- We use "_" to stand for any single piece of syntax.
- Memory locations are denoted by l.
- We assume an implicit program/class table; we use the notation $C.m$ to get the method declaration for m within class C, similarly we use $C.f$ to get the declaration of field f, and $C.i$ to get the declaration of the i^{th} field.
- Memory, denoted by $\sigma : l \rightarrow C\{ls\}$, is a finite map from locations, l, to annotated tuples, $C\{ls\}$, representing objects; here C is the class name and ls are the field values. We use the notation C^{σ}_{l} to get the class name of l and $\sigma[l.f = l']$ to update a field of l, $\sigma[l.f]$ to access one. The notation σ, σ' combines the two memories, and requires that $dom(\sigma)$ is disjoint from $dom(\sigma')$.
- We assume a typing judgement of form $\sigma; \Gamma \vdash e : T$, this says that the expression e has type T, where the classes of any locations are stored in σ and the types of variables are stored in the environment $\Gamma : x \rightarrow T$.
- We allow the type system to impose any additional constraints it needs on method bodies. Our example type system in Appendix B for example requires that the method bodies are well-typed and only use `capsule` local variables once. However, our proofs in Appendix A do not assume any such restrictions.

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

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

Other than the `invariant` method of `True`, we impose no requirements on the methods of the `Bool` interface or its classes, in particular, they could be used to provide logical operations.[19]

[19] In particular, `if` statements can be supported using Church encoding: we would have a `Bool.if` method of form `read method` T `if(`T `ifTrue,` T `ifFalse)`, for an appropriate type T. The body of `True.if` will then be `ifTrue`, and the body of `False.if` will be `ifFalse`. In this way, x`.if(`t`,`f`)` will return t if x is "true" and b if it is "false". To ensure that t and f themselves are evaluated if and only if x is "true", the `Bool.if` method could instead be passed objects with `apply` methods, whose bodies will be t and f, respectively. If we added syntax sugar for lambdas, as in Java 8, we could then do x`.if(() ->` t`, () ->` f`).apply()`

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

Figure 1: Grammar

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

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

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

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

Grammar

The grammar is defined in Figure 1.

We use μ for our reference capabilities, and κ for field kinds. We don't model the preexisting L42 `capsule` fields, but instead model our novel `rep` fields, which can only be initialised/updated with `capsule` values. If `capsule` fields where added, they would not make our invariant protocol more interesting, as long as they do not provide a backdoor to create improper `capsule` references.

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

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

The three other expressions are:

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

- Monitor expressions ($\texttt{M(}l\texttt{;}\ e\texttt{;}\ e'\texttt{)}$) represent our runtime injected invariant checks. The location l refers to the object whose invariant is being checked, e represents the behaviour of the expression, and e' is the invariant check, which will initially be $\texttt{(read}\ l\texttt{).invariant()}$. The body of the monitor, e, is evaluated first, then the invariant check in e' is evaluated. If e' evaluates to an `imm True` (i.e. an `imm` reference to an instance of `True`), then the whole monitor expression will return the value of e, otherwise if it evaluates to a reference to a non-`True` value (i.e. an `imm` reference to an instance of a class other than `True`), the monitor expression is an *error*, and evaluation will proceed with the nearest enclosing `catch` block, if any. For example, assuming $\texttt{(read}\ l\texttt{).invariant()}$ terminates, we will have $\sigma|\texttt{M(}l\texttt{;}\ \texttt{new Foo();}\ \texttt{(read}\ l\texttt{).invariant())} \to \sigma, l' \mapsto \texttt{Foo}\{\}|\texttt{M(}l\texttt{;}\ l'\texttt{;}\ \texttt{(read}\ l\texttt{).invariant())} \to^* \sigma'|\texttt{M(}l\texttt{;}\ l'\texttt{;}\ \mu\, l''\texttt{)}$, i.e. we first reduce `new Foo()` to a value, then we reduce $\texttt{(read}\ l\texttt{).invariant()}$. If $\mathrm{C}^{\sigma}_{l''} = \texttt{True}$, then the invariant check succeeded and so the monitor will reduce to the result of `new Foo()`, i.e. $\sigma|\texttt{M(}l\texttt{;}\ \texttt{new Foo();}\ \texttt{(read}\ l\texttt{).invariant())} \to^* \sigma'|l'$; otherwise, the monitor expression $\texttt{M(}l\texttt{;}\ l'\texttt{;}\ \mu\, l''\texttt{)}$ will be stuck (it is an *error*), and the reduction will proceed to the `catch` block of the nearest enclosing `try`–`catch` (if any).

- `try`–`catch` expressions ($\texttt{try \{}e\texttt{\} catch \{}e'\texttt{\}}$), which as in many other expression based languages[20], evaluate e, and if successful, return its result, otherwise if e is an *error*, evaluation will reduce to e'. During reduction, `try`–`catch` expressions will be annotated as $\texttt{try}^{\sigma}\texttt{\{}e\texttt{\} catch \{}e'\texttt{\}}$, where σ is the state of the memory before the body of the `try` block begins execution. This annotation has no effect on the runtime, but is used by the proofs to model strong exception safety: objects in σ are not mutated by the body of the `try`. Note that as mentioned before, this strong limitation is only needed for unchecked exceptions, in particular, invariant failures. Our calculus only models unchecked exceptions/errors, however L42 also supports checked exceptions, and `try-catch`es over them impose no limits on object mutation during the `try`. This is safe since checked exceptions can not leak out of invariant methods or ref mutators: in both cases our protocol requires their `throws` clause to be empty. For example, we could have $\sigma|\texttt{try \{}e\texttt{\} catch \{}e'\texttt{\}} \to \sigma|\texttt{try}^{\sigma}\texttt{\{}e\texttt{\} catch \{}e'\texttt{\}} \to^* \sigma, \sigma'|\texttt{try}^{\sigma}\texttt{\{}\mathit{error}\texttt{\} catch \{}e'\texttt{\}} \to \sigma, \sigma'|e' \to^* \sigma'', \sigma'|v$. Thus the body of the `try` (e) has not modified σ, but it may have created new objects, which will be in σ'; the `catch` block on the other hand (e') can freely mutate σ into σ''. Note that the objects that e created (i.e. those in σ'), will not be reachable in e' (since σ has not been modified), i.e. an implementation could garbage collect them upon entering the `catch` block.

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

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

We provide several expression contexts, $\mathcal{E}$, $\mathcal{E}_v$, and $\mathcal{E}_r$. The standard evaluation context [37, Chapter 19], $\mathcal{E}_v$, represents the left-to-right evaluation order, an $\mathcal{E}_v$ is like an e, but with a *hole* ($\square$) in place of a sub-expression, but all the expression to the left of the hole must already be fully evaluated. This is used to model the standard left to right evaluation order: the hole denotes the location of the next sub-expression that will be evaluated. We use the notation $\mathcal{E}_v[e]$ to fill in the hole, i.e. $\mathcal{E}_v[e]$ returns $\mathcal{E}_v$ but with the single occurrence of $\square$ replaced by e. For example, if $\mathcal{E}_v = \square.m\texttt{()}$ then $\mathcal{E}_v[\texttt{new } C\texttt{()}] = \texttt{new } C\texttt{()}.m\texttt{()}$.

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

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

$error(\sigma, e)$ iff:

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

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

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

Reference Capability Operations

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

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

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

Well-Formedness Criteria

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

- `invariant()` methods must follow the requirements of Section 3, except that for simplicity method calls on `this` are not allowed.[21] This means that for every non-interface class C, $C.\texttt{invariant} = \texttt{read method imm Bool invariant()}\ e$, where e can only use `this` as the receiver of an `imm` or `rep` field access. Formally, this means that for all $\mathcal{E}$ where $e = \mathcal{E}[\texttt{this}]$, we have:
  - $\mathcal{E} = \mathcal{E}'[\square.f]$, for some $\mathcal{E}'$
  - $C.f = \kappa\ \_\ f$
  - $\kappa \in \{\texttt{imm}, \texttt{rep}\}$

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

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

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

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

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

(CALL) $\sigma|\mathcal{E}_v[\_\ l_0.m\texttt{(}\_\ l_1, ..., \_\ l_n\texttt{)}] \to \sigma|\mathcal{E}_v[e'\ \texttt{as}\ \mu']$, where:
$\mathrm{C}^\sigma_{l_0}.m = \mu_0\ \texttt{method}\ \mu'\ \_\ m\texttt{(}\mu_1\ \_\ x_1, ..., \mu_n\ \_\ x_n\texttt{)}\ e$
$e' = e[\texttt{this} \coloneqq \mu_0\, l_0, x_1 \coloneqq \mu_1\, l_1, ..., x_n \coloneqq \mu_n\, l_n]$
if $\mu_0 = \texttt{mut}$ then there are no f and $\mathcal{E}$ with $\mathrm{C}^\sigma_{l_0}.f = \texttt{rep}\ \_\ f$ and $e = \mathcal{E}[\texttt{this.}f]$

(CALL MUTATOR) $\sigma|\mathcal{E}_v[\_\ l_0.m\texttt{(}\_\ l_1, ..., \_\ l_n\texttt{)}] \to \sigma|\mathcal{E}_v[\texttt{M(}l_0\texttt{;}\ e\ \texttt{as}\ \mu'\texttt{;}\ \texttt{(read}\ l_0\texttt{).invariant())}]$, where:
$\mathrm{C}^\sigma_{l_0}.m = \texttt{mut method}\ \mu'\ \_\ m\texttt{(}\mu_1\ \_\ x_1, ..., \mu_n\ \_\ x_n\texttt{)}\ \mathcal{E}[\texttt{this.}f]$
$\mathrm{C}^\sigma_{l_0}.f = \texttt{rep}\ \_\ f$
$e = \mathcal{E}[\texttt{this.}f][\texttt{this} \coloneqq \texttt{mut}\ l_0, x_1 \coloneqq \mu_1\, l_1, ..., x_n \coloneqq \mu_n\, l_n]$

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

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

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

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

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

Figure 2: Reduction rules

- Rep mutators must also follow the requirements in Section 3, such methods must not use `this`, except for the single access to the `rep` field, and they must not have `mut` or `read` parameters, or a `mut` return type. Formally, this means that for any C, m, and f, if $C.f$ = `rep` _ f and $C.m$ = `mut method` μ' _ m`(`μ_1 _ _, ..., μ_n _ _`)` $\mathcal{E}$[`this.`f]:
  - `this` $\notin \mathcal{E}$
  - $\mu_1 \notin$ {`mut`, `read`}, ..., $\mu_n \notin$ {`mut`, `read`}
  - $\mu' \neq$ `mut`
- We require that the method bodies do not contain runtime expressions. Formally, for all C_0 and m with $C_0.m$ = _ `method` _ m`(`_ _, ..., _ _`)` e, e contains no l, `M(_; _; _)`, or `try`$^{\sigma'}$`{_} catch {_}` expressions.
- We also assume some general sanity requirements: every C mentioned in the program or in any well typed expression has a single corresponding `class`/`interface` definition; the Cs in an `implements` are all names of `interface`s; the C in a `new` C`(`es`)` expression denotes a `class`; the `implements` relationship is acyclic; the fields of a `class` have unique names; methods within a `class`/`interface` (other than `mut` methods in `Cap`) have unique names; and parameters of a method have unique names and are not named `this`.
- For simplicity of the type-system and associated proof, we require that every method in the (indirect) super-interfaces of a class be implemented with exactly the same signature, i.e. if we have a `class` C `implements` _ `{_;`Ms`}`, and `interface` C' `implements` _ `{`Ss`}`, where C' is reachable through the `implements` clauses starting from C, then for all $S \in Ss$, there is some e with $S\,e \in Ms$.

Reduction Rules

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

- NEW/NEW TRUE creates a new object. NEW is used when creating a non-`True` object, it returns a monitor expression that will check the new object's invariant, and if that succeeds, return a `mut` reference to the object. NEW TRUE is for creating an instance of `True`, it simply returns a `mut` reference to the new object, *without* checking its invariant. The separate NEW TRUE rule is needed as the invariant of `True` is itself defined to perform `new True()`, so using the NEW rule would cause an infinite recursion. This is sound since *manually* calling invariant on `True` will return a `True` reference. Note that although we do not define what *fresh* actually returns, since it is a *function* these reduction rules are deterministic: l_0 is uniquely defined for any given σ.
- ACCESS looks up the value of a field in the memory and returns it, annotated with the appropriate reference capability (see above for the definition of μ::κ).
- UPDATE updates the value of a field, returning a monitor that re-checks the invariant of the receiver, and if successful, will return the receiver of the update as `mut`. Note that this does *not* check that the receiver of the field update has an appropriate reference capability, it is the responsibility of the type-system to ensure that this rule is only applied to a `mut` or `capsule` receiver. For soundness, we return a `mut` reference even when the receiver is `capsule`. Promotion can then be used to convert the result to a `capsule`, provided the new field value is appropriately encapsulated.
- CALL/CALL MUTATOR looks for a corresponding method definition in the receiver's class, and reduces to its body with parameters appropriately substituted. The parameters are substituted with the reference capabilities of the method's signature, not the capabilities at the call-site, this is used by the proofs to show that further reductions will respect the capabilities in the method signature. We wrap the body of the method call in an `as` expression to ensure that the returned μ is actually as the method signature specified; for example, a method declared as returning a `read` might actually return a `mut`, but the `as` expressions will soundly change it to a `read`, thus preventing it from being used for

mutation. As with **as** expressions in general, the type system is required to ensure that this will not break our reference capability guarantees in Appendix A. The CALL MUTATOR rule is like CALL, but is used when the method is a rep mutator (a **mut** method that accesses a **rep** field): it additionally wraps the method body in a monitor expression that will re-check the invariant of the receiver once the body of the method has finished reducing. Note that as `Cap` has no **rep** fields and can have multiple definitions of the same method, the CALL rule allows for non-determinism, but only if the receiver is of class `Cap` and the method is a **mut** method.

- AS simply changes the reference capability to the one indicated. Note that our requirements on the type-system, given in Appendix A, ensure that inappropriate promotions (e.g. **imm** to **mut**) will be ill-typed.
- TRY ENTER will annotate a **try**–**catch** with the current memory state, before any reduction occurs within the **try** part. In Appendix A, we require the type system to ensure strong exception safety: that the objects in the saved σ are never modified. Note that the grammar for $\mathcal{E}_v$ prevents the body of an *unannotated* **try** block from being reduced, thus ensuring that this rule is applied first.
- TRY OK simply returns the body of a **try** block once it has successfully reduced to a value. TRY ERROR on the other hand reduces to the body of the **catch** block if its **try** block is an *error* (an invariant failure that is *not* enclosed by an inner **try** block). Note that the grammar for $\mathcal{E}_v$ prevents the body of a **catch** block from being reduced, instead TRY ERROR must be applied first; this ensures that the body of a **catch** is only reduced if the **try** part has reduced to an *error*.
- MONITOR EXIT reduces a successful invariant check to the body of the monitor. If the invariant check on the other hand has failed, i.e. has returned a non-`True` reference, it will be an *error*, and TRY ERROR will proceed to the nearest enclosing **catch** block.

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

Statement of Soundness

We define a deterministic reduction arrow to mean that exactly one reduction is possible:
$\sigma|e \Rightarrow \sigma'|e'$ iff $\sigma|e \rightarrow \sigma'|e'$, and $\forall \sigma'', e''$, $\sigma|e \rightarrow \sigma''|e''$, implies $\sigma''|e'' = \sigma'|e'$

We say that an object is *valid* when calling its `invariant()` method would deterministically produce an **imm** `True` in a finite number of steps, i.e. assuming the type system is sound, this means it does not evaluate to a non-`True` reference, fail to terminate, or produce an *error*. We also require that evaluating `invariant()` preserves existing memory, however new objects can be freely created and mutated:
$\mathit{valid}(\sigma, l)$ iff $\sigma|(\texttt{read}\, l)\texttt{.invariant()} \Rightarrow^+ \sigma, \sigma'|\texttt{imm}\, l$ where $\mathrm{C}_l^{\sigma,\sigma'} = \texttt{True}$.
To allow the `invariant()` method to be called on an invalid object, and access fields on such an objects, we define the set of trusted execution steps as the call to `invariant()` itself, and any field accesses inside its evaluation:
$\mathit{trusted}(\mathcal{E}_r, l)$ iff, either:
- $\mathcal{E}_r = \mathcal{E}_v[\texttt{M}(l;\ \_\,;\ \square\texttt{.invariant()})]$, or
- $\mathcal{E}_r = \mathcal{E}_v[\texttt{M}(l;\ \_\,;\ \mathcal{E}'_v[\square\texttt{.}f])]$.

The idea being that the $\mathcal{E}_r$ is like an $\mathcal{E}_v$ but it has a hole where a reference can be, thus $\mathit{trusted}(\mathcal{E}_r, l)$ holds when the very next reduction we are about to perform is $\mu\, l\texttt{.invariant()}$ or $\mu\, l.f$. As we discuss in our proof of Soundness, any such $\mu\, l.f$ expression came from the body of the `invariant()` method itself, since l can not occur in the *ROG* of any of its fields mentioned in the `invariant()` method.[22]

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

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

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

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

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

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

Theorem 1 (Soundness)**.**

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

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

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

6. Case Studies

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

6.1. An interactive GUI

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

Our case study involves a GUI with containers (`SafeMovable`s) and `Button`s. The `SafeMovable` class has an invariant to ensure that its children are graphically contained within it and do not overlap. The `Button`s move their `SafeMovable` when pressed. We have a `Widget` interface, which provides methods to get `Widget`s' size and position as well as children (a list of `Widget`s). Both `SafeMovable`s and `Button`s implement `Widget`. Crucially, since the children of `SafeMovable` are stored in a list of `Widget`s it can contain other `SafeMovable`s, and all queries to their size and position are dynamically dispatched. Such queries are also used in `SafeMovable`'s invariant. Here we show a simplified version[23], where `SafeMovable` has just one `Button` and certain sizes and positions are fixed. Note that `Widgets` is a class representing a mutable list of `mut Widget`s.

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

```
class SafeMovable implements Widget {
  rep Box box; Int width = 300; Int height = 300;

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

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

Our example shows how to encode interactive GUI programs, where widgets may circularly reference other widgets. In order to perform this case study we had to first implement a simple GUI Library in L42. This library uses object capabilities to draw the widgets on screen, as well as fetch and dispatch events. Importantly, neither our application, nor the underlying GUI library requires back doors, into either reference or object capabilities.

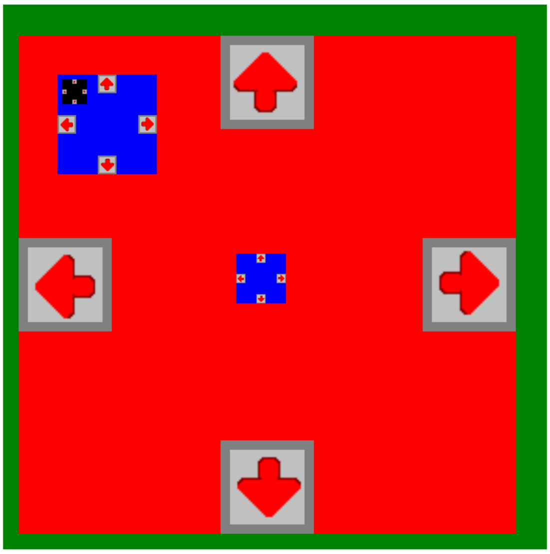
The Invariant

`SafeMovable` is the only class in our GUI that has an invariant, our system automatically checks it in two places: the end of its constructor and the end of its `dispatch(e)` method (which is a rep mutator). There are no other checks inserted since we never do a direct field update on a `SafeMovable`. The code for the invariant is just a couple of simple nested loops:[24]

```
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;
}
```

[24]We could make the code slightly more efficient by not comparing each pair of widgets twice. However, code efficiency is not the priority here.

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



Our Experiment

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

Comparison With Visible State Semantics

As an experiment, we set our implementation to generate invariant checks following the visible state semantics approaches of D and Eiffel [39, 40], where the invariant of the receiver is instead checked at the start and end of *every* public (in D) and qualified[25] (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' [40], 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;[26] in comparison our invariant protocol only performed 77 calls. The number of checks is exponential in the depth of the GUI: the invariant of a **SafeMovable** will call the width(), height(), left(), and top() methods of its children, which may themselves be **SafeMovable**s, and hence such calls may invoke further invariant checks. Note that width() and height() are simply getters for fields, whereas the other two are non-trivial *methods*. Concluding, we have shown that when an invariant check queries other objects with invariants the visible state semantics may cause an exponential explosion in the number of checks.

Spec# Comparison

We also encoded our example in Spec#[27]; that relies on pack/unpack; also called inhale/exhale or the Boogie methodology. In pack/unpack, an object's invariant is checked only by the explicit pack operations. In order for this to be sound, some form of aliasing and/or mutation control is necessary. Spec# uses a theorem prover, together with source code annotations. Spec# can be used for full static verification, but it conveniently allows invariant checks to be performed at runtime, whilst statically verifying aliasing, purity and other similar standard properties. This allows us to closely compare our approach with Spec#.

As the back-end of the L42 GUI library is written in Java, we did not port it to Spec#, rather we just simulated it, and don't actually display a GUI in Spec#. We ran our code through the Spec# verifier (powered by Boogie [41]), which only gave us 2 warnings[28], because the invariant of **SafeMovable** was not

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

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

[27] We compiled Spec# using the latest available source (from 19/9/2014).

[28] We used assume statements, equivalent to Java's assert, to dynamically check array bounds. This aligns the code with

known to hold at the end of its constructor and `dispatch(e)` method. Thus, like our system, Spec# checks the invariant at those two points at runtime. Thus the code is equivalently verified in both Spec# and L42; in particular it performed exactly the same number (77) of runtime invariant checks.

While the same numbers of checks are performed, we do not have the same guarantee provided by our approach: Spec#/Boogie does not soundly handle the non-deterministic impact of I/O, thus it does not properly prevent us from writing unsound invariants that may be non-deterministic. We also encoded our GUI in Microsoft Code Contracts [42], whose unsound heuristic also calls the invariant 77 times. However Code Contract does not enforce the encapsulation of `children()`, thus this approach is even less sound than Spec#.

Note how both our L42 and Spec# code required us to use the box pattern for our **`SafeMovable`**, due to the cyclic object graph caused by the **`Action`**s of **`Button`**s needing to change their enclosing **`SafeMovable`**'s position. We found it quite difficult to encode the GUI in Spec#, due to its unintuitive and rigid ownership discipline. In particular we needed to use many more annotations, which were larger and had greater variety. The following table shows the annotation burden, for the *program* that defines and displays the **`SafeMovable`**s and our GUI; as well as the *library* which defines **`Button`**s, **`Widget`**, and event handling. We only count constructs Spec# adds over C# as annotations, we also do not count annotations related to array bounds or null checks:

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

To encode the GUI example in L42, the only annotations we needed were the 3 reference capabilities: **`mut`**, **`read`**, and **`capsule`** (**`rep`** fields in the actual L42 language use the **`capsule`** keywords to minimise language complexity); Our Spec# code requires purity, immutability, ownership, method pre/post-conditions and method modification annotations. In addition, it requires the use of 4 different ownership functions including explicit ownership assignments. In total we used 18 different kinds of annotations in Spec#. The table presents token and character counts to compare against Spec#'s annotations, which can be quite long and involved, whereas ours are just single keywords. Consider for example the Spec# pre-condition on **`SafeMovable`**'s constructor:

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

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

6.2. A Comparison of a Simple Example in Spec#

Suppose we have a **`Cage`** class which contains a **`Hamster`**; the **`Cage`** will move its **`Hamster`** along a path. We would like to ensure that the **`Hamster`** does not deviate from the path. We can express this as the invariant of **`Cage`**: the position of the **`Cage`**'s **`Hamster`** must be within the path (stored as a field of **`Cage`**). This example is interesting since it relies on **`List`**s and **`Point`**s that are not designed with **`Hamster/Cage`**s in mind.

```
class Point {
  Double x; Double y; Point(Double x, Double y) {..}
  @Override read method Bool equals(read Object that) {
    if (!(that instanceof Point)) { return false; }
    Point p = (Point)that;
    return this.x == p.x && this.y == p.y; }}
```

L42, which also performs such checks at runtime.

```
class Hamster { Point pos; Hamster(Point pos) {..} }//pos is imm by default
class Cage {
  rep Hamster h;
  List<Point> path; //path is imm by default
  Cage(capsule Hamster h, List<Point> path) {..}
  read method Bool invariant() { return this.path.contains(this.h.pos); }
  mut method Void move() {
    Int index = 1 + this.path.indexOf(this.pos());
    this.moveTo(this.path.get(index % this.path.size())); }
  read method Point pos() { return this.h.pos; }
  mut method Void moveTo(Point p) { this.h.pos = p; }
}
```

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

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

Note: since only `Cage` has an invariant, only the code of `Cage` needs to be handled carefully; allowing the code for `Point` and `Hamster` to be unremarkable. Thus our verification approach is more self contained and modular. This contrasts with Spec#: all code involved in verification needs to be designed with verification in mind [43].

Comparison with Spec#

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

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

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

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

Though we don't want either `pos` or `path` to ever mutate, Spec# currently has no way of enforcing that an *instance* of a non-immutable class is itself immutable.[29] In Spec#, an `invariant()` can only access fields on owned or immutable objects, thus necessitating our use of the **Peer** and **Rep** annotations on the `pos` and `path` fields.

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

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

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

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

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

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

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

In Spec#, we had to add 10 different annotations, of 8 different kinds, worth a total of 20 tokens. In comparison, our approach requires only 8 simple keywords of 3 different kinds, for a total of 8 tokens. However, we needed to write separate `pos()` and `moveTo(p)` methods.

6.3. A Worst Case for the Number of Invariant Checks

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

[29] There is a paper [44] 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 reference capability system we use, which allows an initially mutable object to be promoted to immutable.

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

```
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); }
    }
  }
  rep 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;
  }}
```

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

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

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

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

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

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

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

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

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

Spec# relies on the absence of arithmetic overflow, and performs runtime checks to ensure this[30], as such the verifier concludes that the field increment in `processDay(dayOfYear)` cannot break the invariant. Spec# is able to avoid some invariant checks in this case by relying on all arithmetic operations performing runtime overflow checks; whereas integer arithmetic in L42 has the common wrap around semantics.

The annotations we had to add in the Spec# version[31] were similar to our previous examples, however since the fields of **Person** all have immutable classes/types, we only needed to add the invariant itself. In order to implement the `addChild(child)` method we were forced to do a shallow clone of the new child (this also caused a couple of extra runtime invariant checks). Unlike L42 however, we did not need to create a box to hold the `parents` and `children` fields, instead we wrapped the body of the **Family**`.processDay(dayOfYear)` method in an **expose** (**this**) block. In total we needed 16 annotations, worth a total of 45 tokens, this is slightly worse than the code following our approach that we showed above, which has 14 annotations and 14 tokens.

6.4. Encoding Examples from Spec# Papers

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

- *Verification of Object-Oriented Programs with Invariants* [3]: this paper introduces their methodology. In their examples section (pages 41–47), they show how their methodology would work in a class hierarchy with **Reader** and **ArrayReader** classes. The former represents something that reads characters, whereas the latter is a concrete implementation that reads from an owned array. They extend this further with a **Lexer** that owns a **Reader**, which it uses to read characters and parse them into tokens. They also show an example of a **FileList** class that owns an array of file names, and a **DirFileList** class that extends it with a stronger invariant. All of these examples can be represented in L42[32]. The most interesting considerations are as follow:
  - Their **ArrayReader** class has a `relinquishReader()` method that 'unpacks' the **ArrayReader** and returns its owned array. The returned array can then be freely mutated and passed around by other code. However, afterwards the **ArrayReader** will be 'invalid', and so one can only call methods on it that do not require its invariant to hold. However, it may later be 'packed' again

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

(after its invariant is checked). In contrast, our approach requires the invariant of all usable objects to hold. We can still relinquish the array, but at the cost of making the **ArrayReader** forever unreachable. This can be done by declaring `relinquishReader()` as a **capsule method**, this works since our type modifier system guarantees that the receiver of such a method is not aliased, and hence cannot be used again. Note that Spec# itself cannot represent the `relinquishReader()` method at all, since it does not provide explicit pack and unpack operations, rather its **expose** statement performs both an unpack and a pack, thus we cannot unpack an **ArrayReader** without repacking it in the same method.

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

- *Object Invariants in Dynamic Contexts* [45]: 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[33] (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* [28]: this paper shows how one can verify invariants over interacting objects, where neither owns/contains the others. They have multiple examples which utilise the 'subject/observer' pattern, where a 'subject' has some state that an 'observer' wants to keep track of. In their **Subject**/**View** example, **View**s are created with references to **Subject**s, and copies of their state. When a **Subject**'s state is modified, it calls a method on its attached **View**s, notifying them of this update. The invariant is that a **View**'s copy of its **Subject**'s state is up to date. Their **Master**/**Clock** example is similar, a **Clock** contains a reference to a **Master**, and saves a copy of the **Master**'s time. The **Master** has a **Tick** method that increases its time, but unlike the **Subject**/**View** example, the **Clock** is not notified. The invariant is that the **Clock**'s time is never ahead of its **Master**'s. Our protocol is unable to verify these interactions, because the interacting objects are not immutable or encapsulated by each other.

7. Patterns

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

The SubInvariant Pattern

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

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

[33] 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.

```
interface Person{ mut method Bool accept(read Account a,read Transaction t); }
interface Transaction{ mut method List<Transfer> compute(); }
//Here List<T> represents a list of immutable Ts.
class Transfer{ Int money;
  method Void execute(mut AccountBox that){// Gain some money, or lose some money
    if(this.money>0){ that.income+=money; }
    else{ that.expenses -= money; }
  }
}
class AccountBox{
  UInt income=0; UInt expenses=0;
  read method Bool subInvariant(){ return this.income >= this.expenses; }
  //An 'AccountBox' is like a 'potentially invalid Account':
  //we may observe income >= expenses
}
class Account{
  rep AccountBox box; mut Person holder;
  read method Bool invariant(){ return this.box.subInvariant(); }
  // 'h' could be aliased elsewhere in the program
  Account(mut Person h){ this.holder=h; this.box=new AccountBox(); }
  mut method Void transfer(mut Transaction ts){
    if(this.holder.accept(this, ts)){ this.transferInner(ts.compute()); }
  }
  // rep mutator, like an 'expose(this)' statement
  private mut method Void transferInner(List<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 rep mutator (`transferInner`) allows this by behaving like a Spec# **`expose`** block: during its body (the **`for`** loop) we don't know or care if **`this`**`.invariant()` is **`true`**, but at the end it will be checked. For this to make sense, we make **`Transfer`**`.execute` take an **`AccountBox`** instead of an **`Account`**: it cannot assume that the invariant of **`Account`** holds, and it is allowed to modify the fields of `that` without needing to check it. Though rep 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 object capabilities system), and other externally accessible objects (such as a **`mut Transaction`**) cannot be mutated during such a batch operation.

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

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

- Have useful objects that are not entirely encapsulated: the **`Person`** `holder` is a **`mut`** field; this is fine since it is not mentioned in the `invariant()` method.
- Wrap normal methods over rep mutators: `transfer` is not a rep mutator, so it can use **`this`** multiple times and take a **`mut`** parameter.
- Perform multiple state updates with only a single invariant check: the loop in `transferInner(ts)` can perform multiple field updates of `income` and `expenses`, however the `invariant()` will only be checked at the end of the loop.

- Temporarily break an invariant: it is fine if during the **for** loop, `expenses > income`, provided that this is fixed before the end of the loop.

- Pass the state of an 'invalid' object around, in a safe manner: an **AccountBox** contains the state of **Account**, but not the invariant method. Note how programmers can use conventional private types to control how such 'invalid' versions of objects are exposed in the public API, for example by declaring **AccountBox** as a private nested class. In contrast, if **invalid** was a type system feature, then any user defined type would intrinsically expose the existence of both variants in the public API.

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

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

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

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

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

```
interface HasSubInvariant{ read method Bool subInvariant(); }

class SafeMovable implements Widget,HasSubInvariant {
  Int width = 300; Int height = 300;
  Int left; Int top;  // Here we do not use a box, thus all the state
  mut Widgets c;      // is in SafeMovable.
  mut Widget parent;//We add a parent field

  @Override read method Int left(){ return this.left; }
  @Override read method Int top(){ return this.top; }
  @Override read method Int width(){ return this.width; }
  @Override read method Int height(){ return this.height; }
  @Override read method read Widgets children(){ return this.c; }
  @Override mut method Void dispatch(Event e){
    for(mut Widget w :this.c){ w.dispatch(e); }
  }
  @Override read method Bool subInvariant(){ /*same of original GUI*/ }

  SafeMovable(mut Widget parent,mut Widgets c){
    this.c=c;          //SafeMovable no longer has an invariant,
    this.left=5;       //so we impose no restrictions on its constructor
```

```
    this.top=5;
    this.parent=parent;
    c.add(new Button(0,0,10,10,new MoveAction(this));
  }
}

class MoveAction implements Action{
  mut SafeMovable o;
  MoveAction(mut SafeMovable o){ this.o = o; }
  mut method Void process(Event e){
    this.o.left+=1;
    Widget p = this.o.parent;
    ... // mutate p to re-establish its subInvariant
  }
}
class WidgetWithInvariant implements Widget{
  rep Widget w;
  @Override read method Int left(){ return this.w.left; }
  @Override read method Int top(){ return this.w.top; }
  @Override read method Int width(){ return this.w.width; }
  @Override read method Int height(){ return this.w.height; }
  @Override read method read Widgets children(){ return this.w.c; }
  @Override mut method Void dispatch(Event e){ w.dispatch(e); }
  @Override read method Bool invariant(){ return wInvariant(w); }
  static method Bool wInvariant(read Widget w){
    for(read Widget wi:w.children()){ if(!wInvariant(wi)){ return false; } }
    //Check that the subInvariant of all of w's descendants holds
    if(!(w instanceof HasSubInvariant)){ return true; }
    HasSubInvariant si = (HasSubInvariant)w;
    return si.subInvariant();
  }
  WidgetWithInvariant(capsule Widget w){ this.w = w; }
}
... // main expression
//#$ is a capability operation making a Gui object
mut Widget top=new WidgetWithInvariant(new SafeMovable(...))
Gui.#$().display(top);
```

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

Importantly, this allows the graph of widgets to be cyclic and for each to freely mutate each other, even if such mutations (temporarily) violate their `subInvariant`'s. In this way a widget can access its parent (whose `subInvariant()` may not hold) in order to re-establish it. Note that this trade off is logically unavoidable: in order to manipulate a parent in order to fix it, the parent must be reachable, but by mutating a `Widget`'s position, its parent may become invalid. Thus if `Widget`s were to encode their validity in their `invariant()` methods they could not have access to their parents. Instead, by encoding their validity in a `subInvariant()` method, they can access invalid widgets, but this comes at a cost: the programmer must reason as to when `Widgets` are valid, as we described above.

The Transform Pattern

Recall the GUI case study from Section 6.1, where we had a `Widget` interface and a `SafeMovable` (with an invariant) that implements `Widget`. Suppose we want to allow `Widget`s to be scaled, we could add `mut` setters for `width()`, `height()`, `left()`, and `top()` in the `Widget` interface. However, if we also wish to scale its children we have a problem, since `Widget.children()` returns a `read Widgets`, which does not allow mutation. We could of course add a `mut` method `zoom(w)` to the `Widget` interface, however this does not scale if more operations are desired. If instead `Widget.children` returned a `mut Widgets`, it would be difficult for `Widget` implementations, such as **SafeMovable**, to mention their `children()` in their `invariant()`. A simple and practical solution would be to define a `transform(t)` method in **Widget**, and a **Transformer** interface like so:

```
interface Transformer<T> { capsule method Void apply(mut T elem); }

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

The `transform` method offers an expressive power similar to **mut** getters, but prevents **Widgets** from leaking out.[34] With a **Transformer**, a `zoom(w)` function could be simply written as:

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

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

Here, we assume lambdas, as in Java, are sugar for normal objects that implement an interface with a single abstract method. As an example, we could use the following sound rules to determine what lambdas are allowed: **imm** lambda objects implementing an interface with an **imm** method which only captures final **imm** variables, **mut** lambdas implementing a **mut** method which only captures final **imm** and **mut** variables, and **capsule** lambdas implementing a **capsule** method which only captures final **imm** and **capsule** variables.

Using Patterns Together: A General and Flexible Graph Class

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

The **Node**s are guaranteed to be encapsulated by the **Graph**, however they can be arbitrarily modified by user defined transformations using the transform pattern.

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

interface Node{
  read method Bool subInvariant(read Nodes nodes)
  mut method mut Nodes directConnections()
}
class Nodes{//just an ordered set of nodes
  mut method Void add(mut Node n){..}
  read method Int indexOf(read Node n){..}
  mut method Void remove(read Node n){..}
```

[34] Note how this kind of pattern solves a similar problem in ownership systems where an object cannot be modified except under the control of the owner. In our example, this would correspond to the **SafeMovable** being the ‘owner’ of it’s ‘box.c’ field.

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

We now show how our **Graph** library allows the invariant of the various **Node**s to be customised by the library user, and arbitrary transformations can be performed on the **Graph**s. This is a generalisation of the example proposed by [46](section 4.2) as one of the hardest problems when it comes to enforcing invariants.

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

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

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

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

These methods can be implemented like this:

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

```
  mut MyNode n1=this;//single use of capsule 'this'
  ns.add(n1);
  });
}
```

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

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

8. Integration in L42

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

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

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

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

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

In this example, the **Data** decorator generates a factory method, a **mut method Void** `addItem(`**Item** `item)` and a lot of other utility methods, including equality and conversion to string. In particular, the current concrete L42 syntax uses the **capsule** keyword to ensure various properties of a field. The language relies on the presence of annotations or other specific methods to decide what restrictions to apply and properties to ensure. In this case, the presence of the **@Cache.Now** annotation clarifies that the field **capsule Items** `items` is actually a **rep** field as discussed in our work. The **@Cache.Now** annotation causes the `invariant` method to be automatically computed, and recomputed every time a **@Cache.Clear** method is called.

The `X[...]` notation used in `invariant` is an assert statement: it throws an unchecked exception if it's argument is false. Please refer to `Forty2.is` for more information.

9. Related Work

Reference Capabilities

We rely on a combination of reference capabilities supported by at least three languages/lines of research: L42 [6, 7, 8, 9], Pony [10, 11, 47], and Gordon *et al.* [12]. They all support full/deep interpretation, without back doors. Former works [48, 49, 50, 51, 52] (which eventually enabled the work of Gordon *et al.*) do not consider promotion and infers uniqueness/isolation/immutability only when starting from references that have been tracked with restrictive annotations along their whole lifetime. Other approaches like Javari [13, 53] and Rust [33] provide back doors, which are not easily verifiable as being used properly.

Ownership [54, 16, 31] is a popular form of aliasing control often used as a building block for static verification [55, 43]. However, ownership does not require the whole reachable object graph of an object to be 'owned'. This complicates restricting the data accessible by invariants.

Object Capabilities

In the literature, object capabilities are used to provide a wide range of guarantees, and many variations are present. Object capabilities, in conjunction with reference capabilities, are able to enforce purity of code in a modular way, without requiring the use of effects or monads. L42 and Gordon *et al.* use object capabilities simply to reason about I/O and non-determinism. This approach is best exemplified by Joe-E [27], which is a self-contained and minimalistic language using object capabilities (but not reference capabilities) over a subset of Java in order to reason about determinism. However, in order for Joe-E to be a subset of Java, they leverage a simplified model of immutability: immutable classes must be final and have only final fields that refer to immutable classes. In Joe-E, every method that only takes instances of immutable classes is pure. Thus their model would not allow the verification of purity for invariant methods of mutable objects. In contrast our model has a more fine grained representation of mutability: it is *reference-based* instead of *class-based*. Thanks to this crucial difference, in our work every method taking only `read` or `imm` *references* as receivers and parameters is pure, regardless of their class type. In particular, we allow the parameter of such a method to be mutated later on by other code.

Invariant Protocols

Invariants are a fundamental part of the design by contract methodology. Invariant protocols differ wildly and can be unsound or complicated, particularly due to re-entrancy and aliasing [45, 56, 57].

While invariant protocols all check and assume the invariant of an object after its construction, they handle invariants differently across object lifetimes. Popular approaches include:

- The invariants of objects in a *steady* state are known to hold: that is when execution is not inside any of the objects' public methods [5]. Invariants need to be constantly maintained between calls to public methods.
- The invariant of the receiver before a public method call and at the end of every public method body needs to be ensured. The invariant of the receiver at the beginning of a public method body and after a public method call can be assumed [58, 56]. Some approaches ensure the invariant of the receiver of the *calling* method, rather than the *called* method [59]. JML [60] relaxes these requirements for helper methods, whose semantics are the same as if they were inlined.
- The same as above, but only for the bodies of 'selectively exported' (i.e. not instance-private) methods, and only for 'qualified' (i.e. not `this`) calls [57].
- The invariant of an object is assumed only when a contract requires the object be 'packed'. It is checked after an explicit 'pack' operation, and objects can later be 'unpacked' [3].
- Upon calling a method (a.k.a a function/subprogram), the invariant of each parameter (and part/field of a parameter) must be shown to hold; upon returning from a method, the invariant of each parameter

(and their parts) must still hold, and the invariant of the return value (and their parts) must hold [61, 62]. As a relaxation, one approach only requires such invariants to hold if the method is declared within the scope of the invariant's declaration, but visible outside of it [63]. To enable encapsulation of invariants, for any method call located within the scope of an invariant, but calling a method outside this scope, the invariants of each of the call's arguments (and their parts) must be shown to hold [64].

- The same as above, but an invariant may optionally be declared 'strong', requiring that it must hold for every variable/parameter (and their parts) at every well-defined step of execution (a 'sequence point') [61].

These different protocols can be deceivingly similar. Note that all, except the last, of those approaches fail our strict requirements and allow for broken objects to be observed. Some approaches like JML suggest verifying a simpler approach (that method calls preserve the invariant of the *receiver*) but assume a stronger one (the invariant of *every* object, except `this`, holds).

Security and Scalability

Our approach allows verifying an object's invariant independently of the execution context. This is in contrast to the main strategy of static verification [65, 1, 66]: to verify a method, the system assumes the contracts of other methods, and the content of those contracts is the starting point for their proof. Thus, static verification proceeds like a mathematical proof: a program is valid if it is all correct, but a single error invalidates all claims. This makes it hard to perform verification on large programs, or when independently maintained third party libraries are involved. Static verification has more flexible and fine-grained annotations and often relies on a fragile theorem prover as a backend.

To soundly verify code embedded in an untrusted environment, as in gradual typing [67, 68], it is possible to consider a verified core and a runtime verified boundary. One can see our approach as an extremely modularised version of such a system: every class is its own verified core, and the rest of the code could have Byzantine behaviour. Our proofs show that every class that compiles/type checks is soundly handled by our protocol, independently of the behaviour of code that uses such a class or any other surrounding code.

Our approach works both in a library setting and with the open world assumption. Consider for example the work of Parkinson [69]: he verified a property of the `Subject/Observer` pattern. However, the proof relies on (any override of) the `Subject.register(Observer)` method respecting its contract. Such assumption is unrealistic in a real-world system with dynamic class loading, and could trivially be broken by a user-defined `EvilSubject`: checking contracts at load time is impractical and is not done by any verification systems we know of.

Static Verification

AutoProof [70] is a static verifier for Eiffel that also follows the Boogie methodology, but extends it with *semantic collaboration* where objects keep track of their invariants' dependencies using ghost state.

Dafny [1] is a language where all code is statically verified. It supports invariants with its `{:autocontracts}` annotation, which treats a class's `Valid()` function as the invariant and injects pre and post-conditions following visible state semantics. However it requires objects to be newly allocated (or cloned) before another object's invariant may depend on it. Dafny is also generally highly restrictive with its rules for mutation and object construction, it also does not provide any means of performing non-deterministic I/O.

Spec# [66] 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 [71] where (implicit) annotations are used to specify and modify the owner of objects and whether their invariants are required to hold. Invariants can be *ownership* based [3], where an invariant only depends on objects it owns; or *visibility* based [28, 72], where an invariant may depend on objects it doesn't own, provided that the class of such objects know about this dependence. Unlike our approach, Spec# does not restrict the aliases that may exist for an object, rather it restricts object mutation: an object cannot be modified if the invariant of its owner is required to hold. This allows invariants to query owned mutable objects whose reachable object graph is not fully encapsulated. However as we showed in Section 6.1, it can become much more difficult to work with and requires significant annotation, since merely having an alias to an object is insufficient to modify it or call its methods. Spec# also works with existing .NET libraries by annotating them with contracts, however such annotations are not verified. Spec#, like our approach, does perform runtime checks for invariants and throws unchecked exceptions on failure. However Spec# does not allow soundly recovering from an invariant failure, since catching unchecked exceptions in Spec# is intentionally unsound. [73]

Static verification of multi object invariants is a very difficult problem. Many of the modularity issues discussed in "Modular invariants for layered object structures" [59] do not apply to our environment: by checking the invariant at run time it is not a problem if we do not know the implementation we depends on, making us more flexible. Using their terminology, our work would be *encapsulation based* and not *visibility based*. However, our encapsulation strategies are much more flexible. Our box pattern can be used to emulate many *visibility based* invariants, simply by putting the invariant into a box containing all involved objects.

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 [74] discovered that developers expect short-circuit semantics and arithmetic exceptions in specification languages to follow the semantics of the underlying language; 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 [75]. Dafny [1] uses a hybrid approach: it has mostly the same language for both specification and execution. Specification ('ghost') contexts can use uncomputable constructs such as universal quantification over infinite sets, whereas runtime contexts allow mutation, object allocation and print statements. The semantics of shared constructs (such as short circuiting logic operators) is the same in both contexts. Most runtime verification systems, such as ours, use a metacircular approach: specifications are simply code in the underlying language. Since specifications are checked at runtime, they are unable to verify uncomputable contracts.

Ensuring determinism in a non-functional language is challenging. Spec# recognizes the need for purity/determinism when method calls are allowed in contracts [76] '*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 recognise that many tools unsoundly use option (c), such as AsmL [77]. Spec# aims to follow (b) but only considers non-determinism caused by memory mutation, and allows other non deterministic operations, such as I/O and random number generation. In Spec# the following verifies:
`[Pure] bool uncertain() {return new Random().Next() % 2 == 0;}`
And so `assert uncertain() == uncertain();` also verifies, but randomly fails with an exception at runtime. As you can see, failing to handle non-determinism jeopardises reasoning. A simpler and more restrictive solution to these problems is to restrict 'pure' functions so that they can only read final fields and call other pure functions. This is the approach used by [78]. One advantage of their approach is that invariants (which must be 'pure') can read from a chain of final fields, even when they are contained in otherwise mutable objects. However their approach completely prevents invariants from mutating newly allocated objects, thus greatly restricting how computations can be performed.

Runtime Verification Tools

By looking to surveys [79, 80] and the extensive MOP project [81], it seems that most runtime verification tools empower users to implement the kind of monitoring they see fit for their specific problem at hand. This means that users are responsible for deciding, designing, and encoding both the logical properties and the instrumentation criteria [81]. 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 [82] lets you write Python conditions that are verified on a set of Python objects, but the programmer needs to be able to predict which objects are in need of being checked and to use a simple domain specific language to target them. Hence if a programmer makes a mistake while using this domain specific language, invariant checking will not be triggered. Some tools are intentionally unsound and just perform invariant checking following some heuristic that is expected to catch most failures: such as jmlrac [58] and Microsoft Code Contracts [83].

Many works attempt to move out of the 'runtime verification tool' philosophy to ensure runtime verification monitors work as expected, as for example the study of contracts as refinements of types [84]. However, such work is only interested in pre and post-conditions, not invariants.

Our invariant protocol is much stricter than visible state semantics, and keeps the invariant under tight control. Gopinathan *et al.*'s. [5] approach keeps a similar level of control: relying on powerful aspect-oriented support, they detect any field update in the whole reachable object graph 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 soundness guarantee does not scale to programs using such features.

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

Chaperones and impersonators [85] lifts the techniques of gradual typing [86, 67, 68] to work on general purpose predicates, where values can be wrapped to ensure an invariant holds. This technique is very powerful and can be used to enforce pre and post-conditions by wrapping function arguments and return values. This technique however does not monitor the effects of aliasing, as such they may notice if a contract has been broken, but not when or why. In addition, due to the difficulty of performing static analysis in weakly typed languages, they need to inject runtime checking code around every user-facing operation.

10. Conclusion

In this paper we (1) identified language features that soundly support representation invariants in object-oriented verification; (2) presented a full formalism for our approach with capabilities that is proved to soundly guarantee that all objects involved in execution are valid; (3) conducted extensive case studies showing that we require orders of magnitude fewer runtime checks than *visible state semantics* and approximately 31% fewer annotations (with 3 1/2 times fewer tokens)[35] than equivalent versions in Spec#. We hope that as a result of this work, the software verification community will make more use of the advanced general purpose language features, such as capabilities, appearing in modern languages to achieve its goals.

Our approach follows the principles of *offensive programming* [87] where no attempt to fix or recover an invalid object is performed. Failures (unchecked exceptions) are raised close to their cause: at the end of constructors creating invalid objects and immediately after field updates and instance methods that invalidate their receivers.

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

[35]Calculated by combining the counts from our GUI, Hamster, and Family case studies.

References

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

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

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

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

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

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

[7] M. Servetto, D. J. Pearce, L. Groves, A. Potanin, Balloon types for safe parallelisation over arbitrary object graphs, in: WODET 2014 - Workshop on Determinism and Correctness in Parallel Programming, 2013. doi:doi=10.1.1.353.2449.

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

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

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

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

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

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

[14] A. Birka, M. D. Ernst, A practical type system and language for reference immutability, in: ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2004), 2004, pp. 35–49. doi:10.1145/1035292.1028980.

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

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

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

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

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

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

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

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

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

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

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

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

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

[28] M. Barnett, D. A. Naumann, Friends need a bit more: Maintaining invariants over shared state, in: Mathematics of Program Construction, 7th International Conference, MPC 2004, Stirling, Scotland, UK, July 12-14, 2004, Proceedings, 2004, pp. 54–84. doi:10.1007/978-3-540-27764-4_5.
[29] W. Dietl, P. Müller, Universes: Lightweight ownership for jml, JOURNAL OF OBJECT TECHNOLOGY 4 (8) (2005) 5–32.
[30] D. Cunningham, W. Dietl, S. Drossopoulou, A. Francalanza, P. Müller, A. J. Summers, Universe types for topology and encapsulation, in: Formal Methods for Components and Objects, 2008, pp. 72–112.
[31] W. Dietl, S. Drossopoulou, P. Müller, Generic universe types, in: ECOOP'07 - Object-Oriented Programming, Vol. 4609 of Lecture Notes in Computer Science, Springer, 2007, pp. 28–53. doi:10.1007/978-3-540-73589-2_3.
[32] Y. A. Feldman, O. Barzilay, S. Tyszberowicz, Jose: Aspects for design by contract, in: Software Engineering and Formal Methods, 2006. SEFM 2006. Fourth IEEE International Conference on, IEEE, 2006, pp. 80–89. doi:10.1109/SEFM.2006.26.
[33] N. D. Matsakis, F. S. Klock II, The rust language, in: ACM SIGAda Ada Letters, Vol. 34, ACM, 2014, pp. 103–104. doi:10.1145/2663171.2663188.
[34] J. Bloch, Effective Java (2Nd Edition) (The Java Series), 2nd Edition, Prentice Hall PTR, 2008.
[35] D. L. Detlefs, K. R. M. Leino, G. Nelson, Wrestling with rep exposure (1998).
[36] A. Igarashi, B. C. Pierce, P. Wadler, Featherweight Java: a minimal core calculus for Java and GJ, ACM Transactions on Programming Languages and Systems 23 (3) (2001) 396–450.
[37] B. C. Pierce, Types and programming languages, MIT press, 2002.
[38] R. N. S. Rowe, S. J. Van Bakel, Semantic types and approximation for featherweight java, Theor. Comput. Sci. 517 (2014) 34–74. doi:10.1016/j.tcs.2013.08.017.
URL https://doi.org/10.1016/j.tcs.2013.08.017
[39] A. Alexandrescu, The D Programming Language, 1st Edition, Addison-Wesley Professional, 2010.
[40] B. Meyer, Eiffel: The Language, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.
[41] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, K. R. M. Leino, Boogie: A modular reusable verifier for object-oriented programs, in: Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures, 2005, pp. 364–387. doi:10.1007/11804192_17.
[42] M. Fähndrich, M. Barnett, F. Logozzo, Embedded contract languages, in: Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010, 2010, pp. 2103–2110. doi:10.1145/1774088.1774531.
[43] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, H. Venter, Specification and verification: the spec# experience, Communications of the ACM 54 (6) (2011) 81–91. doi:10.1145/1953122.1953145.
[44] K. R. M. Leino, P. Müller, A. Wallenburg, Flexible immutability with frozen objects, in: Verified Software: Theories, Tools, Experiments, Second International Conference, VSTTE 2008, Toronto, Canada, October 6-9, 2008. Proceedings, 2008, pp. 192–208. doi:10.1007/978-3-540-87873-5_17.
[45] K. R. M. Leino, P. Müller, Object invariants in dynamic contexts, in: European Conference on Object-Oriented Programming, Springer, 2004, pp. 491–515. doi:10.1007/978-3-540-24851-4_22.
[46] A. J. Summers, S. Drossopoulou, P. Müller, The need for flexible object invariants, in: International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming, IWACO '09, ACM, New York, NY, USA, 2009, pp. 6:1–6:9. doi:10.1145/1562154.1562160.
URL http://doi.acm.org/10.1145/1562154.1562160
[47] S. Clebsch, Pony: co-designing a type system and a runtime, Ph.D. thesis (2017).
[48] J. Boyland, Semantics of fractional permissions with nesting, ACM Transactions on Programming Languages and Systems 32 (6) (2010). doi:10.1145/1749608.1749611.
[49] J. Boyland, Checking interference with fractional permissions, in: International Static Analysis Symposium, Springer, 2003, pp. 55–72.
[50] J. Hogg, Islands: Aliasing protection in object-oriented languages, in: ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1991, ACM Press, 1991, pp. 271–285.
[51] F. Smith, D. Walker, J. G. Morrisett, Alias types, in: Proceedings of the 9th European Symposium on Programming Languages and Systems, ESOP '00, Springer-Verlag, London, UK, UK, 2000, pp. 366–381.
URL http://dl.acm.org/citation.cfm?id=645394.651903
[52] A. Aiken, J. S. Foster, J. Kodumal, T. Terauchi, Checking and inferring local non-aliasing, in: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003, 2003, pp. 129–140. doi:10.1145/781131.781146.
[53] J. Boyland, Why we should not add readonly to Java (yet), Journal of Object Technology 5 (5) (2006) 5–29. doi:10.5381/jot.2006.5.5.a1.
[54] D. Clarke, J. Östlund, I. Sergey, T. Wrigstad, Ownership types: A survey, in: D. Clarke, J. Noble, T. Wrigstad (Eds.), Aliasing in Object-Oriented Programming. Types, Analysis and Verification, Vol. 7850 of Lecture Notes in Computer Science, Springer, 2013, pp. 15–58. doi:10.1007/978-3-642-36946-9_3.
[55] P. Müller, Modular specification and verification of object-oriented programs, Springer-Verlag, 2002. doi:10.1007/3-540-45651-1.
[56] S. Drossopoulou, A. Francalanza, P. Müller, A. J. Summers, A unified framework for verification techniques for object invariants, in: European Conference on Object-Oriented Programming, Springer, 2008, pp. 412–437. doi:10.1007/978-3-540-70592-5_18.
[57] B. Meyer, Class invariants: Concepts, problems, solutions, arXiv preprint arXiv:1608.07637 (2016).
[58] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, E. Poll, An overview of jml tools and applications, International Journal on Software Tools for Technology Transfer 7 (3) (2005) 212–232.

doi:10.1007/s10009-004-0167-4.
[59] P. Müller, A. Poetzsch-Heffter, G. T. Leavens, Modular invariants for layered object structures, Sci. Comput. Program. 62 (3) (2006) 253–286. doi:10.1016/j.scico.2006.03.001.
URL https://doi.org/10.1016/j.scico.2006.03.001
[60] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Muller, Joseph Kiniry, Patrice Chalin, Daniel M. Zimmerman, Werner Dietl, JML Reference Manual (2013).
URL http://www.eecs.ucf.edu/~leavens/JML//refman/jmlrefman.pdf
[61] P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, V. Prevosto, Acsl: Ansi/iso c specification language: Version 1.16 (2020).
URL https://www.frama-c.com/download/acsl.pdf
[62] F. Bobot, J.-C. Filliâtre, C. Marché, G. Melquiond, A. Paskevich, The why3 platform: Version 1.5 (Apr. 2022).
URL https://why3.lri.fr/doc/
[63] I. Intermetrics, I. The MITRE Corporation, A. Consultants, Ada-Europe, Ada reference manual: 2012 edition (2012).
URL http://www.ada-auth.org/standards/ada12.html
[64] AdaCore, A. U. Ltd, Spark 2014 reference manual: Version 2020 (Apr. 2020).
URL https://www.adacore.com/papers/spark-2014-reference-manual-release-2020
[65] C. A. R. Hoare, An axiomatic basis for computer programming, Commun. ACM 12 (10) (1969) 576–580. doi:10.1145/363235.363259.
URL https://doi.org/10.1145/363235.363259
[66] M. Barnett, K. R. M. Leino, W. Schulte, The spec# programming system: An overview, in: Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, CASSIS'04, Springer-Verlag, Berlin, Heidelberg, 2005, pp. 49–69. doi:10.1007/978-3-540-30569-9_3.
[67] A. Takikawa, T. S. Strickland, C. Dimoulas, S. Tobin-Hochstadt, M. Felleisen, Gradual typing for first-class classes, in: Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012, 2012, pp. 793–810. doi:10.1145/2384616.2384674.
[68] T. Wrigstad, F. Z. Nardelli, S. Lebresne, J. Östlund, J. Vitek, Integrating typed and untyped code in a scripting language, in: Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010, 2010, pp. 377–388. doi:10.1145/1706299.1706343.
[69] M. Parkinson, Class invariants: The end of the road?, Aliasing, Confinement and Ownership in Object-oriented Programming (IWACO) (2007) 9.
[70] N. Polikarpova, J. Tschannen, C. A. Furia, B. Meyer, Flexible invariants through semantic collaboration, in: FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings, 2014, pp. 514–530. doi:10.1007/978-3-319-06410-9_35.
[71] D. A. Naumann, M. Barnett, Towards imperative modules: Reasoning about invariants and sharing of mutable state, Theor. Comput. Sci. 365 (1-2) (2006) 143–168. doi:10.1016/j.tcs.2006.07.035.
URL https://doi.org/10.1016/j.tcs.2006.07.035
[72] K. R. M. Leino, P. Müller, Object invariants in dynamic contexts, in: ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings, 2004, pp. 491–516. doi:10.1007/978-3-540-24851-4_22.
[73] K. R. M. Leino, W. Schulte, Exception safety for c#, Proceedings of the Second International Conference on Software Engineering and Formal Methods, 2004. SEFM 2004. (2004) 218–227.
[74] P. Chalin, Are the logical foundations of verifying compiler prototypes matching user expectations?, Formal Aspects of Computing 19 (2) (2007) 139–158. doi:10.1007/s00165-006-0016-1.
[75] P. Chalin, F. Rioux, Jml runtime assertion checking: Improved error reporting and efficiency using strong validity, FM 2008: Formal Methods (2008) 246–261doi:10.1007/978-3-540-68237-0_18.
[76] M. Barnett, D. A. Naumann, W. Schulte, Q. Sun, 99.44% pure: Useful abstractions in specifications, in: ECOOP workshop on Formal Techniques for Java-like Programs (FTfJP), 2004. doi:10.1.1.72.3429.
[77] M. Barnett, W. Schulte, Runtime verification of .net contracts, Journal of Systems and Software 65 (3) (2003) 199–208. doi:10.1016/S0164-1212(02)00041-9.
[78] C. Flanagan, Hybrid types, invariants, and refinements for imperative objects, in: In International Workshop on Foundations and Developments of Object-Oriented Languages, 2006.
[79] J. Voigt, W. Irwin, N. Churcher, Comparing and Evaluating Existing Software Contract Tools, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 49–63. doi:10.1007/978-3-642-32341-6_4.
[80] Y. Falcone, S. Krstić, G. Reger, D. Traytel, A Taxonomy for Classifying Runtime Verification Tools, in: RV 2018 - 18th International Conference on Runtime Verification, Limassol, Cyprus, 2018, pp. 1–18.
URL https://hal.inria.fr/hal-01882410
[81] P. O. Meredith, D. Jin, D. Griffith, F. Chen, G. Roşu, An overview of the mop runtime verification framework, International Journal on Software Tools for Technology Transfer 14 (3) (2012) 249–289. doi:10.1007/s10009-011-0198-6.
[82] M. Gorbovitski, T. Rothamel, Y. A. Liu, S. D. Stoller, Efficient runtime invariant checking: A framework and case study, in: Proceedings of the 6th International Workshop on Dynamic Analysis (WODA 2008), ACM Press, 2008. doi:10.1145/1401827.1401837.
[83] M. Fähndrich, M. Barnett, F. Logozzo, Embedded contract languages, in: Proceedings of the 2010 ACM Symposium on Applied Computing, ACM, 2010, pp. 2103–2110. doi:10.1145/1774088.1774531.
[84] R. B. Findler, M. Felleisen, Contract soundness for object-oriented languages, in: ACM SIGPLAN Notices, Vol. 36, ACM,

2001, pp. 1–15. doi:10.1145/504311.504283.
[85] T. S. Strickland, S. Tobin-Hochstadt, R. B. Findler, M. Flatt, Chaperones and impersonators: run-time support for reasonable interposition, in: Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012, 2012, pp. 943–962. doi:10.1145/2384616.2384685.
[86] A. Takikawa, D. Feltey, E. Dean, M. Flatt, R. B. Findler, S. Tobin-Hochstadt, M. Felleisen, Towards practical gradual typing, in: LIPIcs-Leibniz International Proceedings in Informatics, Vol. 37, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015. doi:10.4230/LIPIcs.ECOOP.2015.4.
[87] R. Stephens, Beginning Software Engineering, Wiley, 2015.

A. Invariant Protocol Proof and Type System Requirements

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

Auxiliary Definitions

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

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

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

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

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

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

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

Thus the $MROG$ of l are the objects that could be mutated via a reference to l.

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

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

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

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

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

Now we can define what it means for an l to be *mutatable*[36] by an expression e: something reachable from l can also be reached by using a **mut** or **capsule** reference in e, and traversing through any number of **mut** or **rep** fields:

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

- $l' \in ROG(\sigma, l)$,
- $\mu\, l'' \in e$ with $\mu \in \{\texttt{mut}, \texttt{capsule}\}$, and
- $l' \in MROG(\sigma, l'')$.

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

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

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

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

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

Type System Requirements

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

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

Requirement 1 (Type Consistency)**.**

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

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

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

Requirement 2 (Imm Consistency)**.**

If $\mathit{validState}(\sigma, \mathcal{E}[e])$ and $\mathit{immutable}(\sigma, e, l)$, then not $\mathit{mutatable}(\sigma, e, l)$.

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

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

Requirement 3 (Mut Consistency)**.**

If $\mathit{validState}(\sigma, \mathcal{E}[e])$, $l \in \mathit{dom}(\sigma)$, not $\mathit{mutatable}(\sigma, e, l)$, and $\sigma|e \to^* \sigma'|e'$, then not $\mathit{mutatable}(\sigma', e', l)$.

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

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

Requirement 4 (Capsule Consistency).

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

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

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

Requirement 5 (Mut Update).

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

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

Requirement 6 (Strong Exception Safety).

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

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

Useful Lemmas

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

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

Lemma 1 (No Dangling).

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

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

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

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

Lemma 2 (Lost Forever).

If $validState(\sigma, \mathcal{E}[e])$, and $\sigma|e \rightarrow^* \sigma'|e'$, then

$\forall l \in dom(\sigma)$, if not $reachable(\sigma, e, l)$, then not $reachable(\sigma', e', l)$.

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

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

Lemma 3 (Non-Mutating).

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

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

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

Lemma 4 (Determinism).

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

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

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

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

Rep Field Soundness

Now we define and prove important properties about our novel `rep` fields. We first start with a few core auxiliary definitions. To simplify the notation, we define the *repFields* of an l to be the set of `rep` field names for l:

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

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

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

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

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

We use $\sigma \setminus l$ to remove the location l, thus $(\sigma, l \mapsto C\{ls\}) \setminus l = \sigma$.

We say that an l and f is *confined* if $l.f$ is not *mutatable* without passing through l:

$confined(\sigma, l, f)$ iff not $mutatable(\sigma \setminus l, e, \sigma[l.f])$.

We say that an l is *repConfined* if each of its `rep` fields are *confined*:

$\forall f \in repFields(\sigma, l)$ we have $confined(\sigma, l, f)$.

We say that an l is *repMutating* if we are in a monitor for l which must have been introduced by CALL MUTATOR:

$repMutating(\sigma, e, l)$ iff $e = \mathcal{E}[\texttt{M(}l\texttt{;}\ e'\texttt{;}\ \_\texttt{)}]$, with $e' \neq \texttt{mut}\,l$.

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

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

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

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

Theorem 2 (Rep Field Soundness)**.**

If $\mathit{validState}(\sigma, e)$ then $\forall l$ with $\mathit{reachable}(\sigma, e, l)$, we have:

- not $\mathit{repCircular}(\sigma, l, f)$, and
- either:
  - $\mathit{repConfined}(\sigma, l)$ and not $\mathit{repMutating}(\sigma, e, l)$, or
  - $\mathit{headNotObservable}(\sigma, e, l)$.

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

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

1. (NEW/NEW TRUE) $\sigma'|\mathcal{E}_v[\texttt{new}\ C\texttt{(}\mu_1\, l_1, .., \mu_n\, l_n\texttt{)}] \to \sigma|\mathcal{E}_v[e']$, where $\sigma = \sigma', l_0 \mapsto C\{l_1, .., l_n\}$, and by Type Cosnsistency, we have $\texttt{class}\ C\ \texttt{implements}\ \_\ \texttt{\{}\kappa_1\ \_\ f_1, .., \kappa_n\ \_\ f_n\texttt{;\}}$.
   (a) We have that l_0 is not *repCircular*: by No Dangling, we have that $\forall l' \in \mathit{dom}(\sigma')$, $\mathit{ROG}(\sigma', l') \subseteq \mathit{dom}(\sigma')$. By our notational conventions for ",", it follows that $l_0 \notin \mathit{dom}(\sigma')$. Now consider each $i \in [1, n]$, since the pre-existing σ' was not modified, it follows that $\mathit{ROG}(\sigma', l_i) = \mathit{ROG}(\sigma, \sigma[l_0.f_i])$. By No Dangling we have that $\mathit{ROG}(\sigma, \sigma[l_0.f_i]) \subseteq \mathit{dom}(\sigma)$, and so $l_0 \notin \mathit{ROG}(\sigma, \sigma[l_0.f_i])$, thus each $l_0.f_i$ is not *circular*.
   (b) Ever *reachable* $l' \neq l_0$ is not *repCircular*: Since reduction didn't modify the fields of any pre-existing l', by the inductive hypothesis, we have that l' is still not *repCircular*.
   (c) The new l_0 is *repConfined* and not *repMutating*:
      - Consider each $i \in [1, n]$ with $\kappa_i = \texttt{rep}$. By Type Consistency and Capsule Consistency, l_i was *encapsulated* and so $\mathit{ROG}(\sigma', l_i)$ cannot be *mutatable* from $\mathcal{E}_v$. Thus, we don't have $\mathit{mutatable}(\sigma \setminus l_0, \mathcal{E}_v[e'], l_i)$, and so each of l_0s `rep` fields is *confined*.
      - We trivially have that l_0 is not *repMutating* since $l_0 \notin \mathit{dom}(\sigma')$, by No Dangling, there can't be any monitor expressions for it in $\mathcal{E}_v$.
   (d) Every *reachable* $l' \neq l_0$ that was *repConfined* and not *repMutating* still is:
      - Suppose we have made it so that for some $f' \in \mathit{repFields}(\sigma', l')$, $l'.f'$ is no longer *confined*. Since we didn't modify the *ROG* of l' nor the *ROG* of any other pre-existing l'', we must have that $\sigma'[l'.f']$ is now *mutatable* through $l_0.f_i$, for some $i \in [1, n]$. This requires that l_i is an initialiser for a `mut` or `rep` field, which by Type Consistency means that $\mu_i \leq \texttt{mut}$. But then $\sigma'[l'.f']$ was already *mutatable* through $\mu_i\, l_i$, so $l'.f'$ can't have already been *confined*, a contradiction.
      - We can't have caused l' to be *repMutating* since we haven't introduced any monitor expressions, nor modified any existing ones.
   (e) Every *reachable* $l' \neq l_0$ is *headNotObservable*: by No Dangling, $l' \in \mathit{dom}(\sigma')$, so by Lost Forever, l' must have already been *reachable*. Thus, by the inductive hypothesis, l' must be *headNotObservable*, but we haven't removed any monitor expression or field accesses (because the arguments to the constructor are all fully reduced values), thus l' is still *headNotObservable*.
2. (ACCESS) $\sigma|\mathcal{E}_v[\mu\, l.f] \to \sigma|\mathcal{E}_v[\mu{::}\kappa\, \sigma[l.f]]$, where $C_l^\sigma.f = \kappa\ \_\ f$:
   (a) No *reachable* l' is *repCircular*: this holds by the inductive hypothesis and the fact that we haven't mutated memory.
   (b) If l is *reachable* and it was *repConfined* and not *repMutating*, than it still is:
      - If $\kappa \neq \texttt{rep}$, then we can't have broken *confined* for any $f' \in \mathit{repFields}(\sigma, l)$, since by definition of *repConfined*, $\sigma[l.f']$ can't have been *mutatable* through $\sigma[l.f]$.

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

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

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

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

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

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

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

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

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

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

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

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

- Suppose $l'' = l$ and $f \in repFields(\sigma', l)$, by Type Consistency and Capsule Consistency, l' is *encapsulated*, thus l' is not *mutatable* from $\mathcal{E}_v$, and l is not *reachable* from l'. Hence l' is still *encapsulated*, and so $l.f$ is still *confined*.

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

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

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

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

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

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

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

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

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

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

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

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

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

- By definition of *error*, we have $e = \mathcal{E}'_v[\texttt{M(}l\texttt{;}\ v\texttt{;}\ v'\texttt{)}]$.
- If the monitor was introduced by NEW or UPDATE, then $v = \texttt{mut}\,l$. And so *headNotObservable* can't have held for l since $l = l'$, and v was not the receiver of a field access. Thus by the inductive hypothesis, l must have been *repConfined* and not *repMutating*, a contradiction.
- By definition of *validState* and our well-formedness rules on method bodies, we must have that monitor must introduced by CALL MUTATOR, due to a call to a rep mutator on l.[37]
- From our reduction rules, it follows that we were previously in a state $\sigma_i|e_i$, where $i \in [1, m-1]$, e_i is of form $\mathcal{E}''_v[e'']$, and the next state was obtained by said application of the CALL MUTATOR rule to e''.
- Moreover, it follows that $\mathcal{E}''_v = \mathcal{E}_v[\texttt{try}^{\sigma'}\texttt{\{}\mathcal{E}'_v\texttt{\}}\ \texttt{catch}\ \texttt{\{}e'\texttt{\}}]$, as no reduction rules modify the $\mathcal{E}_v$.
- We must not have had that l was *headNotObservable*, since e'' would contain l as the receiver of a method call. Thus, by our inductive hypothesis, in state i, l was *repConfined* and not *repMutating*.
- By Strong Exception Safety and No Dangling, every l' *reachable* from $\mathcal{E}_v[e']$ has not been mutated, i.e. $\sigma(l') = \sigma_i(l') = \sigma'(l)$.
- Since nothing *reachable* has been mutated, it follows that l is still *repConfined*.
- By *validState* and our well-formedness rules on method bodies, it follows that e' contains no monitor expressions.
- Moreover, since l was not *repMutating* in $\mathcal{E}_v[\texttt{try}^{\sigma'}\texttt{\{}\mathcal{E}'_v[e'']\texttt{\}}\ \texttt{catch}\ \texttt{\{}e'\texttt{\}}]$, and e' contains no monitors, l it follows that l is not *repMutating* in $\mathcal{E}_v[e']$.

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

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

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

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

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

- If the monitor was introduced by NEW or UPDATE, then $\mu\, l' = \texttt{mut}\,l$. And so *headNotObservable* can't have held for l since $l = l'$, and v was not the receiver of a field access. Thus by the inductive hypothesis, l must have been *repConfined* and not *repMutating*, a contradiction.
- By definition of *validState* and our well-formedness rules on method bodies, we must have that monitor must introduced by CALL MUTATOR, due to a call to a rep mutator on l.
- From our reduction rules, it follows that we were previously in a state $\sigma_i|e_i$, where $i \in [1, m-1]$, e_i is of form $\mathcal{E}'_v[e']$, and the next state was obtained by said application of the CALL MUTATOR rule to e'.
- Moreover, it follows that $\mathcal{E}'_v = \mathcal{E}_v$, as no reduction rules modify the $\mathcal{E}_v$.
- We must not have had that l was *headNotObservable*, since e' would contain l as the receiver of a method call. Thus, by our inductive hypothesis, in state i, l was *repConfined* and not *repMutating*.

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

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

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

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

Stronger Soundness

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

We say that an object is *monitored* if execution is currently inside of a monitor for that object, and the monitored expression e_1 does not contain a reference to l as a *proper* sub-expression:
$monitored(e, l)$ iff $e = \mathcal{E}_v[\texttt{M(}l\texttt{;}\ e'\texttt{;}\ \_\texttt{)}]$ and $l \in e'$ only if $e' = \_\ l$.
A monitored object is associated with an expression that cannot observe it, but may reference its internal representation directly. In this way, we can safely modify its representation before checking its invariant. The idea is that at the start the object will be valid and e' will reference l; but during reduction, l will be used to modify the object, but not observe it; only after that moment, the object may become invalid.

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

Theorem 3 (Stronger Soundness)**.**
If $validState(\sigma, e)$ then $\forall l$, if $reachable(\sigma, e, l)$, then $valid(\sigma, l)$ or $monitored(e, l)$.

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

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

1. (NEW) $\sigma'|\mathcal{E}_v[\texttt{new}\ C\texttt{(\_}\,l_1, .., \texttt{\_}\,l_n\texttt{)}] \to \sigma', l_0 \mapsto C\{l_1, .., l_n\}|\mathcal{E}_v[\texttt{M(}l_0\texttt{; mut}\,l_0\texttt{; (read}\,l_0\texttt{).invariant())}]$:
   - Clearly the newly created object, l, is *monitored*.
   - This rule does not modify pre-existing memory, introduce pre-existing ls into the main expression, nor remove monitors on other ls, by the inductive hypothesis, every $l' \neq l_0$ is still *valid* (due to Determinism), or *monitored*.
2. (NEW TRUE) $\sigma'|\mathcal{E}_v[\texttt{new True()}] \to \sigma', l_0 \mapsto \texttt{True}\{\}|\mathcal{E}_v[\texttt{mut}\,l_0]$:
   - The $\texttt{True}$ class is required to have an invariant of $\texttt{new True()}$, so as with c in the base case above, we have that l_0 is *valid*.
   - As in the above case for NEW, since we didn't modify pre-existing memory, introduce pre-existing ls into the main expression, nor remove monitors, by the inductive hypothesis, every $l' \neq l_0$ is still *valid* or *monitored*.
3. (UPDATE) $\sigma'|\mathcal{E}_v[\mu\, l\texttt{.}f\texttt{=}v] \to \sigma|\mathcal{E}_v[e']$, where $e' = \texttt{M(}l\texttt{; mut}\,l\texttt{; (read}\,l\texttt{).invariant())}$:
   - Clearly l is now *monitored*.
   - Consider any other l', where $l \in ROG(\sigma', l')$ and l' was *valid*; now suppose we just made l' in*valid*. By our well-formedness criteria, $\texttt{invariant()}$ can only accesses $\texttt{imm}$ and $\texttt{rep}$ fields, thus by Non-Mutating, and Determinism, we must have that l was in the ROG of $\sigma'[l'\texttt{.}f']$, for some $f' \in repFields(\sigma', l')$.

     Since $l \neq l'$, l' can't have been *repConfined*. Thus, by Rep Field Soundness, l' was *headNotObservable*, and so $\mathcal{E}_v[\mu\, l\texttt{.}f\texttt{=}v]$ is of form $\mathcal{E}'_v[\texttt{M(}l'\texttt{;}\ e''\texttt{;}\ e'''\texttt{)}]$:
     - As the ROG of l' has just been mutated, and since e''' was must have come from the reduction of $\texttt{(read}\,l'''\texttt{).invariant()}$, if follows from Determinism, that we cannot currently be inside e'''.
     - Thus, $\mathcal{E}_v = \mathcal{E}'_v[\texttt{M(}l'\texttt{;}\ \mathcal{E}''_v\texttt{;}\ e'''\texttt{)}]$, where $\mathcal{E}''_v[\mu\, l\texttt{.}f\texttt{=}v] = e''$.
     - Suppose that l' was not *reachable* in e'', then clearly $l' \notin e''$, since $l' \neq l$, it follows that $l' \notin \mathcal{E}''_v[e']$, and so l' is *monitored*.
     - Otherwise, by definition of *headNotObservable*, we have that $e'' = \mathcal{E}[\texttt{mut}\,l'\texttt{.}f'']$ for some $f'' \in repFields(\sigma', l')$, and where l' is not *reachable* in $\mathcal{E}$.
     - By the proof for the TRY ERROR case of Rep Field Soundness, the monitor must have come from a call to a rep mutator, in a state where l' was *repConfined*. Thus, we were previously in a state $\sigma_i|e_i$, for some $i \in [0, m-1]$, immediately after a CALL MUTATOR; moreover, e_i is of form $\mathcal{E}'_v[\texttt{M(}l'\texttt{;}\ e'_i\texttt{;}\ \texttt{\_)}]$, immediately after a CALL MUTATOR, where e'_i is of form $\mathcal{E}'[\texttt{mut}\,l'\texttt{.}f''']$.
     - By Rep Field Soundness, l' is not *reachable* through $\sigma'[l'\texttt{.}f''']$,. By the proof for the CALL/CALL MUTATOR case of Rep Field Soundness, we have that l' is not *reachable* through $\mathcal{E}'$. Thus, by Lost Forever, once $\texttt{mut}\,l'\texttt{.}f'''$ has been reduced, l' must be un*reachable*, and it follows that $\texttt{mut}\,l'\texttt{.}f'' = \texttt{mut}\,l'\texttt{.}f'''$
     - By Mut Update, l is *mutatable* in the current state, thus by Mut Consistency, we have that it was also *mutatable* when CALL MUTATOR rule was applied. But we have that l' was *repConfined*, so since $l \in ROG(\sigma', \sigma'[l'\texttt{.}f'])$, we have that l can only be *mutatable* through l'.
     - By Lost Forever, the only way we could have obtain a reference to l was by reducing $\texttt{mut}\,l'\texttt{.}f''$, but we haven't done that yet, a contradiction.
   - Every other *valid* l', where $l \notin ROG(\sigma', l')$ is still *valid* by Determinism.

- As in the above case, since we don't remove any monitors, any other l' that was *monitored*, is still *monitored*.

4. (TRY ERROR) $\sigma|\mathcal{E}_v[\texttt{try}^{\sigma'}\texttt{\{}e\texttt{\}}\ \texttt{catch}\ \texttt{\{}e'\texttt{\}}] \rightarrow \sigma|\mathcal{E}_v[e']$, where $error(\sigma, e) = \mathcal{E}'_v[\texttt{M(}l\texttt{;}\ \_\texttt{;}\ \_\texttt{)}]$:
   - As with the case for TRY ERROR in the proof of Rep Field Soundnes, we were previously in a state $\sigma_i|e_i$, where $e_i = \mathcal{E}_v[\texttt{try}^{\sigma'}\texttt{\{\_\}}\ \texttt{catch}\ \texttt{\{\_\}}]$, and $\sigma_i = \sigma'$.
   - By definition of *error*, we have that l is not *valid* in σ, since monitor expressions always start of as an invariant calls.
   - Suppose l is still *reachable* in $\sigma|\mathcal{E}_v[e']$, by Strong Exception Safety, we have $l \in dom(\sigma')$. Thus by the inductive hypothesis, we have that l was *valid* or *monitored* in the state $\sigma'|e_i$.
   - If l was *monitored*, then by *validState* and our well-formedness rules on method bodies, said monitor must have been introduced by the NEW, UPDATE, or CALL MUTATOR reduction rules.
   - The NEW and UPDATE rules monitor a value, which cannot reduce to a `try`–`catch`, so the monitor must have been introduced by CALL MUTATOR.
   - But by our well-formedness rules on rep mutators, the body of the called method cannot mention l except to read a field, as shown in the case for UPDATE above, l will be un*reachable* once the field access has been reduced, which by Lost Forever is a contradiction, as l is *reachable* through e.
   - Thus, l can't have been *monitored* in $\sigma'|e_i$, so it must have been *valid*.
   - Also by Strong Exception Safety, we have that nothing reachable from l could have been modified, that is $\forall l' \in ROG(\sigma', l)$, we have $\sigma'(l') = \sigma(l')$. By Lost Forever, and our reduction rules, any memory location not *reachable* from a call $\texttt{(read}\,l\texttt{).invariant()}$ cannot affect its reduction.
   - Thus, by Determinism, and the fact that l was valid in σ', we have that l is still *valid*, a contradiction.
   - Thus, l cannot be *reachable*, so the fact that it is in*valid* is irrelevant.
   - As in the above case for NEW, since we didn't modify any memory, or remove any other monitors, by the inductive hypothesis every $l' \neq l$ is still *valid* or *monitored*.

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

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

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 5 (Imm Not Circular)**.**
If $validState(\sigma, e)$, $\forall f, l$, if $reachable(\sigma, e, l)$, $\mathrm{C}^{\sigma}_{l}.f = \texttt{imm}\ \_\ f$, then $l \notin ROG(\sigma, \sigma[l.f])$.

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

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

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

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

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

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

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

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

B. Example Type System and Proof of Requirements

In this section we formalise a lightweight version of the L42 type system. We then prove that it satisfies the requirements in Appendix A, and hence soundly supports our invariant protocol. This demonstrates that our protocol can be satisfied by a realistic type system.

New Notations

First we define the usual subclass hierarchy:
$C \leq C'$ iff:

- $C' = C$,
- $\exists C''$ with $C \leq C''$ and $C'' \leq C'$, or
- we have `class` C `implements` Cs `{_;_}` or `interface` C `implements` Cs `{_}` and $C' \in Cs$.

Then we define subtyping:
$\mu\, C \leq \mu'\, C'$ iff $\mu \leq \mu'$ and $C \leq C'$

Recall our definition for $\mu \leq \mu'$:

- $\mu \leq \mu$, for any μ
- `imm` $\leq$ `read`
- `mut` $\leq$ `read`
- `capsule` $\leq$ `mut` and `capsule` $\leq$ `imm`, and `capsule` $\leq$ `read`

$$\text{(TSub)}\ \frac{\sigma;\Gamma\vdash e : T}{\sigma;\Gamma\vdash e : T'}\ T\le T' \qquad \text{(TVar)}\ \frac{}{\sigma;\Gamma\vdash x : \Gamma(x)} \qquad \text{(TRef)}\ \frac{}{\sigma;\Gamma\vdash \mu\, l : \mu\, \mathrm{C}_l^{\sigma}}$$

$$\text{(TNew)}\ \frac{\sigma;\Gamma\vdash e_1 : \widetilde{\kappa_1}\, C_1 \quad \dots \quad \sigma;\Gamma\vdash e_n : \widetilde{\kappa_n}\, C_n}{\sigma;\Gamma\vdash \texttt{new}\ C\texttt{(}e_1,\dots,e_n\texttt{)} : \texttt{mut}\ C}\ \begin{array}{l}\texttt{class}\ C\ \texttt{implements}\ \_\ \texttt{\{}Fs\texttt{;}\ \_\texttt{\}}\\ Fs = \kappa_1\, C_1\ \_,\dots,\kappa_n\, C_n\ \_\end{array}$$

$$\text{(TAccess)}\ \frac{\sigma;\Gamma\vdash e : \mu\, C}{\sigma;\Gamma\vdash e\texttt{.}f : \mu{::}\kappa\, C'}\ C.f = \kappa\, C'\, f \qquad \text{(TUpdate)}\ \frac{\sigma;\Gamma\vdash e : \texttt{mut}\ C \quad \sigma;\Gamma\vdash e' : \widetilde{\kappa}\, C'}{\sigma;\Gamma\vdash e\texttt{.}f\ \texttt{=}\ e' : \texttt{mut}\ C}\ C.f = \kappa\, C'\, f$$

$$\text{(TCall)}\ \frac{\sigma;\Gamma\vdash e_0 : \mu\, C \quad \sigma;\Gamma\vdash e_1 : T_1 \quad \dots \quad \sigma;\Gamma\vdash e_n : T_n}{\sigma;\Gamma\vdash e_0\texttt{.}m\texttt{(}e_1,\dots,e_n\texttt{)} : T'}\ \begin{array}{l}S = \mu\, \texttt{method}\ T'\, m\texttt{(}T_1\ \_,\dots,T_n\ \_\texttt{)}\\ C.m \in \{S,\ S\ \_\}\end{array}$$

$$\text{(TAs)}\ \frac{\sigma;\Gamma\vdash e : \mu\, C}{\sigma;\Gamma\vdash e\ \texttt{as}\ \mu' : \mu'\, C}\ \mu\le\mu' \qquad \text{(TAsCapsule)}\ \frac{\sigma;\widehat{\Gamma}\vdash e : \texttt{mut}\ C}{\sigma;\Gamma\vdash e\ \texttt{as capsule} : \texttt{capsule}\ C}$$

$$\text{(TTryCatch1)}\ \frac{\sigma;\widehat{\Gamma}\vdash e : T \quad \sigma;\Gamma\vdash e' : T}{\sigma;\Gamma\vdash \texttt{try}\ \texttt{\{}e\texttt{\}}\ \texttt{catch}\ \texttt{\{}e'\texttt{\}} : T} \qquad \text{(TTryCatch2)}\ \frac{\sigma;\Gamma\vdash e : T \quad \sigma;\Gamma\vdash e' : T}{\sigma;\Gamma\vdash \texttt{try}^{\sigma'}\ \texttt{\{}e\texttt{\}}\ \texttt{catch}\ \texttt{\{}e'\texttt{\}} : T}$$

$$\text{(TMonitor)}\ \frac{\sigma;\Gamma\vdash e : T \quad \sigma;\Gamma\vdash e' : \mu\, \texttt{Bool}}{\sigma;\Gamma\vdash \texttt{M(}l\texttt{;}\ e\texttt{;}\ e'\texttt{)} : T}\ l\in dom(\sigma)$$

Figure B.3: Type rules

Note that $\mu\le\mu'$, $C\le C'$, and $T\le T'$ are all reflexive and transitive.

Now we define a notation that converts `mut` reference capabilities to `read`:
$\widehat{\texttt{mut}} = \texttt{read}$ and $\widehat{\mu} = \mu$, if $\mu \ne \texttt{mut}$
Note that we always have $\mu \le \widehat{\mu}$ and $\widehat{\widehat{\mu}} = \widehat{\mu}$

We extend this to convert all `mut` variables in an typing environment to `read`:
$\widehat{\Gamma}(x) = \widehat{\mu}\, C$ iff $\Gamma(x) = \mu\, C$
Note that we always have $\widehat{\emptyset} = \emptyset$, $\widehat{\widehat{\Gamma}} = \widehat{\Gamma}$, and $\Gamma(x) \le \widehat{\Gamma}(x)$.

We also extend this to convert all `mut` references in an expression to `read`:
$\widehat{e} = e[\mu_1\, l_1 \coloneqq \widehat{\mu_1}\, l_1,\dots,\mu_n\, l_n \coloneqq \widehat{\mu_n}\, l_n]$, where $\{\mu_1\, l_1,\dots,\mu_n\, l_n\} = \{v \in e\}$

Finally, we define a notation to mean that two expressions are identical, except perhaps for reference capability annotations on references:
$e \sim e'$ iff $e[\mu_1\, l_1 \coloneqq \texttt{read}\, l_1,\dots,\mu_n\, l_n \coloneqq \texttt{read}\, l_n] = e'[\mu_1\, l_1 \coloneqq \texttt{read}\, l_1,\dots,\mu_n\, l_n \coloneqq \texttt{read}\, l_n]$,
where $\{\mu_1\, l_1,\dots,\mu_n\, l_n\} = \{v \in e\} \cup \{v \in e'\}$.
Note that the above requires that the μs of an `as` expression are the same, i.e. $e\ \texttt{as}\ \mu \sim e'\ \texttt{as}\ \mu'$ only if $\mu = \mu'$.

Type System

We present the typing rules in Figure B.3:

- TSub is the standard "subsumption" rule, an expression with a type T also has any supertype T', in particular this works with our reference capabilities, e.g. an expression of type $\texttt{imm}\ C$ also has type $\texttt{read}\ C$.
- TVar simply looks up the type of an x in the environment Γ. Note that this requires that $x \in dom(\Gamma)$,

i.e. that there are no undefined variables.

- TREF types a reference with the given capability by looking up the memory σ to determine the appropriate class. Note that this requires that $l \in dom(\sigma)$, i.e. that there are no dangling pointers. However, it does *not* impose any restrictions on the reference capability μ, for example an expression with two `capsule` references with the same l is considered well-typed by our type system, the proofs of our various type system requirements ensure that such an expression cannot be a *validState*, i.e. they will not actually occur when reducing a valid initial program.

- TNEW types a `new` expression by checking that there is an initialising expression for each field f_i, that has the corresponding class C_i and capability $\widetilde{\kappa_i}$. See Section 5 for the definition of $\widetilde{\kappa}$.

- TACCESS types a field access expression by checking that the receiver has the given field. The μ::κ computes the resulting reference capability in the same way as the ACCESS reduction rule, although at runtime the result of the expression may have a more specific reference capability.

- TUPDATE types a field update expression by checking that the receiver has the given field, and the new value has the appropriate type. As with the NEW rule, we use $\widetilde{\kappa}$ to compute the required reference capability. This rule requires the receiver of the update to be typeable as `mut`, this ensures that only `mut` and `capsule` references can be used to mutate an object.

- TCALL types a method call by looking for the appropriate method/signature in the receivers class. If the receivers class is an interface, then $C.m$ will be of form S, otherwise it will be of form S _ and hence have a method body, but we do not use this extra information. We check that the receiver conforms to the reference capability of the method, and check that each argument conforms to the corresponding parameter type. Note that we don't need to know whether the called method is a rep mutator or not, as the runtime will only introduce an extra invariant check, and not alter the result of the method.

- TAS types an `as` expression that is trivially sound because the body of the expression conforms to the target reference capability. This allows the reference capability of an expression to be restricted, e.g. if $\mu' =$ `read`, the `as` expression cannot be used as the receiver of a field update, even if $\mu =$ `mut`.

- TASCAPSULE is the `capsule` promotion rule, it is the main way the type system is practical. As `as` expressions must have come from a method body, we will initially have $\emptyset; \widehat{\Gamma} \vdash e : \texttt{mut}\, C$, and so e will contain no references. In particular, this means that if e uses any `mut` variables in Γ it can only see them as `read`, in particular, our typing rules ensure that such a variable cannot be stored in the heap, nor can any part of its *ROG* be accessed as `mut` (because TACCESS will type such an access as `read` or `imm`). This is enough to ensure that once the variables in Γ have been substituted for values and the body is reduced to a value, no `mut` or `read` variables in Γ will be *reachable* from the result of e. Thus every object *reachable* from the result of e will be a newly created object, *immutable*, or *reachable* only through `capsule` variables in Γ. This ensures that the result is *encapsulated* as the non-*immutable* objects reachable from a `capsule` variable in Γ will not be *reachable* elsewhere in the program.

  During reduction, we will type the expression under $\sigma; \emptyset$, and so e may contain `mut` references, however this does not break our guarantees since we previously typed the expression under $\emptyset; \widehat{\Gamma}$, and so any such references must have been created during the reduction of e, and cannot have come from the $\widehat{\Gamma}$.

  The full L42 language supports more promotions, such as `read` to `imm`. These could be added to our type system, but would greatly complicate our proofs. The TASCAPSULE rule is sufficient to demonstrate that our invariant protocol can be supported in a system with promotions.

- TTRYCATCH1 types a `try`–`catch` expression that has yet to be reduced, similar to the TASCAPSULE rule, we require the `try` part to be typeable under $\widehat{\Gamma}$. This ensures strong exception safety as $\widehat{\Gamma}$ contains no `mut` variables, and so the only way e can obtain a `mut` reference is from a `capsule` variable or a freshly created object. In addition, since the only preexisting objects that can be seen as `mut` are those *reachable* from `capsule` variables in Γ, there is no way for e to store any state in a place that e' could observe it.

- TTRYCATCH2 is used to type annotated `try–catch` expressions during reduction, as such expression cannot occur in method bodies, we will always have $\Gamma = \emptyset$. As with the TASCAPSULE typing rule, since `try–catch` expressions can only be introduced through method calls, we don't need extra type restrictions. In particular, the check that $\emptyset; \widehat{\Gamma} \vdash e : T$ holds from within a method body is sufficient to reason over `try–catch`es in the main expression.

- MONITOR type checks monitor expressions introduced by reduction, the l will refer to the monitored object, e will compute the result of the entire expression (provided the invariant check succeeds) and the e' will be the `invariant` check itself. Note that e will be computed *before* e'. The side condition on l is not strictly needed as it follows directly from No Dangling. Note that from our signature of the `invariant` method and Type Preservation below, e' will always have type `imm Bool`, however we need to allow an arbitrary μ for our Bisimulation lemma below.

We use the above typing rules to type-check each method against their declared return type, under the assumption that their parameters and receiver have the appropriate type. We also require that each method use a `capsule` parameter at most once. Formally, we require that:

$\forall C_0, m$ if $C_0.m = \mu_0\,\mathtt{method}\,T\,m(\mu_1\,C_1\,x_1, ..., \mu_n\,C_n\,x_n)\;e$, we require:

- $\emptyset; \mathtt{this} \mapsto \mu_0\,C_0, x_1 \mapsto \mu_1\,C_1, ..., x_n \mapsto \mu_n\,C_n \vdash e : T$, and
- $\forall i \in [1, n]$, if $\mu_i = \mathtt{capsule}$, then $\forall \mathcal{E}$ with $e = \mathcal{E}[x]$, $x \notin \mathcal{E}$.

Finally, we define a $\vdash \sigma$ notation to verify that memory respects the class table.

$\vdash \sigma$ iff $\forall l_0 \in dom(\sigma)$:

- $\sigma(l_0) = C_0\{l_1, ..., l_n\}$,
- we have `class` C_0 `implements` _ `{`Fs`;`_`}`,
- $Fs =$ _ C_1 _, ..., _ C_n _, and
- $\mathrm{C}^{\sigma}_{l_1} \leq C_1, ..., \mathrm{C}^{\sigma}_{l_n} \leq C_n$.

Thus $\vdash \sigma$ ensures that there are no dangling pointers, each object has a proper class (and not an interface), they have the appropriate number of fields, and each field value has an appropriate class. Note that $\vdash \sigma$ doesn't require the field kinds are respected, this is ensured by the below proofs of our type system requirements.

Lemmas

Often we need to use the properties guaranteed by the type-rules for a specific form of expression, to this aim we define a slightly different typing judgement that excludes the TSUB rule:

$\sigma; \Gamma \vdash e :: T$ iff $\sigma; \Gamma \vdash e : T$ holds by a rule other than TSUB.

Note that $\sigma; \Gamma \vdash e :: T$ may still use TSUB for the *sub*expressions of e.

Now we prove that we can always extract a $\sigma; \Gamma \vdash e :: T$ from a $\sigma; \Gamma \vdash e : T'$ judgement:

Lemma 6 (Type Rule)**.**

$\sigma; \Gamma \vdash e : T$ holds if and only if $\sigma; \Gamma \vdash e :: T'$ holds for some $T' \leq T$

Proof. The "only if" direction holds directly from induction on the length of the type derivation of $\sigma; \Gamma \vdash e : T$ and the fact that $\leq$ is transitive. The "if" direction holds trivially since $\sigma; \Gamma \vdash e :: T'$ implies $\sigma; \Gamma \vdash e : T'$, and then TSUB can be used to get $\sigma; \Gamma \vdash e : T$ ☐

This lemma means that if we know the syntactic form of a well-typed expression e, we can use Type Rule to determine which of the non-TSUB rules must have applied.

Now we show that the type system types references according to their reference capability and class:

Lemma 7 (Ref Type)**.**

$\sigma; \emptyset \vdash \mu\,l : T$ if and only if $\mu\,\mathrm{C}^{\sigma}_{l} \leq T$.

Proof. Follows immediately from Type Rule and the TREF and TSUB typing rules. $\square$

We note that if an expression is well-typed, then each subexpression must also be well-typed. Note that the proof is non-trivial as we sometimes type a subexpression under $\widehat{\Gamma}$ and not Γ.

Corollary 1 (Nested Type)**.**

If $\sigma;\Gamma \vdash \mathcal{E}[e] : T$, then $\sigma;\Gamma \vdash e :: T'$, for some T'.

Proof. We prove this by induction on the size of $\mathcal{E}$. The base case follows trivially from Type Rule.

In the inductive case, by Type Rule and the structure of our typing rules, we have $\mathcal{E} = \mathcal{E}'[\mathcal{E}'']$ where $\mathcal{E}'' \neq \square$ and is otherwise minimal. By the inductive hypothesis, we have that $\sigma;\Gamma \vdash \mathcal{E}''[e] :: T''$ holds for some T''. Since e is a direct subexpression of $\mathcal{E}''$, each such rule has a premise of form $\sigma;\Gamma \vdash e : T'''$ or $\sigma;\widehat{\Gamma} \vdash e : T'''$, for some T'''.

If $\sigma;\widehat{\Gamma} \vdash e : T'''$, we can turn such a typing derivation into one for $\sigma;\Gamma \vdash e : T'''$,

by replacing each occurrence of a $(\textsc{TVar})\ \dfrac{}{\sigma;\widehat{\Gamma} \vdash x : \widehat{\Gamma}(x)}$ with $(\textsc{TSub})\ \dfrac{(\textsc{TVar})\ \dfrac{}{\sigma;\Gamma \vdash x : \Gamma(x)}}{\sigma;\Gamma \vdash x : \widehat{\Gamma}(x)}$.

The side condition for TSUB trivially holds as we always have $\Gamma(x) \leq \widehat{\Gamma}(x)$. Note that this works even if the typing derivation for $\sigma;\widehat{\Gamma} \vdash e : T'''$ itself uses the TASCAPSULE or TTRYCATCH1 rules, since $\widehat{\widehat{\Gamma}} = \widehat{\Gamma}$.

Thus we have $\sigma;\Gamma \vdash e : T'''$, and so by Type Rule, we have $\sigma;\Gamma \vdash e :: T'$, for some T'. $\square$

Now we show that if we have a $\sigma;\Gamma \vdash e : T$ then we can substitute each variable in $dom(\Gamma)$ with an appropriate reference, and e will still have type T:

Lemma 8 (Substitution)**.**

If $dom(\Gamma) = \{x_1, .., x_n\}$, $\emptyset;\Gamma \vdash e : T$, and $\mu_1\, \mathrm{C}^{\sigma}_{l_1} \leq \Gamma(x_1), .., \mu_n\, \mathrm{C}^{\sigma}_{l_n} \leq \Gamma(x_n)$,

then $\sigma;\emptyset \vdash e[x_1 \coloneqq \mu_1\, l_1, .., x_n \coloneqq \mu_n\, l_n] : T$.

Proof. Let $e' = e[x_1 \coloneqq \mu_1\, l_1, .., x_n \coloneqq \mu_n\, l_n]$. The proof then follows by induction on the size of the typing derivation applied to obtain $\sigma;\Gamma \vdash e : T$. We then proceed by cases on the typing rule that gave us $\sigma;\Gamma \vdash e : T$, show that we con obtain $\sigma;\emptyset \vdash e' : T$:

- Suppose the TVAR typing rule applied, i.e. $e = x$ and $T = \Gamma(x)$. Thus there is some $i \in [1, n]$ with $x_i = x$ and $e' = \mu_i\, l_i$. By the TREF typing rule, we have $\sigma;\emptyset \vdash e' : \mu_i\, \mathrm{C}^{\sigma}_{l_i}$. Since $\mu_i\, \mathrm{C}^{\sigma}_{l_i} \leq \Gamma(x_i)$, by the TSUB typing rule, we have $\sigma;\emptyset \vdash e' : \Gamma(x_i)$, as required.

- Suppose the TASCAPSULE typing rule applied, i.e. $e = e_0$ `as capsule` and $T =$ `capsule` C, for some e_0 and C, where $\emptyset;\widehat{\Gamma} \vdash e_0 :$ `mut` C. Thus $e' = e'_0$ `as capsule` where $e'_0 = e_0[x_1 \coloneqq \mu_1\, l_1, .., x_n \coloneqq \mu_n\, l_n]$.

  Note that $dom(\widehat{\Gamma}) = \Gamma$, and consider each $i \in [1, n]$, $\Gamma(x_i)$ will be of form $\mu'_i\, C_i$ where $\widehat{\Gamma}(x_i) = \widehat{\mu'_i}\, C_i$, $\mu_i \leq \mu'_i$, and $\mathrm{C}^{\sigma}_{l_i} \leq C_i$. Clearly $\mu'_i \leq \widehat{\mu'_i}$ and so $\mu_i \leq \widehat{\mu'_i}$, thus we have $\mu_i\, \mathrm{C}^{\sigma}_{l_i} \leq \widehat{\Gamma}(x_i)$.

  By the above and the inductive hypothesis, we have that $\sigma;\emptyset \vdash e'_0 :$ `mut` C. Thus by TASCAPSULE and the fact that $\widehat{\emptyset} = \emptyset$, we have $\sigma;\emptyset \vdash e'_0$ `as capsule` $:$ `capsule` C, as required.

- Suppose the TTRYCATCH1 typing rule applied, i.e. $e =$ `try {`e_0`} catch {`e_1`}` for some e_0 and e_1, where $\emptyset;\widehat{\Gamma} \vdash e_0 : T$ and $\emptyset;\Gamma \vdash e_1 : T$. Thus $e' =$ `try {`e'_0`} catch {`e'_1`}` where $e'_0 = e_0[x_1 \coloneqq \mu_1\, l_1, .., x_n \coloneqq \mu_n\, l_n]$ and $e'_1 = e_1[x_1 \coloneqq \mu_1\, l_1, .., x_n \coloneqq \mu_n\, l_n]$. By the above TASCAPSULE case and the inductive hypothesis, we have $\sigma;\emptyset \vdash e'_0 : T$. By the inductive hypothesis, we also have $\sigma;\emptyset \vdash e'_1 : T$. Thus by the TTRYCATCH1 rule we have $\sigma;\emptyset \vdash$ `try {`e'_0`} catch {`e'_1`}` $: T$.

- Suppose the TREF or TMONITOR rules applied, then we would have an $l \in dom(\emptyset)$, a contradiction.

- Otherwise, the TSUB, TUPDATE, TNEW, TACCESS, TTRYCATCH2, TCALL, or TAS typing rule applied. The side conditions of these rules (if any) do not depend on the Γ or σ, nor the xs or vs in the expression, thus the side conditions still hold for a conclusion of form $\sigma;\emptyset \vdash e' : T$.

  Now consider each premise of these rules (if any). Each such premise is of form $\emptyset;\Gamma \vdash e_0 : T_0$, where e_0 is a subexpression of e. Thus there is a corresponding subexpression e'_0 of e' such that

$e'_0 = e_0[x_1 \coloneqq \mu_1\, l_1, ..., x_n \coloneqq \mu_n\, l_n]$. Thus by the inductive hypothesis we have $\sigma; \emptyset \vdash e'_0 : T_0$, which is the corresponding premise for a conclusion of form $\sigma; \emptyset \vdash e' : T$.

Thus we can use the same typing rule to obtain a conclusion of form $\sigma; \emptyset \vdash e' : T$. ☐

We show that if a method call on fully reduced values is well-typed, the receiver and each argument satisfies the method signature, and once these have been substituted in, the body has the appropriate type.

Lemma 9 (Method Type)**.**
If $\vdash \sigma$ and $\sigma; \emptyset \vdash \mu_0\, l_0\texttt{.}m\texttt{(}\mu_1\, l_1, ..., \mu_n\, l_n\texttt{)} : T$, then:
1. $C^{\sigma}_{l_0}.m = \mu'_0\, \texttt{method}\, T'\, m\texttt{(}\mu'_1\, C_1\, x_1, ..., \mu'_n\, C_n\, x_n\texttt{)}\, e$,
2. $\mu_0 \leq \mu'_0$,
3. $\mu_1\, C^{\sigma}_{l_1} \leq \mu'_1\, C_1, ..., \mu_n\, C^{\sigma}_{l_n} \leq \mu'_n\, C_n$,
4. $\sigma; \emptyset \vdash e[\texttt{this} \coloneqq \mu'_0\, l_0, x_1 \coloneqq \mu'_1\, l_1, ..., x_n \coloneqq \mu'_n\, l_n] : T'$, and
5. $T' \leq T$.

Proof.

1. By Type Rule, the TCall typing rule rule applied, and so $\mu_0\, l_0$ is well-typed, and by Ref Type, $C^{\sigma}_{l_0}$ is well-defined. Moreover, by $\vdash \sigma$, we have that $C^{\sigma}_{l_0}$ is not an interface, so by our grammar, we have $C^{\sigma}_{l_0}.m = S\, e$ where $S = \mu'_0\, \texttt{method}\, T'\, m\texttt{(}\mu'_1\, C_1\, x_1, ..., \mu'_n\, C_n\, x_n\texttt{)}$ for some e.
2. By the TCall typing rule applied, so we have $\sigma; \emptyset \vdash \mu_0\, l_0 : \mu\, C$, for some μ and C. By Ref Type, we have $\mu_0 \leq \mu$ and $C^{\sigma}_{l_0} \leq C$.

   If C is an interface, then by our well-formedness rules on the class table, we have $C.m = S$. Otherwise, C is a class, and by our well-formedness rules on the class table, we have $C^{\sigma}_{l_0} = C$.

   Regardless, we have $C.m \in \{S,\, S\, e\}$. By the TCall typing rule, this means that $\mu = \mu'_0$, thus $\mu_0 \leq \mu'_0$.
3. Consider each $i \in [1, n]$. Since $C.m \in \{S,\, S\, e\}$, by the TCall rule we have $\sigma; \emptyset \vdash \mu_i\, l_i : \mu'_i\, C_i$. By Ref Type, we thus have $\mu_i\, C^{\sigma}_{l_i} \leq \mu'_i\, C_i$.
4. By our well-formedness rules on methods, we have $\emptyset; \Gamma \vdash e : T'$, where $\Gamma = \texttt{this} \mapsto \mu'_0\, C^{\sigma}_{l_0}, x_1 \mapsto \mu'_1\, C_1, ..., x_n \mapsto \mu'_n\, C_n$. Since $\mu_0\, C^{\sigma}_{l_0} \leq \mu'_0\, C^{\sigma}_{l_0}$ and $\mu_1\, C^{\sigma}_{l_1} \leq \mu'_1\, C_1, ..., \mu_n\, C^{\sigma}_{l_n} \leq \mu'_n\, C_n$, by Substiution, we have $\sigma; \emptyset \vdash e[\texttt{this} \coloneqq \mu'_0\, l_0, x_1 \coloneqq \mu'_1\, l_1, ..., x_n \coloneqq \mu'_n\, l_n] : T'$.
5. Finally, since $C.m \in \{S,\, S\, e\}$, by Type Rule and the TCall call rule, we have $T' \leq T$. ☐

We now present a lemma needed to reason over the types of monitor expressions.Monitor expressions starting with an invariant call are well-typed provided the body is well-typed.

Lemma 10 (Monitor Type)**.**
If $\vdash \sigma$, $l \in dom(\sigma)$, and $\sigma; \emptyset \vdash e : T$ then $\sigma; \emptyset \vdash \texttt{M(}l\texttt{;}\, e\texttt{;}\, \texttt{(read}\, l\texttt{).invariant())} : T$.

Proof. We can construct the following typing derivation:

$$\dfrac{\sigma; \emptyset \vdash e : T \qquad \dfrac{\dfrac{}{\sigma; \emptyset \vdash \texttt{read}\, l : \texttt{read}\, C^{\sigma}_{l}}\,(\text{TRef})}{\sigma; \emptyset \vdash \texttt{(read}\, l\texttt{).invariant()} : \texttt{imm}\, \texttt{Bool}}\,(\text{TCall})}{\sigma; \Gamma \vdash \texttt{M(}l\texttt{;}\, e\texttt{;}\, \texttt{(read}\, l\texttt{).invariant())} : T}\,(\text{TMonitor})$$

By our well-formedness rules on the class table, we have $C^{\sigma}_{l}\texttt{.invariant} = \texttt{read}\, \texttt{method}\, \texttt{imm}\, \texttt{Bool}\, \texttt{invariant()}\, \_$, since $\vdash \sigma$ ensures that C^{σ}_{l} is not an interface. Thus the side condition required by the TCall rule holds, as does the $l \in dom(\sigma)$ condition required by TMonitor. ☐

We now prove the standard soundness property of any type system: reduction respects the type of an expression. Note that this holds for any well-typed expression and well-formed memory, even those that are not *validState*. Note as discussed before, our type system does not directly verify the required properties of our reference capabilities (such as preventing simultaneous `imm` and `mut` references to the same object), rather we prove those separately below.

Theorem 4 (Type Preservation)**.**
If $\vdash \sigma$, $\sigma; \emptyset \vdash e : T$ and $\sigma | e \rightarrow^{n} \sigma' | e'$, then $\vdash \sigma'$ and $\sigma'; \emptyset \vdash e' : T$.

Proof. The proof is by induction on n. In the first base case, we assume $n = 0$ and the conclusion trivially holds since $\sigma' = \sigma$ and $e' = e$.

In the second base case, we assume $n = 1$, i.e. $\sigma|e \to \sigma'|e'$. We note by Type Rule that we have $\sigma; \emptyset \vdash e :: T'$, for some $T' \leq T$. We will then show that $\sigma'; \emptyset \vdash e' : T'$ holds by induction on the size of e.

In the base case for our inner induction, we assume that there is no $\mathcal{E}_v$ and e_0 where $\mathcal{E}_v \neq \square$ and $e = \mathcal{E}_v[e_0]$. We now proceed by cases on the reduction rule applied:

- Suppose that the NEW/NEW TRUE rule was applied, i.e. we have $e = \texttt{new}\ C\texttt{(}\mu_1\, l_1, \ldots, \mu_n\, l_n\texttt{)}$, $\sigma' = \sigma, l_0 \mapsto C\{l_1, \ldots, l_n\}$, and $e' \in \{\texttt{M(}l_0\texttt{;}\ \texttt{mut}\, l_0\texttt{;}\ \texttt{(read}\, l_0\texttt{).invariant())}, \texttt{mut}\, l_0\}$, and $l_0 = \mathit{fresh}(\sigma)$. By the TNew typing rule, we have $T' = \texttt{mut}\, C$, and a declaration $\texttt{class}\ C\ \texttt{implements}\ \_\ \texttt{\{}Fs\texttt{;}\ \_\texttt{\}}$ with $Fs = \kappa_1\, C_1\ \_, \ldots, \kappa_n\, C_n\ \_$.

  Now consider each $i \in [1, n]$, clearly $\mathrm{C}^{\sigma'}_{l_i} = \mathrm{C}^{\sigma}_{l_i}$, and by the TNew typing rule, we have $\sigma; \emptyset \vdash \mu_i\, l_i : \widetilde{\kappa_i}\ C_i$, and so by Ref Type we have $\mathrm{C}^{\sigma'}_{l_i} \leq C_i$.

  Furthermore, since $l_0 = \mathit{fresh}(\sigma)$, $l_0 \notin \mathit{dom}(\sigma)$, by the above and the fact that $\vdash \sigma$, we have $\vdash \sigma'$, as required.

  Clearly $\mathrm{C}^{\sigma'}_{l_0} = C$, so by Ref Type, we have $\sigma'; \emptyset \vdash \texttt{mut}\, l_0 : \texttt{mut}\, C$.

  If $e' = \texttt{mut}\, l_0$ then we are done. Otherwise, $e' = \texttt{M(}l_0\texttt{;}\ \texttt{mut}\, l_0\texttt{;}\ \texttt{(read}\, l_0\texttt{).invariant())}$, and by Monitor Type, we have $\sigma'; \emptyset \vdash e' : \texttt{mut}\, C$ as required.

- Suppose the ACCESS rule was applied, i.e. we have $e = \mathcal{E}_v[\mu\, l.f]$, $\sigma' = \sigma$, and $e' = \mu{::}\kappa\, \sigma[l.f]$, where $\mathrm{C}^{\sigma}_{l'}.f = \kappa\, C\, f$. By the TAccess typing rule, we have $\sigma; \emptyset \vdash \mu\, l : \mu'\, C'$ where C' is a class (since the side condition on TAccess requires C' to have a field). By Ref Type, we have $\mu \leq \mu'$ and $\mathrm{C}^{\sigma}_{l} \leq C'$. Since $\vdash \sigma$, C^{σ}_{l} is a class, so by our well-formedness rules on the class table, since C' is also a class, we have $\mathrm{C}^{\sigma}_{l} = C'$. Thus by the TAccess typing rule, since $\mathrm{C}^{\sigma}_{l'}.f = \kappa\, C\, f$, we have $T' = \mu'{::}\kappa\, C$.

  If $\kappa = \texttt{imm}$ then $\mu{::}\kappa = \mu'{::}\kappa = \texttt{imm}$ and so trivially $\mu{::}\kappa \leq \mu'{::}\kappa$. Otherwise, $\mu{::}\kappa = \mu$ and $\mu'{::}\kappa = \mu'$; since $\mu \leq \mu'$, we thus have $\mu{::}\kappa \leq \mu'{::}\kappa$.

  Since $\vdash \sigma$, we have $\mathrm{C}^{\sigma}_{\sigma[l.f]} \leq C$, and since $\mu{::}\kappa \leq \mu'{::}\kappa$, by Ref Type, we have $\sigma; \emptyset \vdash \mu{::}\kappa\, \sigma[l.f] : T'$, as required.

- Suppose the UPDATE rule was applied, i.e. we have $e = \mathcal{E}_v[\mu\, l.f = \mu'\, l']$, $\sigma' = \sigma[l.f = l']$, and $e' = \texttt{M(}l\texttt{;}\ \texttt{mut}\, l\texttt{;}\ \texttt{(read}\, l\texttt{).invariant())}$. By the TUpdate typing rule, we have $\sigma; \emptyset \vdash \mu\, l : \texttt{mut}\, C$, where $C.f = \kappa\, C'\, f$. As with the ACCESS case above, we have $\mathrm{C}^{\sigma}_{l} = C$. Thus by the TUpdate typing rule, we have $\sigma; \emptyset \vdash \mu'\, l' : \widetilde{\kappa}\, C'$ and $T' = \texttt{mut}\, C$.

  By Type Ref we have $\mathrm{C}^{\sigma}_{l'} \leq C'$. Clearly $\mathrm{C}^{\sigma'}_{l} = \mathrm{C}^{\sigma}_{l}$ and $\mathrm{C}^{\sigma'}_{l'} = \mathrm{C}^{\sigma}_{l'}$, and so we have $\mathrm{C}^{\sigma'}_{l}.f = \kappa\, C'\, f$ with $\mathrm{C}^{\sigma'}_{l'} \leq C'$. As σ' differs from σ only at $l.f$, by the above and the fact that $\vdash \sigma$, we have $\vdash \sigma'$.

  By Ref Type we have $\sigma; \emptyset \vdash \texttt{mut}\, l : \texttt{mut}\, C$, thus by Monitor Type, we have $\sigma; \emptyset \vdash e' : \texttt{mut}\, C$ as required.

- Suppose that the CALL/CALL MUTATOR rule was applied, i.e. we have $e = \_\ l_0.m\texttt{(}\_\ l_1, \ldots, \_\ l_n\texttt{)}$, $\sigma' = \sigma$, $e' \in \{e''\ \texttt{as}\, \mu', \texttt{M(}l_0\texttt{;}\ e''\ \texttt{as}\, \mu'\texttt{;}\ \texttt{(read}\, l_0\texttt{).invariant())}\}$, $e'' = e'''[\texttt{this} := \mu_0\, l_0, x_1 := \mu_1\, l_1, \ldots, x_n := \mu_n\, l_n]$, and $\mathrm{C}^{\sigma}_{l_0}.m = \mu_0\, \texttt{method}\, \mu'\, C\, m\texttt{(}\mu_1\ \_\ x_1, \ldots, \mu_n\ \_\ x_n\texttt{)}\ e'''$. By Method Type, we have $\sigma; \emptyset \vdash e'' : \mu'\, C'$ and so $\mu'\, C \leq T'$. Thus, since $\mu' \leq \mu'$, by the TAs typing rule, we have $\sigma; \emptyset \vdash e''\ \texttt{as}\, \mu' : \mu'\, C$. Finally, by the TSub typing rule, we have $\sigma; \emptyset \vdash e''\ \texttt{as}\, \mu' : T'$.

  If $e' = e''\ \texttt{as}\, \mu'$ then we are done. Otherwise, $e' = \texttt{M(}l_0\texttt{;}\ e''\ \texttt{as}\, \mu'\texttt{;}\ \texttt{(read}\, l_0\texttt{).invariant())}$, and by Monitor Type, we have $\sigma; \emptyset \vdash e' : T'$ as required.

- Suppose the AS rule was applied, i.e. we have $e = \mu\, l\ \texttt{as}\, \mu'$, $\sigma' = \sigma$, and $e' = \mu'\, l$. By the TAs and TAsCapsule typing rules, we have some C with $\sigma; \emptyset \vdash \mu\, l : \_\ C$ (since $\widehat{\emptyset} = \emptyset$) and $T' = \mu'\, C$. Thus, by Ref Type, we have $\mathrm{C}^{\sigma}_{l} \leq C$, and $\sigma; \emptyset \vdash \mu'\, l : \mu'\, C$ as required.

- Suppose the TRY ENTER rule was applied, i.e. we have $e = \texttt{try}\ \{e_1\}\ \texttt{catch}\ \{e_2\}$, $\sigma' = \sigma$, and $e' = \texttt{try}^{\sigma}\{e_1\}\ \texttt{catch}\ \{e_2\}$. By the TTryCatch1 typing rule, we have $\sigma; \emptyset \vdash e_1 : T'$ and $\sigma; \emptyset \vdash e_2 : T'$, thus by the TTryCatch2 rule we have $\sigma; \emptyset \vdash \texttt{try}^{\sigma}\{e_1\}\ \texttt{catch}\ \{e_2\} : T'$, as required.

- Suppose the TRY OK rule was applied, i.e. we have $e = \texttt{try}^{\sigma''}\{v\}\ \texttt{catch}\ \{\_\}$, $\sigma' = \sigma$, and $e' = v$. By the TTryCatch2 typing we have $\sigma; \emptyset \vdash v : T'$, as required.

- Suppose the TRY ERROR rule was applied, i.e. we have $e = \texttt{try}^{\sigma''}\{e_1\}\ \texttt{catch}\ \{e_2\}$, $\sigma' = \sigma$, and $e' = e_2$, where $error(\sigma, e_1)$. By the TTryCatch2 typing we have $\sigma; \emptyset \vdash e_2 : T'$, as required.

- Otherwise, the MONITOR EXIT rule was applied, i.e. we have $e = \texttt{M}(l; v; \mu\, l')$, $\sigma' = \sigma$, and $e' = v$, where $\mathrm{C}^{\sigma}_{l'} = \texttt{True}$. By the TMonitor typing rule we have $\sigma; \emptyset \vdash v : T'$ as required.

In the inductive case for our inner induction, we have some e_0 and minimal $\mathcal{E}_v \neq \Box$, where $e = \mathcal{E}_v[e_0]$; thus e_0 is a direct subexpression of e. By the structure of our reduction rules we have $\sigma|e_0 \to \sigma'|e'_0$ and $e' = \mathcal{E}_v[e'_0]$. Clearly e' is not of form v, so the typing rule used to obtain $\sigma; \emptyset \vdash \mathcal{E}_v[e_0] :: T'$ must not have been TSub, TVar, or TRef.

Now we can use the same typing rule that gave us $\sigma; \emptyset \vdash \mathcal{E}_v[e_0] : T'$ to obtain $\sigma'; \emptyset \vdash \mathcal{E}_v[e'_0] : T'$, as required; this step is valid since:

- The typing rule will require a premise of form $\sigma'; \emptyset \vdash \mathcal{E}_v[e'_0] : T_0$, for some T_0. Since $\sigma; \emptyset \vdash \mathcal{E}_v[e_0] : T'$, we must also have $\sigma; \emptyset \vdash e_0 : T_0$, and since we have $\sigma|e_0 \to \sigma'|e'_0$, by the inductive hypothesis, we have $\vdash \sigma'$ and $\sigma'; \emptyset \vdash e'_0 : T_0$.

- The typing rule may require other premises, each of form $\sigma'; \emptyset \vdash e_1 : T_1$, where e_1 is a direct subexpression of $\mathcal{E}_v$. Since $\sigma; \emptyset \vdash \mathcal{E}_v[e_0] : T'$, we must also have $\sigma; \emptyset \vdash e_1 : T_1$. Regardless of the reduction rule applied to get $\sigma|e_0 \to \sigma'|e'_0$, we have $\forall l \in dom(\sigma)$, $\mathrm{C}^{\sigma}_{l} = \mathrm{C}^{\sigma'}_{l}$, and so we also have $\sigma'; \emptyset \vdash e_1 : T_1$ (since the only typing rule that depends on the σ is the TRef rule, but since we have not altered the value of any C^{σ}_{l}, such a rule is still valid under σ').

- The typing rule may require side-conditions to hold. But these are the same side-conditions that $\sigma; \emptyset \vdash \mathcal{E}_v[e_0] : T'$ has, since no side-conditions depend on the value of σ nor the values of any subexpressions. Note that the side-conditions may depend on the *types*, but as shown above, the direct subexpressions of $\mathcal{E}_v[e_0]$ have the same types as those of $\mathcal{E}_v[e_0]'$.

Finally, in the inductive case of our outer induction, we have $n = k + 1$ and $\sigma|e \to^k \sigma_k|e_k \to \sigma'|e'$. By the inductive hypothesis we have that $\vdash \sigma_k$ and $\sigma_k; \emptyset \vdash e_k : T$ and so by the base case for $n = 1$, we have $\vdash \sigma'$ and $\sigma'; \emptyset \vdash e' : T$, as required. $\square$

As a simple corollary, any subexpression obtained from reducing a valid initial memory and main expression is well-typed.

Corollary 2 (Valid Type)**.**
If $validState(\sigma, \mathcal{E}[e])$ then $\vdash \sigma$ and $\sigma; \emptyset \vdash e :: T$, for some T.

Proof. By definition of $validState$, we have some e_0 and T_0 with $\sigma_0; \emptyset \vdash e_0 : T_0$, $\sigma_0 = c \mapsto \texttt{Cap}\{\}$ and $\sigma_0|e_0 \to^* \sigma|\mathcal{E}[e]$. Clearly $\vdash \sigma_0$ and $\texttt{Cap}$ is defined to be a class with no fields. Thus by Type Preservation we have $\sigma; \emptyset \vdash \mathcal{E}[e] : T_0$. Finally, by Nested Type and Type Rule we have $\sigma; \emptyset \vdash e :: T$, for some T'. $\square$

Now we present a simple lemma relating *immutable* with *MROG* and *mutatable*:

Lemma 11 (Immutable ROG)**.**
If not $immutable(\sigma, e, l)$ and $l \in ROG(\sigma, l')$, then:
1. $l \in MROG(\sigma, l')$, and
2. if $\texttt{mut}\ l' \in e$ or $\texttt{capsule}\ l' \in e$, then $mutatable(\sigma, e, l)$.

Proof.
1. l cannot be in the ROG of l' through any $\texttt{imm}$ fields (or else l would be *immutable*), and so it must be in $ROG(\sigma, l')$ only through $\texttt{mut}$ or $\texttt{rep}$ fields, and so it is in $MROG(\sigma, l')$
2. Follows immediately from the above and the definition of *mutatable*. $\square$

Finally, we show that reduction does not depend on reference capabilities: if we have an expression e_0, then any memory & expression that could result from reducing e_0 can also be obtained by reducing e'_0 (except that the resulting expression may differ in reference capabilities). Note that the resulting memory will be identical, as memory does not contain reference capabilities. This lemma is needed to reason over our $\sigma;\widehat{\Gamma} \vdash e : T$ judgements: any state obtained by reducing e after substituting in references according to Γ, will also be obtainable by reducing e after substituting according to $\widehat{\Gamma}$.

Lemma 12 (Bisimulation)**.**

If $e_0 \sim e'_0$ and $\sigma_0|e_0 \to^n \sigma|e$, then we have some e' where $\sigma_0|e'_0 \to^n \sigma_n|e'$ and $e \sim e'$.

Proof. The proof is by induction on n. In the first base case, we assume $n = 0$, and so we have $\sigma = \sigma_0$, $e = e_0$, and we can set $e' = e'_0$ so that $\sigma_0|e'_0 \to^0 \sigma|e'$ and $e \sim e'$ holds.

In the second base case, we assume $n = 1$. Let e_1 and $\mathcal{E}_v$ be such that $e_0 = \mathcal{E}_v[e_1]$ and $\mathcal{E}_v$ is maximal. By the structure of our reduction rules, we have that $e = \mathcal{E}_v[e_2]$, for some e_2. Since $\mathcal{E}_v[e_1] \sim e'_0$, there exists $\mathcal{E}'_v$ and e'_1 such that $e'_0 = \mathcal{E}'_v[e'_1]$ and $e_1 \sim e'_1$. We now proceed by cases on the reduction rule applied, and construct an e'_2 with $\sigma|\mathcal{E}'_v[e'_1] \to \sigma|\mathcal{E}'_v[e'_2]$ and $e_2 \sim e'_2$:

- Suppose the ACCESS rule applied, i.e. we have $e_1 = \mu\, l.f$, $\sigma = \sigma_0$, and $e_2 = \mu{::}\kappa\, \sigma_0[l.f]$, where $\mathrm{C}^{\sigma}_{l}.f = \kappa \_ f$. Since $e_1 \sim e'_1$, we have $e'_1 = \mu'\, l.f$, for some μ'. Let $e'_2 = \mu'{::}\kappa\, \sigma_0[l.f]$, then clearly $e_2 \sim e'_2$. Since the value of κ does not depend on the value of μ, we can apply the ACCESS rule again to get $\sigma_0|\mathcal{E}'_v[\mu'\, l.f] \to \sigma|\mathcal{E}'_v[e_2]$, as required.

- Suppose the TRY ERROR rule applied, i.e. $e_1 = \mathtt{try}^{\sigma'}\{e_3\}\ \mathtt{catch}\ \{e_4\}$, $\sigma = \sigma_0$, and $e_2 = e_4$, where $\mathit{error}(\sigma, e_3)$. Since $e_1 \sim e'_1$, we have $e'_1 = \mathtt{try}^{\sigma'}\{e'_3\}\ \mathtt{catch}\ \{e'_4\}$, with $e_3 \sim e'_3$ and $e_4 \sim e'_4$. Let $e'_2 = e'_4$, by the above we have $e_2 \sim e'_2$. As the definition of *error* does not depend on μs, we have $\mathit{error}(\sigma, e'_3)$. Thus we can apply the TRY ENTER rule again, yielding $\sigma_0|\mathcal{E}'_v[e'_1] \to \sigma|\mathcal{E}'_v[e'_2]$, as required.

- Suppose the MONITOR EXIT rule applied, i.e. $e_1 = \mathtt{M}(l;\, v;\, \mu\, l')$, $\sigma = \sigma_0$, and $e_2 = v$, where $\mathrm{C}^{\sigma_0}_{l'} = \mathtt{True}$. As this rule doesn't depend on the value of μ, this is similar to the TRY ERROR case above, except that we have $e'_1 = \mathtt{M}(l;\, v';\, \mu'\, l')$, with $v \sim v'$, and we set $e'_2 = v'$.

- Suppose the TRY ENTER rule applied, i.e. $e_1 = \mathtt{try}\ \{e_3\}\ \mathtt{catch}\ \{e_4\}$, $\sigma = \sigma_0$, and $e_2 = \mathtt{try}^{\sigma_0}\{e_3\}\ \mathtt{catch}\ \{e_4\}$. This is similar to the TRY ERROR case above, except that we have $e'_2 = \mathtt{try}\ \{e'_3\}\ \mathtt{catch}\ \{e'_4\}$, with $e_3 \sim e'_3$ and $e_4 \sim e'_4$, and we set $e'_2 = \mathtt{try}^{\sigma_0}\{e'_3\}\ \mathtt{catch}\ \{e'_4\}$.

- Suppose the TRY OK rule applied, i.e. $e_1 = \mathtt{try}^{\sigma'}\{v\}\ \mathtt{catch}\ \{\_\}$, $\sigma = \sigma_0$, and $e_2 = v'$. This is similar to the TRY ERROR case above, except that we have $e'_2 = \mathtt{try}^{\sigma'}\{v'\}\ \mathtt{catch}\ \{\_\}$, with $v \sim v'$, and we set $e'_2 = v'$.

- Otherwise the AS, NEW TRUE, UPDATE, CALL, or CALL MUTATOR rule applied. Let $e'_2 = e_2$, we thus trivially have $e'_2 \sim e_2$. As these reduction rules do not depend on the capabilities of references[38] in e_1 or $\mathcal{E}_v$, either in their side-conditions, or their right-hand-sides, $\sigma_0|\mathcal{E}'_v[e'_1] \to \sigma|\mathcal{E}'_v[e'_2]$ is also a valid reduction, as required.

As $\mathcal{E}_v[e_1] \sim \mathcal{E}'_v[e'_1]$, it follows from the above that $\mathcal{E}_v[e_2] \sim \mathcal{E}'_v[e'_2]$, so set $e' = \mathcal{E}'_v[e'_2]$, and then we have $\sigma_0|e'_0 \to \sigma|e'$ and $e \sim e'$, as required.

In the inductive case, we have $n = k + 1$ and $\sigma_0|e_0 \to^k \sigma_k|e_k \to \sigma|e$. By the inductive hypothesis, we have some e'_k such that $\sigma_0|e'_0 \to^k \sigma_k|e'_k$ and $e_k \sim e'_k$, so by the base case for $n = 1$, we have some e' with $\sigma_k|e'_k \to \sigma|e'$ and $e' \sim e$, thus we have $\sigma_0|e'_0 \to^{k+1} \sigma|e'$ as required. ☐

Conventional Soundness

For the purposes of our invariant protocol and the requirements in Appendix A, we do not require that well-typed programs do not get stuck during reduction, e.g. because a non-existent method is called. However, to show that our system is practical, we prove the key property below: every well-typed expression can either continue to be reduced, it is a value, or it contains an uncaught exception (i.e. an invariant failure). Thus showing that our type system satisfies the conventional soundness notion of Progress + Type Preservation.

[38] Note that the AS rule does depend on the μ' in "$\mu\, l\, \mathtt{as}\, \mu'$", but that μ' is not attached to a *reference*.

Theorem 5 (Progress)**.**

If $\vdash \sigma$ and $\sigma; \emptyset \vdash e : T$ then either:

- e is of form v,
- $error(\sigma, e)$, or
- $\exists e', \sigma'$ with $\sigma|e \to \sigma'|e'$.

Proof. The proof is by induction on the size of e: we assume the theorem holds for all subexpressions (if any) of e, and show that it holds for the entire e.

Suppose that there is no e' or σ' with $\sigma|e \to \sigma'|e'$, then this means that none of the reduction rules applied. Note that by Type Rule we have some T' with $\sigma; \emptyset \vdash e :: T'$.

Suppose that reduction is stuck because there is no rule whose left-hand-side matches $\sigma|e$. From the grammar for $\mathcal{E}_v$ and e, the only way this could occur is if e is of form x. But there is no way to obtain $\sigma; \emptyset \vdash x :: T'$, because the TVar rule would set $T' = \emptyset(x)$, which is undefined.

Thus, there are matching reduction rules, but none of their side-conditions/right-hand-sides are satisfiable. Consider each such rule:

- Suppose the new rule matches, and so $e = \mathcal{E}_v[\texttt{new}\ C\texttt{(}\_\ l_1, \ldots, \_\ l_n\texttt{)}]$. $fresh(\sigma)$ is well-defined since there is always some $l_0 \notin dom(\sigma)$. Thus, we must have $C = \texttt{True}$. By definition, the $\texttt{True}$ class contains no fields, thus by our TNew rule, we have $n = 0$, and so the new true rule applies, whose side-condition is satisfiable, a contradiction..

- Suppose the new true rule applies, then as with new above, the side condition is satisfiable, a contradiction.

- Suppose the access rule matches, and so $e = \mathcal{E}_v[\mu\, l\texttt{.}f]$. By our TAccess typing rule we require that $\sigma; \emptyset \vdash \mu\, l : \_\ C$, for some C, and $C.f$ is defined. By Type Ref we have that $\mathrm{C}^\sigma_l \leq C$. Thus $l \in dom(\sigma)$, moreover, since $\vdash \sigma$, it follows that C^σ_l is a class (and not an interface). Thus by our well-formedness rules on the class table, we have $\mathrm{C}^\sigma_l = C$. By $\vdash \sigma$, since $C.f$ exists, it follows that $\sigma[l.f]$ is defined. Thus every part of the side-condition of access is well defined, a contradiction.

- Suppose the update rule matches, and so $e = \mathcal{E}_v[\_\ l\texttt{.}f \texttt{ = } \_\, l']$. By our TUpdate typing rule, we have $\sigma; \emptyset \vdash \mu\, l : \_\ C$, for some C, where $C.f$ is defined. By the above case for TAccess, we thus have that $\mathrm{C}^\sigma_l = C$ and $\sigma[l.f]$ is defined. Thus $\sigma[l.f = l']$ is also well-defined, and so the right-hand-side of the update rule is satisfiable, a contradiction.

- Suppose the call rule matches, and so $e = \mathcal{E}_v[\_\ l_0\texttt{.}m\texttt{(}\_\ l_1, \ldots, \_\ l_n\texttt{)}]$. By Method Type, we have that $\mathrm{C}^\sigma_{l_0}.m = \mu_0\, \texttt{method}\, \mu'\ \_\ m\texttt{(}\mu_1\ \_\ x_1, \ldots, \mu_n\ \_\ x_n\texttt{)}\, e'$ is well-defined. Thus we must have that $\mu_0 = \texttt{mut}$ and $e' = \mathcal{E}[\texttt{this.}f]$ with $\mathrm{C}^\sigma_{l_0}.f = \texttt{rep}\ \_\ f$, which satisfies the side-conditions of the call mutator rule, a contradiction.

- Suppose the call mutator rule matches, and so $e = \mathcal{E}_v[\_\ l_0\texttt{.}m\texttt{(}\_\ l_1, \ldots, \_\ l_n\texttt{)}]$. As above, by Method Type, we have that $\mathrm{C}^\sigma_{l_0}.m = \mu_0\, \texttt{method}\, \mu'\ \_\ m\texttt{(}\mu_1\ \_\ x_1, \ldots, \mu_n\ \_\ x_n\texttt{)}\, e'$. Thus the only way the side conditions are unsatisfiable is if $\mu_0 \neq \texttt{mut}$, e' is not of form $\mathcal{E}[\texttt{this.}f]$, or $\mathrm{C}^\sigma_s l_0.f$ is not of form $\texttt{rep}\ \_\ f$, but then the side-conditions for the call rule are satisfiable, a contradiction.

- Suppose the try error rule matches, then $e = \mathcal{E}_v[\texttt{try}^{\sigma'}\texttt{\{}e'\texttt{\}}\ \texttt{catch}\ \texttt{\{}e''\texttt{\}}]$. Thus we have that its side-condition, $error(\sigma, e')$, does not hold. If e' is of form v, then the try ok rule applies. Thus by the inductive hypothesis, we must have some σ' and e''' such that $\sigma|e' \to \sigma'|e'''$. And so $e = \mathcal{E}'_v[e']$, where $\mathcal{E}'_v = \mathcal{E}_v[\texttt{try}^{\sigma'}\texttt{\{}\square\texttt{\}}\ \texttt{catch}\ \texttt{\{}e''\texttt{\}}]$. Thus we can use the same rule that got us $\sigma|e' \to \sigma'|e'''$ to instead give us $\sigma|\mathcal{E}'_v[e'] \to \sigma'|\mathcal{E}'_v[e''']$, a contradiction. Note that this works because the reduction rules never look at the actual value of the $\mathcal{E}_v$.

- Suppose the monitor exit rule matches, then $e = \mathcal{E}_v[e']$ with $e' = \texttt{M(}l\texttt{;}\ v\texttt{;}\ \mu\, l'\texttt{)}$. Thus we have that $\mathrm{C}^\sigma_{l'} \neq \texttt{True}$. Thus $error(\sigma, e')$. If $\mathcal{E}_v$ is of form $\mathcal{E}'_v[\texttt{try}^{\sigma'}\texttt{\{}\mathcal{E}''_v\texttt{\}}\ \texttt{catch}\ \texttt{\{}\_\texttt{\}}]$, where $\mathcal{E}'_v$ is maximal, then the try error rule applies. Thus, as e' is of form $\texttt{M(}l\texttt{;}\ v\texttt{;}\ \mu\, l'\texttt{)}$ and $\mathrm{C}^\sigma_{l'} \neq \texttt{True}$, we have that $error(\sigma, \mathcal{E}_v[e'])$ holds.

- Suppose the AS, TRY ENTER, or TRY OK rules match, these rules have no side-conditions, and the right-hand-sides are trivially satisfiable, a contradiction.

Thus from the above, we must have had that only the MONITOR EXIT rule matched, and $error(\sigma, e)$ holds. $\square$

Proof of Type System Requirements

Finally we prove each of the requirements from Appendix A.

Requirement 1 (Type Consistency)**.**

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

Proof.

1. Follows immediately from Valid Type and our TNEW typing rule.
2. Follows immediately from Valid Type and our TUPDATE typing rule.
3. Follows immediately from Valid Type and Method Type. $\square$

Requirement 5 (Mut Update)**.**

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

Proof. Follows immediately from Valid Type and our TUPDATE rule. $\square$

Now we prove a slightly stronger version of the Mut Consistency requirement, which works for any well-formed memory and well-typed expression, even if they are not a *validState* (i.e. they are not obtainable by reducing a valid initial memory & expression). We will use this stronger property in combination with Bisimulation to reason over expressions typed under a $\widehat{\Gamma}$.

Lemma 13 (Stronger Mut Consistency)**.**

If $\vdash \sigma$, $\sigma; \emptyset \vdash e : T$, $l \in dom(\sigma)$, not $mutatable(\sigma, e, l)$, and $\sigma|e \rightarrow^n \sigma'|e'$, then not $mutatable(\sigma', e', l)$.

Proof. The proof is by induction on n. In the first base case, we assume that $n = 0$, and our lemma trivially holds since $\sigma' = \sigma$ and $e' = e$.

In the second base case, we assume that $n = 1$. We now assume that $mutatable(\sigma', e', l)$, and then proceed by cases on the reduction rule applied and show a contradiction, thus proving that l must not be *mutatable*:

- Suppose the UPDATE rule was applied, i.e. we have some $\mathcal{E}_v$ with $e = \mathcal{E}_v[\mu\, l'\texttt{.}f\texttt{=}\mu'\, l'']$, $\sigma' = \sigma[l'.f = l'']$, and $e' = \mathcal{E}_v[\texttt{M(}l'\texttt{;}\ \texttt{mut}\ l'\texttt{;}\ \texttt{(read}\ l'\texttt{).invariant())}]$. By Type Preservation, Type Rule, and our TUPDATE typing rule, we have $\mu \leq \texttt{mut}$. Since $l' \in MROG(\sigma, l')$, and l was not *mutatable*, we have that $l' \notin ROG(\sigma, l)$, and so we have not mutated the ROG of l, i.e. $ROG(\sigma, l) = ROG(\sigma', l)$. Thus the only way for l to have become *mutatable* is if we have some $l_1 \in ROG(\sigma', l)$ and some l_2 with $\texttt{mut}\ l_2 \in e'$ or $\texttt{capsule}\ l_2 \in e'$, and $l_1 \in MROG(\sigma', l_2)$. Since $\sigma' = \sigma[l'.f = v]$ and l was not previously *mutatable*, we must have caused l_1 to be in $MROG(\sigma', l_2)$ through the fact that $\sigma'(l'.f) = l''$, and so we have that $C^{\sigma'}_{l'}.f = \kappa\, C\, f$ for some $\kappa \in \{\texttt{mut}, \texttt{rep}\}$. Thus before the reduction, we had $l_1 \in MROG(\sigma, l'')$ and $l' \in MROG(\sigma, l_2)$. By Type Preservation, Type Rule, and our TUPDATE typing rule, we have that $\mu' \in \{\texttt{mut}, \texttt{capsule}\}$. Since $l_1 \in MROG(\sigma, l'')$ and $l_1 \in ROG(\sigma, l)$, we thus have $mutatable(\sigma, e, l)$, a contradiction.

- Suppose the ACCESS rule was applied, i.e. we have some $\mathcal{E}_v$ with $e = \mathcal{E}_v[\mu\, l'.f]$, $\sigma' = \sigma$, and $e' = \mathcal{E}_v[v]$, where $v = \mu{::}\kappa\, \sigma[l'.f]$ and $\mathrm{C}^{\sigma}_{l'}.f = \kappa\, C\, f$. As we have not modified memory, the only way for l to have become *mutatable* is via v, i.e. we must have $\mu{::}\kappa \leq \texttt{mut}$ and some $l'' \in ROG(\sigma, l)$ such that $l'' \in MROG(\sigma, \sigma[l'.f])$. By definition of $\mu{::}\kappa$ this implies that $\kappa \in \{\texttt{mut}, \texttt{rep}\}$ and $\mu \leq \texttt{mut}$. So we have that $l'' \in MROG(\sigma, l')$, and $\texttt{mut}\, l' \in e$ or $\texttt{capsule}\, l' \in e$. Thus we must have $mutatable(\sigma, e, l)$, a contradiction.

- Suppose the NEW/NEW TRUE rule was applied, i.e. we have some $\mathcal{E}_v$ with $e = \mathcal{E}_v[\texttt{new}\, C\texttt{(}\mu_1\, l_1, .., \mu_n\, l_n\texttt{)}]$, $\sigma' = \sigma, l' \mapsto C\{l_1, .., l_n\}$, and $e' \in \{\mathcal{E}_v[\texttt{M(}l'\texttt{;}\, \texttt{mut}\, l'\texttt{;}\, \texttt{(read}\, l'\texttt{).invariant())}], \mathcal{E}_v[\texttt{mut}\, l']\}$. Since no preexisting part of σ is modified, we must have that l is now *mutatable* through the $\texttt{mut}\, l'$ reference in e', i.e. we must have some $l'' \in ROG(\sigma, l)$ with $l'' \in MROG(\sigma', l')$. By No Dangling we have $l'' \neq l'$, thus we have that $i \in [1, n]$, $C.i = \kappa\, C'\, f$, $\kappa \in \{\texttt{mut}, \texttt{rep}\}$, and $l'' \in MROG(\sigma, l_i)$. By Type Preservation, Type Rule, and our TNEW typing rule, we have that $\mu_i \leq \texttt{mut}$. Since $l'' \in MROG(\sigma, l_i)$ and $l'' \in ROG(\sigma, l)$, we thus have $mutatable(\sigma, e, l)$, a contradiction.

- Suppose the AS rule was applied, i.e. we have some $\mathcal{E}_v$ with $e = \mathcal{E}_v[\mu\, l'\, \texttt{as}\, \mu']$, $\sigma' = \sigma$, and $e' = \mathcal{E}_v[\mu'\, l']$ By Type Preservation and Type Rule either the TAS or TASCAPSULE typing rule applied. In either case, by Ref Type we have that $\mu' \leq \texttt{mut}$ only if $\mu \leq \texttt{mut}$. As we haven't introduced any other reference or modified any memory, we must have that l is now *mutatable* through $\mu'\, l'$. But them $\mu' \leq \texttt{mut}$ and so $\mu \leq \texttt{mut}$, and hence l was already *mutatable* through $\mu\, l$, a contradiction.

- Suppose that the CALL/CALL MUTATOR rule was applied, i.e. we have some $\mathcal{E}_v$ with $e = \mathcal{E}_v[\mu_0\, l_0\texttt{.}m\texttt{(}\mu_1\, l_1, .., \mu_n\, l_n\texttt{)}]$, $\sigma' = \sigma$, and $e' \in \{e''\, \texttt{as}\, \mu'', \texttt{M(}l_0\texttt{;}\, e''\, \texttt{as}\, \mu''\texttt{;}\, \texttt{(read}\, l_0\texttt{).invariant()}\}$, $e'' = e'''[\texttt{this} := \mu'_0\, l_0, x_1 := \mu'_1\, l_1, .., x_n := \mu'_n\, l_n]$, and $\mathrm{C}^{\sigma}_{l_0} = \mu'_0\, \texttt{method}\, T\, m\texttt{(}\mu'_1\, \_\, x_1, .., \mu'_n\, \_\, x_n\texttt{)}\, e'''$. As we haven't modified memory, for this reduction to have made l *mutatable*, we must have introduced a $\texttt{mut}$ or $\texttt{capsule}$ reference in e''. By our well-formedness rules on method bodies, there are no references in e''', thus l must be *mutatable* through one of the $\mu'_i\, l_i$ references we substituted into e''', for some $i \in [1, n]$, where $\mu'_i \leq \texttt{mut}$. By Type Preservation and Method Type, we have that $\mu_i \leq \mu'_i$, and so $\mu_i \leq \texttt{mut}$ and hence e already had a reference, $\mu_i\, l_i$, through which l was *mutatable*, a contradiction.

- Otherwise, the TRY ENTER, MONITOR EXIT, TRY OK, or TRY ERROR rule was applied. However, memory was not modified, and no new references where added to the main expression, thus we can't have caused *mutatable* to now hold, a contradiction.

In the inductive case, we have $n = k+1$ and $\sigma|e \to^k \sigma_k|e_k \to \sigma'|e'$. By the inductive hypothesis we have not $mutatable(\sigma_k, e_k, l)$. We clearly have $l \in dom(\sigma_k)$, as no reduction rule removes from memory, thus by the base case for $n = 1$, we have not $mutatable(\sigma', e', l)$, as required. □

Similar to Stronger Mut Consistency, we prove a stronger version of Non-Mutating.

Corollary 3 (Stronger Non-Mutating)**.**
If $\vdash \sigma$, $\sigma; \emptyset \vdash e : T$, $l \in dom(\sigma)$, not $mutatable(\sigma, e, l)$, and $\sigma|e \to^* \sigma'|e'$, then $\sigma'(l) = \sigma(l)$

Proof. The proof is the same as for Non Mutating in Appendix A, except we use Stronger Mut Consistency instead of Mut Consistency and use Type Preservation, Type Rule, and the TUPDATE rule instead of Mut Update. □

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

Proof. By Valid Type we have $\vdash \sigma$ and $\sigma; \emptyset \vdash e : T$ for some type T, and so the conclusion holds by Stronger Mut Consistency. □

Now the hardest requirements to prove: Imm Consistency and Capsule Consistency. We need to prove these simultaneously as a $\texttt{capsule}$ can be used where an $\texttt{imm}$ is expected, and our TASCAPSULE typing rule allows the use of $\texttt{imm}$ local variables.

Theorem 6 (Imm–Capsule Consistency).

If $validState(\sigma, e)$, then $\forall l$:
1. if $immutable(\sigma, e, l)$, then not $mutatable(\sigma, e, l)$, and
2. if $e = \mathcal{E}[\texttt{capsule}\, l]$, then $encapsulated(\sigma, \mathcal{E}, l)$.

Proof. We prove this by definition of *validState* and induction on the number of reductions since the initial main expression and memory. The base case is trivial since the main expression cannot contain any `imm` references, and there are no fields in memory, thus nothing can be *immutable*, moreover the main expression cannot contain any `capsule` references.

In the inductive case we assume that our theorem holds for all previous states, we then pick an arbitrary l and prove the two conclusions for the current $\sigma|e$.

1. First we show that Imm Consistency holds. If l was previously *immutable*, by the inductive hypothesis and Mut Consistency, l is still not *mutatable*, as required.

   Now suppose that l was not *immutable* in the previous state, but is now. We then proceed by cases on the reduction rule applied and show that l is now not *mutatable*:

   - (AS) $\sigma|\mathcal{E}_v[\mu\, l' \,\texttt{as}\, \mu'] \to \sigma|e$, where $e = \mathcal{E}_v[\mu'\, l']$. Since l was not *immutable* in $\mathcal{E}_v$ and we haven't modified memory, the only way it could now be *immutable* is if $\mu' = \texttt{imm}$ and $l \in ROG(\sigma, l')$. By Valid Type, we must have that $\mu\, l \,\texttt{as}\, \mu'$ was well-typed by TAs (and not TAsCapsule, as $\mu' \neq \texttt{capsule}$), thus $\mu \leq \texttt{imm}$. Clearly $\mu \neq \texttt{imm}$, since l was not *immutable*. Thus by definition of $\leq$, we have that $\mu = \texttt{capsule}$. Since $l \in ROG(\sigma, l')$, and l was not *immutable*, by Immutable ROG, we have $mutatable(\sigma, \mathcal{E}_v[\texttt{capsule}\, l' \,\texttt{as}\, \mu'], l)$. By the inductive hypothesis, we have $encapsulated(\sigma, \mathcal{E}_v[\square \,\texttt{as}\, \mu'], l')$, and so it follows that not $reachable(\sigma, \mathcal{E}_v, l)$. Thus, we have l is not *reachable* in $\mathcal{E}_v[\mu'\, l']$ except through $\mu'\, l'$, but $\mu' = \texttt{imm}$, so it follows that l is not *mutatable* in $\mathcal{E}_v[\mu'\, l']$.
   - (NEW/NEW TRUE) $\sigma'|\mathcal{E}_v[\texttt{new}\ C\texttt{(}\mu_1\, l_1, \ldots, \mu_n\, l_n\texttt{)}] \to \sigma|e$, where $\sigma = \sigma', l_0 \mapsto C\{l_1, \ldots, l_n\}$, $e = \mathcal{E}_v[e']$, and $e' \in \{\texttt{M(}l_0\texttt{;}\ \texttt{mut}\, l_0\texttt{;}\ \texttt{(read}\, l_0\texttt{).invariant())}, \texttt{mut}\, l_0\}$. By Valid Type, $\texttt{new}\ C\texttt{(}\mu_1\, l_1, \ldots, \mu_n\, l_n\texttt{)}$ was typed by TNew and so we have $\texttt{class}\ C\ \texttt{implements}\ \_\ \texttt{\{}Fs\texttt{;}\_\texttt{\}}$, where $Fs = \kappa_1\ \_\ f_1, \ldots, \kappa_n\ \_\ f_n$. Since l was not *immutable* in σ' through $\mathcal{E}_v$, and existing objects in σ' have not been modified, it follows that l must be *immutable* through e'. As the only object mentioned in e' is l_0, we have $l \in ROG(\sigma, l_0)$. As we haven't modified preexisting objects and $\texttt{imm}\, l_0 \notin e'$, it follows that we have some $i \in [1, n]$ with $\kappa_i = \texttt{imm}$ and $l \in ROG(\sigma, \sigma[l_0.f_i]) = ROG(\sigma, l_i)$. By Valid Type and the TNew typing rule, we have $\mu_i \leq \widetilde{\kappa_i} = \texttt{imm}$. Thus, as with the AS case above, we have $\mu_i = \texttt{capsule}$, and by Immutable ROG, we have that l was *mutatable*. Thus, by the inductive hypothesis, we have that l was previously *reachable* only through the $\mu_i\, l_i$ argument of the `new`. Thus l is not *reachable* through any $l_0.f_j$ with $j \neq i$, and so it follows that l is *reachable* in $\sigma|\mathcal{E}_v[e']$ only through $l_0.f_i$; as f_i is an `imm` field, it follows that l is not *mutatable*.
   - (ACCESS) $\sigma|\mathcal{E}_v[\mu\, l'.f] \to \sigma|e$, where $e = \mathcal{E}_v[\mu{::}\kappa\, \sigma[l'.f]]$ and $\mathrm{C}^{\sigma}_{l'}.f = \kappa\ \_\ f$. As we have not modified memory, it follows that l is *immutable* through the newly introduced reference to $\sigma[l'.f]$. As l was not previously *immutable* and the main expression already contained $\mu\, l'$, it follows that l is not in the ROG of any `imm` fields that are *reachable* through l'. Thus the only way l is now *immutable* is if we just introduced an `imm` reference to it, i.e. if $l = \sigma[l'.f]$ and $\mu{::}\kappa = \texttt{imm}$. By definition of $\mu{::}\kappa$, we have that either $\mu = \texttt{imm}$ or $\kappa = \texttt{imm}$. In the former case, $\texttt{imm}\, l$ would be in the main expression, in the latter case, l would be *reachable* through an `imm` field of $\mu\, l$; either way l must have been *immutable*, a contradiction.
   - (UPDATE) $\sigma'|\mathcal{E}_v[\mu\, l'.f = \mu'\, l''] \to \sigma|e$, where $\sigma = \sigma'[l'.f = l'']$ and $e = \mathcal{E}_v[\texttt{M(}l'\texttt{;}\ \texttt{mut}\, l'\texttt{;}\ \texttt{(read}\, l'\texttt{).invariant())}]$. As we haven't introduced any `imm` references to the main expression, any only $\sigma'[l'.f]$ was modified, it follows that for l to now be *immutable* we must have $C.f = \texttt{imm}\ \_\ f$ and $l \in ROG(\sigma, l'')$.

     Suppose $l \notin ROG(\sigma', l'')$. The only difference between σ and σ' is at $l'.f$. And so l must have been added to the ROG of l'' through the new value of $l'.f$, i.e. $\sigma[l'.f]$. But as no other part of σ' was modified, we must have $l \in ROG(\sigma', \sigma[l'.f])$, but $\sigma[l.f] = l''$, a contradiction.

Thus $l \in ROG(\sigma, l'')$. So by the AS case above, we have $\mu' = \texttt{capsule}$, and by Immutable ROG, we have that l was *mutatable*. Thus by the inductive hypothesis, we have that l was previously *reachable* only through $\mu' l''$. Thus l is now *reachable* only through $\sigma[l'.f]$, which is an $\texttt{imm}$ field, and so l is not *mutatable*.

- (CALL/CALL MUTATOR) $\sigma|\mathcal{E}_v[\mu_0\, l_0\texttt{.}m\texttt{(}\mu_1\, l_1, .., \mu_n\, l_n\texttt{)}] \rightarrow \sigma|e$, where $e = \mathcal{E}_v[e']$, $e' \in \{e''\, \texttt{as}\, \mu''$, $\texttt{M(}l_0\texttt{;}\, e''\, \texttt{as}\, \mu''\texttt{;}\, \texttt{(read}\, l_0\texttt{).invariant())}\}$, $e'' = e'''[\texttt{this} := \mu'_0\, l_0, x_1 := \mu'_1\, l_1, .., x_n := \mu'_n\, l_n]$, and $\mathrm{C}^{\sigma}_{l_0} = \mu'_0\, \texttt{method}\, \mu''\, \_\, m\texttt{(}\mu'_1\, \_\, x_1, .., \mu'_n\, \_\, x_n\texttt{)}\, e'''$. By our well-formedness rules on method bodies, there are no locations in e''', thus the only references in e'' are $\mu'_0\, l_0, .., \mu'_n\, l_n$. By definition of *immutable*, and since we have not modified memory, it follows that for some $i \in [1, n]$, $l \in ROG(\sigma, l_i)$ and $\mu'_i = \texttt{imm}$. As with the AS case above, by Valid Type and the TCall typing rule, we have that $\mu_i = \texttt{capsule}$, moreover, by Immutable ROG, $l \in MROG(\sigma, l_i)$. By the inductive hypothesis we have that l_i was *encapsulated* and so it follows that l is not *reachable* from $\mathcal{E}_v$, or through any l_j with $j \neq i$. As the only occurrences of l_i in e'' have reference capability $\mu'_i = \texttt{imm}$, we have that l is not *mutatable* in e''. The only reference to l_i that could be in e' but not in e'' has reference capability $\texttt{read}$, and so l is not *mutatable* in e' either. Finally, since l is not *reachable* in $\mathcal{E}_v$, it follows that l is not *mutatable* in $\mathcal{E}_v[e']$.
- (TRY ENTER/TRY OK/TRY ERROR/MONITOR EXIT) $\sigma|e' \rightarrow \sigma|e$. These rules do not modify memory, nor introduce or change references in the main expression, except perhaps by removing them, i.e. for any $v \in e$, we have $v \in e'$. Thus there is no way we could have made l *immutable*, a contradiction.

2. Now we show that Capsule Consistency holds, by assuming it does not, and then showing a contradiction. Thus we suppose that $e = \mathcal{E}[\texttt{capsule}\, l]$, for some $\mathcal{E}$ where *encapsulated*$(\sigma, \mathcal{E}, l)$ doesn't hold.

   Thus we pick an $l' \in ROG(\sigma, l)$ with *mutatable*$(\sigma, \mathcal{E}[\texttt{capsule}\, l], l')$ and *reachable*$(\sigma, \mathcal{E}, l')$. We now proceed by cases on the reduction rule we just applied, and show a contradiction, thus proving that l must in fact be *encapsulated*:

   - (NEW/NEW TRUE) $\sigma'|\mathcal{E}_v[e''] \rightarrow \sigma|\mathcal{E}[\texttt{capsule}\, l]$, where $\sigma = \sigma', l_0 \mapsto C\{ls\}$, $\mathcal{E}[\texttt{capsule}\, l] = \mathcal{E}_v[e']$, $e' \in \{\texttt{M(}l_0\texttt{;}\, \texttt{mut}\, l_0\texttt{;}\, \texttt{(read}\, l_0\texttt{).invariant())}, \texttt{mut}\, l_0\}$, and $e'' = \texttt{new}\, C\texttt{(}vs\texttt{)}$.
     - Suppose $\mathcal{E}$ is of form $\mathcal{E}_v[\mathcal{E}']$, i.e. the hole in $\mathcal{E}$ is within e'. But there are no $\texttt{capsule}$s in e', a contradiction.
     - Otherwise, $\mathcal{E}$ is not of form $\mathcal{E}_v[\mathcal{E}']$, i.e. the hole in $\mathcal{E}$ is within $\mathcal{E}_v$, and so $\texttt{capsule}\, l \in \mathcal{E}_v$ and $e' \in \mathcal{E}$. As we didn't modify $\mathcal{E}_v$, this $\texttt{capsule}\, l$ must have been in the previous state, i.e. we have some $\mathcal{E}'$ with $\mathcal{E}_v[e''] = \mathcal{E}'[\texttt{capsule}\, l]$ and $e'' \in \mathcal{E}'$ (since the hole in $\mathcal{E}$ is not within the hole in $\mathcal{E}_v$). By No Dangling, $l \in dom(\sigma')$, and since we didn't modify any preexisting objects, we have $ROG(\sigma, l) = ROG(\sigma', l)$. By the inductive hypothesis, we have *encapsulated*$(\sigma', \mathcal{E}', l)$, and by Mut Consistency, we have *mutatable*$(\sigma', \mathcal{E}'[\texttt{capsule}\, l], l')$, and since $l' \in ROG(\sigma, l)$, it follows that not *reachable*$(\sigma', \mathcal{E}', l')$.

       Suppose l' is *reachable* through the part of $\mathcal{E}_v$ that overlaps with $\mathcal{E}$, then there is some $l'' \in \mathcal{E}_v$ with $l' \in ROG(\sigma, l'')$. By No Dangling, $l'' \in dom(\sigma')$, and since preexisting memory wasn't modified, it follows that $l' \in ROG(\sigma', l'')$; since l'' is in the part of $\mathcal{E}_v$ that overlaps with $\mathcal{E}$, which is identical to the part of $\mathcal{E}_v$ that overlaps with $\mathcal{E}'$, we have $l'' \in \mathcal{E}'$, and so we have *reachable*$(\sigma', \mathcal{E}', l')$, a contradiction

       Otherwise, l' is *reachable* through e', clearly $l' \in dom(\sigma')$, and so by Lost Forever, we have *reachable*$(\sigma', \texttt{new}\, C\texttt{(}vs\texttt{)}, l')$. But $\texttt{new}\, C\texttt{(}vs\texttt{)} \in \mathcal{E}'$, and so we also have *reachable*$(\sigma, \mathcal{E}', l')$, which is still a contradiction.

       Note that the above steps do not depend on the actual forms of e' and e'', nor the reduction rule applied, they only require *validState*$(\mathcal{E}_v[e''])$, $\sigma'|e'' \rightarrow \sigma|e'$, $ROG(\sigma, l) = ROG(\sigma', l)$, and $\mathcal{E}_v[e'] = \mathcal{E}[\texttt{capsule}\, l]$, where $\mathcal{E}$ is not of form $\mathcal{E}_v[\mathcal{E}'']$.
   - (ACCESS) $\sigma|\mathcal{E}_v[\mu\, l''\texttt{.}f] \rightarrow \sigma|\mathcal{E}[\texttt{capsule}\, l]$, where $\mathcal{E}[\texttt{capsule}\, l] = \mathcal{E}_v[\mu{::}\kappa\, \sigma[l''.f]]$.

– Suppose $\mathcal{E} = \mathcal{E}_v$, so $\texttt{capsule}\, l = \mu{::}\kappa\, \sigma[l''.f]$: By definition of $\mu{::}\kappa$, this means that $\mu = \texttt{capsule}$, and so by the inductive hypothesis, we have $\mathit{encapsulated}(\sigma, \mathcal{E}_v[\square.f], l'')$. Since $l' \in \mathit{ROG}(\sigma, l)$ and $l = \sigma[l''.f]$, it follows that $l' \in \mathit{ROG}(\sigma, l'')$. Since l' is *mutatable* in $\mathcal{E}_v[\texttt{capsule}\, l]$, by Mut Consistency, l' is also *mutatable* in $\mathcal{E}_v[\texttt{capsule}\, l''\texttt{.}f]$. Thus, since l'' was *encapsulated* and $l' \in \mathit{ROG}(\sigma, l'')$, it follows that l' is not *reachable* through $\mathcal{E}_v[\square\texttt{.}f]$. Clearly this means l' is not *reachable* through $\mathcal{E}_v$, a contradiction.

– Otherwise, $\texttt{capsule}\, l \in \mathcal{E}_v$, and so by the NEW/NEW TRUE case above, we have a contradiction.

- (UPDATE) $\sigma'|\mathcal{E}_v[\mu\, l''\texttt{.}f\texttt{=}\mu'\, l'''] \to \sigma|\mathcal{E}[\texttt{capsule}\, l]$, where $\sigma = \sigma'[l''.f = l''']$ and $\mathcal{E}[\texttt{capsule}\, l] = \mathcal{E}_v[\texttt{M}(l''\texttt{;}\ \texttt{mut}\, l''\texttt{;}\ (\texttt{read}\, l'')\texttt{.invariant())}]$. Clearly $\texttt{capsule}\, l \in \mathcal{E}_v$, since there are no $\texttt{capsule}$s in the monitor we just reduced to. As the reduction didn't modify $\mathcal{E}_v$, have $\mathcal{E}_v[\mu\, l''\texttt{.}f\texttt{=}\mu'\, l'''] = \mathcal{E}'[\texttt{capsule}\, l]$, for some $\mathcal{E}'$, with $\mu\, l''\texttt{.}f\texttt{=}\mu'\, l''' \in \mathcal{E}'$. By the inductive hypothesis, we have $\mathit{encapsulated}(\sigma', \mathcal{E}', l)$. By Valid Type and our TUPDATE typing rule, we have $\mu = \texttt{mut}$.

  Suppose $l'' \in \mathit{ROG}(\sigma', l)$, then since $\mu = \texttt{mut}$, we have $\mathit{mutatable}(\sigma', \mathcal{E}'[\texttt{capsule}\, l], l'')$, and so it follows from $\mathit{encapsulated}(\sigma', \mathcal{E}', l)$ that not $\mathit{reachable}(\sigma', \mathcal{E}', l)$. But $\mu\, l''\texttt{.}f\texttt{=}v \in \mathcal{E}'$, and so l'' is clearly *reachable* in $\mathcal{E}'$, a contradiction.

  Thus we must have $l'' \notin \mathit{ROG}(\sigma', l)$. As σ only differs from σ' at l'', and $l'' \notin \mathit{ROG}(\sigma', l)$, it follows that the ROG of l can't have changed, i.e. $\mathit{ROG}(\sigma, l) = \mathit{ROG}(\sigma', l)$. Thus, by the NEW/NEW TRUE case above, we have a contradiction.

- (CALL/CALL MUTATOR) $\sigma|\mathcal{E}_v[\mu_0\, l_0\texttt{.}m\texttt{(}\mu_1\, l_1, \ldots, \mu_n\, l_n\texttt{)}] \to \sigma|\mathcal{E}[\texttt{capsule}\, l]$, where $\mathcal{E}[\texttt{capsule}\, l] = \mathcal{E}_v[e']$, $e' \in \{e''\ \texttt{as}\ \mu'', \texttt{M}(l_0\texttt{;}\ e''\ \texttt{as}\ \mu''\texttt{;}\ (\texttt{read}\, l_0)\texttt{.invariant())}\}$, $e'' = e'''[\texttt{this} \coloneqq \mu'_0\, l_0, x_1 \coloneqq \mu'_1\, l_1, \ldots, x_n \coloneqq \mu'_n\, l_n]$, and $\mathrm{C}^{\sigma}_{l_0} = \mu'_0\ \texttt{method}\ \mu''\ \_\ m\texttt{(}\mu'_1\ \_\ x_1, \ldots, \mu'_n\ \_\ x_n\texttt{)}\ e'''$.

  – Suppose $\mathcal{E} = \mathcal{E}_v[\mathcal{E}'']$ for some $\mathcal{E}''$, thus $\mathcal{E}''[\texttt{capsule}\, l] = e'$. Clearly $\texttt{capsule}\, l \in e''$, and by our well-formedness rules on method bodies, $\texttt{capsule}\, l \notin e'''$. Thus we must have some $i \in [0, n]$ with $\mu'_i\, l_i = \texttt{capsule}\, l$. Moreover, if we let $x_0 = \texttt{this}$, then this means that $e''' = \mathcal{E}'''[x_i]$, for some $\mathcal{E}'''$. By Method Type, we have $\mu_i \leq \mu'_i$, and since $\mu'_i = \texttt{capsule}$, we also have $\mu_i = \texttt{capsule}$. If $i \geq 1$, let $\mathcal{E}'_v = \mu_0\, l_0\texttt{.}m\texttt{(}\mu_1\, l_1, \ldots, \mu_{i-1}\, l_{i-1}, \square, \mu_{i+1}\, l_{i+1}, \ldots, \mu_n\, l_n\texttt{)}$; if $i = 0$, let $\mathcal{E}'_v = \square\texttt{.}m\texttt{(}\mu_1\, l_1, \ldots, \mu_n\, l_n\texttt{)}$. Clearly we have $\sigma|\mathcal{E}_v[\mathcal{E}'_v[\texttt{capsule}\, l]] \to \sigma|\mathcal{E}[\texttt{capsule}\, l]$. Thus, by the inductive hypothesis we have $\mathit{encapsulated}(\sigma, \mathcal{E}_v[\mathcal{E}'_v], l)$. By Mut Consistency, we have that l' was *mutatable*, and since $l' \in \mathit{ROG}(\sigma, l)$, it follows that l' is not *reachable* through $\mathcal{E}_v$, or any $\mu_j\, l_j$ with $j \neq i$.

    If $i = 0$, since $\mu'_i = \texttt{capsule}$ and $i = 0$, the method was not a rep mutator, and so the CALL (and not CALL MUTATOR) rule must have applied, thus $e' = e''\ \texttt{as}\ \mu''$, and so l' is *reachable* only through e''.

    Otherwise, if $i \geq 1$, regardless of whether CALL or CALL MUTATOR was applied, as l' is not *reachable* through l_0, l' can only be *reachable* through e''.

    Thus by our well-formedness rules on method bodies, we must have that l' is only *reachable* through each occurrence of $x_i \in e'''$, which have all been substituted with $\mu'_i\, l_i$ (since there are no other references in e''', and l' is not *reachable* through any x_j that has been substituted for $\mu'_j\, l_j$). As our type system requires that each method body mentions a $\texttt{capsule}$ receiver or parameters at most once, it follows that $x_i \notin \mathcal{E}'''$. Since $\mathcal{E}' = \mathcal{E}'''[x_0 \coloneqq \mu'_0\, l_0, \ldots, x_n \coloneqq \mu'_n\, l_n]\ \texttt{as}\ \mu''$, it follows that l' is not *reachable* through $\mathcal{E}'$. Thus l' was not *reachable* through $\mathcal{E}_v$ either, and so it follows that l' is not *reachable* through $\mathcal{E}$, a contradiction.

  – Otherwise, $\texttt{capsule}\, l \in \mathcal{E}_v$, and so by the NEW/NEW TRUE case above, we have a contradiction.

- (AS) $\sigma|\mathcal{E}_v[\mu\, l''\ \texttt{as}\ \mu'] \to \sigma|e$, where $e = \mathcal{E}_v[\mu'\, l'']$.

  – Suppose $\mathcal{E} = \mathcal{E}_v$, and so $\mu'\, l'' = \texttt{capsule}\, l$. This part of the proof is the most complex, as we need to use that fact that $\mu\, l''$ is the result of reducing an expression that was originally typed under $\widehat{\Gamma}$. Thus we need to reason over the entire reduction sequence starting from when the $\texttt{as}$ was initially introduced into the main expression, moreover, the $\widehat{\Gamma}$ typing does not actually prevent the $\texttt{as}$ from originally containing $\texttt{mut}$ references, rather it only restricts how the body of the $\texttt{as}$ can *use* them.

∗ Let σ_0 and e_0 be such that $\sigma_0|\mathcal{E}_v[e_0\,\texttt{as capsule}]$ is the earliest state in our reduction where $\sigma_0|e_0 \rightarrow^* \sigma|\mu\,l\,\texttt{as capsule}$. Thus, $\sigma_0|e_0\,\texttt{as capsule}$ is the state our $\mu\,l\,\texttt{as capsule}$ expression was in before its body began reduction. By definition of *validState* and our reduction rules we must have had that the $e_0\,\texttt{as capsule}$ expression was introduced by a method call.

∗ Thus there is some σ'_0, m, $l_0, .., l_n$, and $\mathcal{E}'_v$, where $\mathcal{E}'_v \in \mathcal{E}_v$ and we have a reduction sequence $\sigma'_0|\_\,l_0.m(\_\,l_1, .., \_\,l_n) \rightarrow \sigma'_0|\mathcal{E}_1[e_0\,\texttt{as capsule}] \rightarrow^* \sigma_0|\mathcal{E}'_v[e_0\,\texttt{as capsule}]$. By our CALL and CALL MUTATOR reduction rules, this $e_0\,\texttt{as capsule}$ expression must have come from the method body. Let $x_0 = \texttt{this}$ and $C_0 = \mathrm{C}^{\sigma'_0}_{l_0}$, then we have some e'_0 and $\mathcal{E}_2$ with:
$C_0.m = \mu_0\,\texttt{method}\,\_\,m(\mu_1\,C_1\,x_1, .., \mu_n\,C_n\,x_n)\;\mathcal{E}_2[e'_0\,\texttt{as capsule}]$, and
$e'_0[x_0 \coloneqq \mu_0\,l_0, .., x_n \coloneqq \mu_n\,l_n] = e_0$.
By our well-formedness rules on method bodies and the Nested Type lemma, we have $\emptyset;\Gamma \vdash e'_0\,\texttt{as capsule} :: \texttt{capsule}\,C$, where $\Gamma = \mu_0\,C_0 \mapsto x_0, .., \mu_n\,C_n \mapsto x_n$, for some C.

Suppose the typing rule used to get $\emptyset;\Gamma \vdash e'_0\,\texttt{as capsule} :: \texttt{capsule}\,C$ was TAs, then we have $\emptyset;\Gamma \vdash e'_0 : \texttt{capsule}\,C$. So by Valid Type, Method Type, and Substitution we have $\vdash \sigma'_0$ and $\sigma'_0;\emptyset \vdash e_0 : \texttt{capsule}\,C$, thus by Type Preservation we have $\mu = \texttt{capsule}$, and by the inductive hypothesis, we have *encapsulated*$(\sigma, \mathcal{E}_v[\Box\,\texttt{as capsule}], l)$, and so clearly we also have *encapsulated*$(\sigma, \mathcal{E}, l)$, a contradiction.

Thus the TAsCapsule type rule must have applied, and so $\emptyset;\widehat{\Gamma} \vdash e'_0 : \texttt{mut}\,C$. Consider each $i \in [0, n]$, we have $\widehat{\Gamma}(x_i) = \widehat{\mu_i}\,C_i$, and by Valid Type and Method Type we have $\mathrm{C}^{\sigma'_0}_{l_i} \leq C_i$. Now note that $e'_0[x_0 \coloneqq \widehat{\mu_0}\,l_0, .., x_n \coloneqq \widehat{\mu_n}\,l_n] = e'_0[x_0 \coloneqq \mu_0\,l_0, .., x_n \coloneqq \mu_n\,l_n][\mu_0\,l_0 \coloneqq \widehat{\mu_0}\,l_0, .., \mu_n\,l_n \coloneqq \widehat{\mu_n}\,l_n] = \widehat{e_0}$, this holds since by our well-formedness rules on method bodies, there are no ls in e'_0. Thus by Substitution, we have $\sigma_0;\emptyset \vdash \widehat{e_0} : \texttt{mut}\,C$, moreover, by Valid Type, we have $\vdash \sigma_0$.

∗ Now consider any l_1 and l_2 with $\texttt{mut}\,l_1 \in e_0$ and $l_2 \in ROG(\sigma_0, l_1)$.

Suppose *mutatable*$(\sigma_0, \widehat{e_0}, l_2)$, then since $\widehat{e_0}$ contains no $\texttt{mut}$ references, it follows that there is some $\mathcal{E}_3$ and l_3 with $e_0 = \mathcal{E}_3[\texttt{capsule}\,l_3]$. By the inductive hypothesis, we have *encapsulated*$(\sigma_0, \mathcal{E}_3, l_3)$. Since l_2 is clearly *mutatable* in $\mathcal{E}_3$, it follows that l_2 is not *reachable* in $\mathcal{E}_3$. But $\texttt{mut}\,l_1 \in \mathcal{E}_3$, and l_2 is *reachable* through l_1, a contradiction.

Thus we have not *mutatable*$(\sigma_0, \widehat{e_0}, l_2)$. Clearly $e_0 \sim \widehat{e_0}$, and since $\sigma_0|e_0 \rightarrow^* \sigma|\mu\,l$, by Bisimulation, there is some μ'' such that $\sigma_0|\widehat{e_0} \rightarrow^* \sigma|\mu''\,l$. Then, since we don't have *mutatable*$(\sigma_0, \widehat{e_0}, l_2)$, and since $\vdash \sigma_0$ and $\sigma;\emptyset \vdash \widehat{e_0} : \texttt{mut}\,C$, by Stronger Non-Mutating, we have $\sigma(l_2) = \sigma_0(l_2)$.

Suppose $l_2 \in MROG(\sigma, l)$. Since $tyr[\sigma_0]\widehat{e_0}\texttt{mut}\,C$, by Type Preservation it follows that $\mu'' \leq \texttt{mut}$ and hence *mutatable*$(\sigma, \mu''\,l, l_2)$. But $\sigma_0|\widehat{e_0} \rightarrow^* \sigma|\mu''\,l$ and not *mutatable*$(\sigma_0, \widehat{e_0}, l_2)$, so by Stronger Mut Consistency we have not *mutatable*$(\sigma, \mu''\,l, l_2)$, a contradiction.

Thus we must have $l_2 \notin MROG(\sigma, l)$.

∗ Now consider any l_4 where *reachable*$(\sigma_0, \mathcal{E}_v, l_4)$.

Suppose $\sigma_0(l_4) \neq \sigma(l_4)$. By Non Mutating, we must have some μ''', l_5, and $\mathcal{E}_4$ with $e_0 = \mathcal{E}_4[\mu'''\,l_5]$, $l_4 \in MROG(\sigma_0, l_5)$, and $\mu''' \leq \texttt{mut}$. By the above, if $\mu''' = \texttt{mut}$, then $\sigma_0(l_4) = \sigma(l_4)$, a contradiction. Hence $\mu''' = \texttt{capsule}$, and by the inductive hypothesis, we have that *encapsulated*$(\sigma_0, \mathcal{E}_v[\mathcal{E}_4], l_5)$. Thus, since l_4 is *mutatable* through $\mu'''\,l_5$, we can't have $\mathit{reachable}(\sigma_0, \mathcal{E}_v[\mathcal{E}_4], l_4)$, a contradiction.

Thus we must have $\sigma_0(l_4) = \sigma(l_4)$.

  * By the above, reduction cannot have modified memory in such a way as to make something *reachable* in $\sigma|\mathcal{E}_v$ that was not previously *reachable* in $\sigma_0|\mathcal{E}_v$. As $\mathit{reachable}(\sigma, \mathcal{E}_v, l')$, it follows that $\mathit{reachable}(\sigma_0, \mathcal{E}_v, l')$ and $l' \in \mathit{dom}(\sigma_0)$. Since $\mathit{mutable}(\sigma, \mathcal{E}_v[\texttt{capsule}\, l], l')$, by Mut Consistency, we have $\mathit{mutable}(\sigma_0, \mathcal{E}_v[e_0], l')$. Since $l' \in \mathit{ROG}(\sigma, l)$, it follows that $\mathit{reachable}(\sigma, \mu\, l, l')$ and so by Lost Forever we have some $\mu''''\, l'' \in e_0$ with $l' \in \mathit{ROG}(\sigma_0, l'')$.

    Suppose $\mu'''' = \texttt{capsule}$. By the inductive hypothesis, we have $\mathit{encapsulated}(\sigma_0, \mathcal{E}_v[\mathcal{E}'], l'')$, where $\mathcal{E}'[\texttt{capsule}\, l''] = e_0$. Since $\mathit{mutable}(\sigma_0, \mathcal{E}_v[e_0], l')$, from definition of *encapsulated*, we have not $\mathit{reachable}(\sigma_0, \mathcal{E}_v[\mathcal{E}'], l')$, and hence not $\mathit{reachable}(\sigma_0, \mathcal{E}_v, l')$. By the above, we can't have mutated anything *reachable* from $\mathcal{E}_v$, so there is no way we could have made $\mathit{reachable}(\sigma, \mathcal{E}_v, l')$ hold, a contradiction.

    Suppose $\mu'''' = \texttt{mut}$. Since $l' \in \mathit{ROG}(\sigma_0, l'')$ and $\texttt{mut}\, l'' \in e_0$, by the above $l' \notin \mathit{MROG}(\sigma, l'')$. Moreover, by the above we have $\mathit{ROG}(\sigma_0, l'') = \mathit{ROG}(\sigma, l'')$, so by Immutable ROG, we have $\mathit{immutable}(\sigma, \mathcal{E}_v[\texttt{capsule}\, l], l')$. Thus by the above Imm Consistency part of the proof, we have not $\mathit{mutable}(\sigma, \mathcal{E}_v[\texttt{capsule}\, l], l')$, a contradiction.

    Suppose $\mu'''' = \texttt{imm}$. Thus $\mathit{immutable}(\sigma_0, \mathcal{E}_v[e_0\ \texttt{as capsule}], l')$, and by the inductive hypothesis, we have not $\mathit{mutable}(\sigma_0, \mathcal{E}_v[e_0\ \texttt{as capsule}], l')$. Since $\sigma_0|\mathcal{E}_v[e_0\ \texttt{as capsule}] \rightarrow^* \sigma|\mathcal{E}_v[\texttt{capsule}\, l]$, by Mut Consistency, we have not $\mathit{mutable}(\sigma, \mathcal{E}_v[\texttt{capsule}\, l], l')$, a contradiction.

    Otherwise, $\mu'''' = \texttt{read}$. If l' is in the *ROG* of any non-`read` reference in e_0, then one of the above cases applies, and we would have a contradiction. If l' was in the *ROG* of any `imm` field in the *ROG* of l'', then $\mathit{immutable}(\sigma_0, \mathcal{E}_v[e_0\ \texttt{as capsule}], l')$ would hold, and by the case for $\mu'''' = \texttt{imm}$ above, we would also have a contradiction. Thus, l' must only be *reachable* through `read` references in e_0, and not through any `imm` fields. We now show that the body of the `as` expression never obtains a non-`read` reference to l', and so it cannot possibly store l' in the *ROG* of l. By Type Consistency and our typing rules, it follows that during reduction, a `read` reference cannot change reference capabilities (because our TAs, TAsCapsule, and TCall rules prohibit this), `read` references cannot be stored on the heap (our TUpdate rule prohibits this), and each field access on a `read` reference produces a `read` or `imm` reference (by definition of the access reduction rule). But, l' isn't in the *ROG* of any `imm` fields in σ_0, so if a field access on a `read` reference in σ_0 returns an `imm`, then l' is not *reachable* through the result of said access (by the access rule). Moreover, as we cannot store a `read` on the heap, during the reduction $\sigma_0|e_0 \rightarrow^* \sigma|\mu\, l$, l' will never enter the *ROG* of an `imm` field, and so will never become *reachable* through an `imm` reference. Thus we have that at each step of our $\sigma_0|e_0 \rightarrow^* \sigma|\mu\, l$ reduction: either l' is not *reachable*, or it is *reachable* only through `read` references. By Valid Type and our TAs and TAsCapsule rules, we have that $\mu \neq \texttt{read}$, hence l' cannot be *reachable* through $\mu\, l$. But we assumed that $l' \in \mathit{ROG}(\sigma, l)$, a contradiction.

- Otherwise, $\texttt{capsule}\, l \in \mathcal{E}_v$, and so by the new/new true case above, we have a contradiction.

- (try enter/try ok/try error/monitor exit) $\sigma|e' \rightarrow \sigma|\mathcal{E}[\texttt{capsule}\, l]$. These rules do not modify memory, introduce references in the main expression, nor change their reference capabilities. Thus it follows that $e' = \mathcal{E}'[\texttt{capsule}\, l]$, for some $\mathcal{E}'$. Furthermore, by the inductive hypothesis, we have $\mathit{encapsulated}(\sigma, \mathcal{E}', l)$, and by Mut Consistency, we have $\mathit{mutable}(\sigma, e', l')$, and so it follows that l' is not *reachable* in $\mathcal{E}'$. But these reduction rules do not introduce any references, duplicate them, nor modify memory since, thus as l' is *reachable* in $\mathcal{E}$, it follows that l' is *reachable* in $\mathcal{E}'$, a contradiction. □

The above theorem allows us to now directly prove the Imm Consistency and Capsule Consistency requirements themselves.

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

Proof. By definition of *immutable* it follows that l is *immutable* in $\mathcal{E}[e]$, thus by Imm–Capsule Consistency we have that l is not *mutatable* in $\mathcal{E}[e]$. By definition of *mutatable*, it follows that l is not *mutatable* in e either. ◻

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

Proof. Follows immediately from Imm–Capsule Consistency. ◻

Finally, we prove Strong Exception Safety, in a manner similar to how we proved the AS case for Capsule Consistency.

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

Proof. By definition of *validState* and our well-formedness rules on method bodies, we must have some e_0, and e_0' with $\mathit{validState}(\sigma, \mathcal{E}_v[\texttt{try \{}e_0\texttt{\} catch \{}e_0'\texttt{\}}])$ and $\sigma|\texttt{try \{}e_0\texttt{\} catch \{}e_0'\texttt{\}} \rightarrow \sigma|\texttt{try}^{\sigma}\texttt{\{}e_0\texttt{\} catch \{}e_0'\texttt{\}} \rightarrow^* \sigma' \mid \texttt{try}^{\sigma}\texttt{\{}e\texttt{\} catch \{}e'\texttt{\}}$. By our grammar for $\mathcal{E}_v$ and our reduction rules we also have $\sigma|e_0 \rightarrow^* \sigma'|e$ and $e_0' = e'$. By Valid State we have that the TTryCatch1 typing rule applied, and hence $\sigma; \emptyset \vdash \texttt{try \{}e\texttt{\} catch \{}e'\texttt{\}} : T$, $\sigma; \emptyset \vdash e_0 : T$, and $\sigma; \emptyset \vdash e' : T$, for some T. By definition of *validState* and our reduction rules we must have had that the $\texttt{try \{}e_0\texttt{\} catch \{}e_0'\texttt{\}}$ expression was introduced by a method call.

Thus there is some σ'', m, and $l_0, \ldots, l_n$, where $\sigma''|\_\, l_0.m(\_\, l_1, \ldots, \_\, l_n) \rightarrow \sigma''|\mathcal{E}[\texttt{try \{}e_0\texttt{\} catch \{}e_0'\texttt{\}}] \rightarrow^* \sigma|\mathcal{E}_v'[\texttt{try \{}e_0\texttt{\} catch \{}e_0'\texttt{\}}]$, where $\mathcal{E}_v' \in \mathcal{E}_v$. Let $x_0 = \texttt{this}$ and $C_0 = \mathrm{C}_{l_0}^{\sigma''}$, then by our CALL/CALL MUTATOR rules we have some e_1, e_1', and $\mathcal{E}'$ with $C_0.m = \mu_0\, \texttt{method}\, \_\, m(\mu_1\, C_1\, x_1, \ldots, \mu_n\, C_n\, x_n)\, \mathcal{E}'[\texttt{try \{}e_1\texttt{\} catch \{}e_1'\texttt{\}}]$ and $e_1[x_0 := \mu_0\, l_0, \ldots, x_n := \mu_n\, l_n] = e_0$. By Nested Type and our well-formedness rules on method bodies, we have that $\emptyset; \Gamma \vdash \texttt{try \{}e_1\texttt{\} catch \{}e_1'\texttt{\}} :: T'$ holds, for $\Gamma = \mu_0\, C_0 \mapsto x_0, \ldots, \mu_n\, C_n \mapsto x_n$, for some T'. Clearly the TTryCatch1 typing rule was used, and so we have $\sigma; \widehat{\Gamma} \vdash e_1 : T'$. As with the AS case in the Capsule Consistency part of the Imm–Capsule Consistency proof above, we have $e_1[x_0 := \widehat{\mu_0}\, l_0, \ldots, x_n := \widehat{\mu_n}\, l_n] = \widehat{e_0}$, where for each $i \in [0, n]$ we have $\widehat{\Gamma}(x_i) = \widehat{\mu_i}\, C_i$. Thus by Valid Type and Method Type, we we have $\mathrm{C}_{l_i}^{\sigma_0'} \leq C_i$, and by Substitution we have $\sigma; \emptyset \vdash \widehat{e_0} : T'$.

Now let $l \in \mathit{dom}(\sigma)$ with $\mathit{reachable}(\sigma, \mathcal{E}_v[e'], l)$. If we don't have $\mathit{reachable}(\sigma, e_0, l)$, then by Lost Forever, the reduction $\sigma|e_0 \rightarrow^* \sigma'|e$ cannot involve an UPDATE on l, i.e. we must have $\sigma'(l) = \sigma(l)$.

Suppose l is *mutatable* through a $\texttt{capsule}$ reference, i.e. we have some $\mathcal{E}''$, l', and l'', with $l' \in \mathit{ROG}(\sigma, l)$, $e_0 = \mathcal{E}''[\texttt{capsule}\, l'']$, and $l' \in \mathit{MROG}(\sigma, l'')$. Clearly we also have $\mathit{mutatable}(\sigma, \mathcal{E}_v[\texttt{try}^{\sigma}\texttt{\{}e_0\texttt{\} catch \{}e_0'\texttt{\}}], l)$, and since $\mathit{validState}(\sigma, \mathcal{E}_v[\texttt{try}^{\sigma}\texttt{\{}e_0\texttt{\} catch \{}e_0'\texttt{\}}])$, it follows from Capsule Consistency, that we do not have $\mathit{reachable}(\sigma, \mathcal{E}_v[\texttt{try}^{\sigma}\texttt{\{}\mathcal{E}\texttt{\} catch \{}e_0'\texttt{\}}], l)$. But this implies not $\mathit{reachable}(\sigma, \mathcal{E}_v, l)$, and since $e_0' = e'$, not $\mathit{reachable}(\sigma, e', l)$ holds. Thus we have not $\mathit{reachable}(\sigma, \mathcal{E}_v[e_0'])$, a contradiction.

Therefore, l is not *mutatable* through any $\texttt{capsule}$ reference in $\widehat{e_0}$, since such a reference would be in e_0, which yields a contradiction.

Since $\widehat{e_0}$ has no $\texttt{mut}$ references, it follows that not $\mathit{mutatable}(\sigma, \widehat{e_0}, l)$. Clearly $e_0 \sim \widehat{e_0}$, and since $\widehat{e_0}$ $\sigma|e_0 \rightarrow^* \sigma'|e$, by Bisimulation, there is some e'' such that $\sigma|\widehat{e_0} \rightarrow^* \sigma'|e''$. Moreover, by Valid Type, $\vdash \sigma$. Thus since $\sigma; \emptyset \vdash \widehat{e_0} : T'$ holds and not $\mathit{mutatable}(\sigma, \widehat{e_0}, l)$, by Stronger Non-Mutating, we have $\sigma(l) = \sigma'(l)$, as required. ◻