Using Capabilities for Strict Runtime Invariant Checking

Isaac Oscar Gariano, Marco Servetto*, Alex Potanin

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

Abstract

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

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

1. Formal Language Model

To model our system we need to formalise an imperative OO language with exceptions, object capabilities, and type system support for reference capabilities and strong exception safety. Formal models of the runtime semantics of such languages are simple, but defining and proving the correctness of such a type system is quite complex, and indeed many such papers exist that have already done this [? ? ? ? ?]. Thus we parametrise our language formalism, and assume we already have an expressive and sound type system enforcing the properties we need, so that we can separate our novel invariant protocol, from the non-novel reference capabilities. We clearly list in Appendix A the requirements we make on such a type system, so that any language satisfying them can soundly support our invariant protocol. In Appendix B we show an example type system, a restricted subset of L42, and prove that it satisfies our requirements. Conceptually our approach can parametrically be applied to any type system supporting these requirements, for example you could extend our type system with additional promotions or generic. To keep our small step reduction semantics as conventional as possible, we base our formalism on Featherweight Java [? ? , Chapter 19], which is a turing complete [?] 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 $C(T_1\,x_1, \ldots, T_n\,x_n)\{\texttt{this}.f_1 = x_1; \ldots; \texttt{this}.f_n = x_n\}$, 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:

*Corresponding Author

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

- 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 $v_1, ..., v_n$ and $l_1, ..., l_n$, 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, $\sigma[l.f = l']$ to update a field of l, $\sigma[l.f]$ to access one, and $\sigma \setminus l$ to delete l (this is only used in our proofs since our small step reduction does not need to delete individual locations). The notation σ, σ' combines the two memories, and requires that $dom(\sigma)$ is disjoint from $dom(\sigma')$.
- We assume a typing judgment of form $\sigma; \Gamma \vdash e : T$, this says that the expression e has type T, where the classes of any locations are stored in σ and the types of variables are stored in the environment $\Gamma : x \rightarrow T$.

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

- `Cap` has no fields,
- instances of `Cap` cannot be created with a `new` expression,
- `Cap`'s `invariant()` method is defined to have a body of '`new True()`', and
- `mut` methods on `Cap`, unlike all other methods, can have the same method name declared multiple times. To enable a typesystem to be sound, we require that methods with the same name have identical signatures. Such methods will model I/O, for example reading a byte from a file could be modelled by having several different `mut method Int 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 **??** is unnecessary for the soundness of our invariant protocol.

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

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

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

Figure 1: Grammar

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.[1]

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 pre existing 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 typesystem.

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 typesystem to ensure certain properties for all references with a given μ. The typesystem is then responsible for rejecting any `as` expression

[1] In particular, `if` statements can be supported using Church encoding: we would have a `Bool.if` method of form $\texttt{read method}\ T\ \texttt{if(}T\ \texttt{ifTrue,}T\ \texttt{ifFalse)}$, 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, $x\texttt{.if(}t\texttt{,}f\texttt{)}$ will return t if x is "true" and b if it is "false".

that could violate this. For example, a $\texttt{mut}\,l\,\texttt{as}\,\texttt{read}$ could be used to prevent l from being used for further mutation, and a $\texttt{mut}\,l\,\texttt{as capsule}$ (if accepted by the typesystem) will guarantee that l is properly *encapsulated*. These `as` expressions are merely a proof device, they do not effect the runtime behaviour, and as in L42, they could simply be inferred by the typesystem when it would be sound to do so.

- Monitor expressions ($\texttt{M(}l\texttt{;}\ e\texttt{;}\ e'\texttt{)}$) 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 $\texttt{imm}\,\texttt{True}$ (i.e. an `imm` reference to an instance of `True`), then the whole monitor expression will return the value of e, otherwise if it evaluates to a reference to a non-`True` value (i.e. an `imm` reference to an instance of a class other than `True`), the monitor expression is an *error*, and evaluation will proceed with the nearest enclosing `catch` block, if any.

- `try`–`catch` expressions ($\texttt{try \{}e\texttt{\} catch \{}e'\texttt{\}}$), which as in many other expression based languages[2], 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.

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(σ, e) iff:

- $e = \mathcal{E}_v[\texttt{M(}l\texttt{;}\ v\texttt{;}\ \mu\,l'\texttt{)}]$
- $C^{\sigma}_{l'} \neq \texttt{True}$
- $\mathcal{E}_v$ 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.

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.

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

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.

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).

[2]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 "$\texttt{try \{return}\,e\texttt{;\} catch (Throwable}\,t\texttt{) \{return}\,e'\texttt{;\}}$".

Reference Capability Operations

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

- $\mu \leq \mu'$ indicates that a reference of capability μ can be be used whenever 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 **??**, except that for simplicity method calls on `this` are not allowed.[3] This means that for every non-interface class C, $C.\texttt{invariant} = \texttt{read method imm Bool invariant()}$ where e can only use `this` as the receiver of an `imm` or `rep` field access. Formally, this means that forall $\mathcal{E}$ where $e = \mathcal{E}[\texttt{this}]$, we have:
  - $\mathcal{E} = \mathcal{E}'[\square.f]$, for some $\mathcal{E}'$
  - $C.f = \kappa\ \_\ f$
  - $\kappa \in \{\texttt{imm}, \texttt{rep}\}$
- Rep mutators must also follow the requirements in **??**, Such methods must not use `this`, except for the single access to the `rep` field, and they must not have `mut` or `read` parameters, or a `mut` return type. Formally, this means that for any C, m, and f, if $C.f = \texttt{rep}\ \_\ f$ and $C.m = \texttt{mut method}\ \mu'\ \_\ m\texttt{(}\mu_1\ \_\ \_, ..., \mu_n\ \_\ \_\texttt{)}\ \mathcal{E}[\texttt{this.}f]$:
  - $\texttt{this} \notin \mathcal{E}$
  - $\mu_1 \notin \{\texttt{mut}, \texttt{read}\}$, ..., $\mu_n \notin \{\texttt{mut}, \texttt{read}\}$
  - $\mu' \neq \texttt{mut}$
- We require that the method bodies do not contain runtime expressions. Formally, for all C_0 and m with $C_0.m = \_\texttt{method}\ \_\ m\texttt{(}\_\ \_, ..., \_\ \_\texttt{)}\ e$, e contains no l, $\texttt{M(}\_\texttt{;}\ \_\texttt{;}\ \_\texttt{)}$, or $\texttt{try}^{\sigma'}\texttt{\{}\_\texttt{\}}\ \texttt{catch}\ \texttt{\{}\_\texttt{\}}$ 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` $C\texttt{(}es\texttt{)}$ expression denotes a `class`; the `implements` relationship is acyclic; the fields of a `class` have unique names; methods within a `class`/`interface` (other than `mut` methods in `Cap`) have unique names; and parameters of a method have unique names and are not named `this`.
- For simplicity of the type-system and associated proof, we require that every method in the (indirect) super-interfaces of a class be implemented with exactly the same signature, i.e. if we have a $\texttt{class}\ C\ \texttt{implements}\ \_\ \texttt{\{}\_\texttt{;}\ Ms\texttt{\}}$, and $\texttt{interface}\ C'\ \texttt{implements}\ \_\ \texttt{\{}Ss\texttt{\}}$, where C' is reachable through the `implements` clauses starting from C, then for all $S \in Ss$, there is some e with $S\ e \in Ms$.

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

($\textsc{new}$) $\sigma|\mathcal{E}_v[\texttt{new}\ C(\_l_1,...,\_l_n)] \rightarrow \sigma, l_0 \mapsto C\{l_1,...,l_n\}|\mathcal{E}_v[\texttt{M}(l_0\texttt{;}\ \texttt{mut}\ l_0\texttt{;}\ \texttt{read}\ l_0\texttt{.invariant())}]$, where:
$l_0 = \mathit{fresh}(\sigma)$ and $C \neq \texttt{True}$
($\textsc{new true}$) $\sigma|\mathcal{E}_v[\texttt{new True()}] \rightarrow \sigma, l_0 \mapsto \texttt{True}\{\}|\mathcal{E}_v[\texttt{mut}\ l_0]$, where:
$l_0 = \mathit{fresh}(\sigma)$
($\textsc{access}$) $\sigma|\mathcal{E}_v[\mu\, l\texttt{.}f] \rightarrow \sigma|\mathcal{E}_v[\mu'\, l']$, where:
$\mathrm{C}^{\sigma}_{l}.f = \kappa\,\_f$, $\mu' = \mu{::}\kappa$, and $l' = \sigma[l.f]$

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

($\textsc{call}$) $\sigma|\mathcal{E}_v[\_l_0\texttt{.}m\texttt{(}\_l_1,...,\_l_n\texttt{)}] \rightarrow \sigma|\mathcal{E}_v[e'\ \texttt{as}\ \mu']$, where:
$\mathrm{C}^{\sigma}_{l_0}.m = \mu_0\ \texttt{method}\ \mu'\,\_m\texttt{(}\mu_1\,\_x_1,...,\mu_n\,\_x_n\texttt{)}\ e$
$e' = e[\texttt{this} := \mu_0\, l_0, x_1 := \mu_1\, l_1, ..., x_n := \mu_n\, l_n]$
if $\mu_0 = \texttt{mut}$ then $\nexists(f, \mathcal{E})$ with $\mathrm{C}^{\sigma}_{l_0}.f = \texttt{rep}\,\_f$ and $e = \mathcal{E}[\texttt{this.}f]$

($\textsc{call mutator}$) $\sigma|\mathcal{E}_v[\_l_0\texttt{.}m\texttt{(}\_l_1,...,\_l_n\texttt{)}] \rightarrow \sigma|\mathcal{E}_v[\texttt{M}(l_0\texttt{;}\ e\ \texttt{as}\ \mu'\texttt{;}\ \texttt{read}\ l_0\texttt{.invariant())}]$, where:

$\mathrm{C}^{\sigma}_{l_0}.m = \texttt{mut method}\ \mu'\,\_m\texttt{(}\mu_1\,\_x_1,...,\mu_n\,\_x_n\texttt{)}\ \mathcal{E}[\texttt{this.}f]$
$\mathrm{C}^{\sigma}_{l_0}.f = \texttt{rep}\,\_f$
$e = \mathcal{E}[\texttt{this.}f][\texttt{this} := \texttt{mut}\ l_0, x_1 := \mu_1\, l_1, ..., x_n := \mu_n\, l_n]$

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

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

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

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

($\textsc{monitor exit}$) $\sigma|\mathcal{E}_v[\texttt{M}(l\texttt{;}\ v\texttt{;}\ \mu\, l'\texttt{)}] \rightarrow \sigma|\mathcal{E}_v[v]$, where $\mathrm{C}^{\sigma}_{l'} = \texttt{True}$

Figure 2: Reduction rules

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

Reduction Rules

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

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

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

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

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

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

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

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

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

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

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

Statement of Soundness

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

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

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

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

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

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

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

The idea being that the $\mathcal{E}_r$ is like an $\mathcal{E}_v$ but it has a hole where a reference can be, thus $\mathit{trusted}(\mathcal{E}_r, l)$ holds when the very next reduction we are about to perform is $\mu\, l\texttt{.invariant()}$ or $\mu\, l\texttt{.}f$. As we discuss in our proof of Soundness, any such $\mu\, l\texttt{.}f$ expression came from the body of the `invariant()` method itself, since l can not occur in the *ROG* of any of its fields mentioned in the `invariant()` method.[4]

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

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

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

By restricting which initial expressions are well-typed, the type-system (such as the one presented in Appendix B) can ensure the required properties of our reference-capabilities (see Appendix A); any standard OO type system can also be used to reject expressions that might try to perform an ill-defined reduction (like reading a field that does not exist). The initial expression cannot contain any runtime expressions, except for `mut` references to the single pre-existing `Cap` object. Note that as `Cap` has no fields and `this` is not of form l, field accesses/updates in the initial main expression can never be reduced. To make the type system and proofs presented in Appendix B simpler, we require that c can only be initially referenced as `mut` and that there are no `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 $\mathit{validState}(\sigma, \mathcal{E}_r[\_\, l])$, then either $\mathit{valid}(\sigma, l)$ or $\mathit{trusted}(\mathcal{E}_r, l)$.

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

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

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

A. Invariant Protocol Proof and Type System Requirements

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

Auxiliary Definitions

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

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

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

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

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

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

- $l' = l$, or
- $\exists f$ such that $C^{\sigma}_{l}.f = \kappa\ \_\ f$, $\kappa \in \{\texttt{mut}, \texttt{rep}\}$, 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*[5] 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 $\mu \in \{\texttt{mut}, \texttt{capsule}\}$
- $l' \in MROG(\sigma, l'')$

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

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

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

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

Type System Requirements

As we do not have a concrete type system, we need to assume some properties about the expressions that it admits. Rather than requiring each expression during reduction to be well-typed, we instead let the

[5]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.

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

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

Requirement 1 (Type Consistency)**.**

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

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

[Isaac: Verify the above changes don't break anything!] 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 $\mathit{validState}(\sigma, \mathcal{E}[\texttt{capsule}\ l])$, then $\mathit{encapsulated}(\sigma, \mathcal{E}, l)$.

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

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

Requirement 5 (Mut Update)**.**
If $\mathit{validState}(\mathcal{E}[\mu\_\texttt{.}\_\texttt{=}\_])$, 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 $\mathit{validState}(\sigma', \mathcal{E}_v[\texttt{try}^{\sigma}\texttt{\{}e\texttt{\}}\ \texttt{catch}\ \texttt{\{}e'\texttt{\}}])$, then $\forall l \in \mathit{dom}(\sigma)$, if $\mathit{reachable}(\sigma, \mathcal{E}_v[e'], l)$, 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 $\mathit{validState}(\sigma', \mathcal{E}_v[\texttt{try}^{\sigma}\texttt{\{\_\}}\ \texttt{catch}\ \texttt{\{}e\texttt{\}}])$, then for any $l \in \mathit{dom}(\sigma')$, if $l \notin \mathit{dom}(\sigma)$, then l is not *reachable* in $\mathcal{E}_v[e]$.

Usefull Lemmas

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

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

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

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

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

Lemma 2 (No Dangling)**.**
If $\mathit{validState}(\sigma, e)$ then:

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

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 $l \in \mathit{dom}(\sigma)$, then $\mathit{ROG}(\sigma, l) \subseteq \mathit{dom}(\sigma)$, similarly with *MROG*.

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

Lemma 3 (Lost Forever)**.**
If $\mathit{validState}(\sigma, \mathcal{E}[e])$, and $\sigma|e \rightarrow^* \sigma'|e'$, then $\forall l \in \mathit{dom}(\sigma)$, if not $\mathit{reachable}(\sigma, e, l)$, then not $\mathit{reachable}(\sigma', e', l)$.

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

We can use our object capability discipline (described in Section 1) to prove that the `invariant()` method is deterministic and does not mutate existing memory:

Lemma 4 (Determinism)**.**
If $\mathit{validState}(\sigma, \mathcal{E}[\texttt{read}\ l\texttt{.invariant()}])$ and $\sigma|\texttt{read}\ l\texttt{.invariant()} \rightarrow^n \sigma'|e'$, for some $n \geq 0$, then:

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

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

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

Rep Field Soundness

[Isaac: Re-read everything from this point!] Now we define and prove important properties about our novel $\texttt{rep}$ 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 $\texttt{rep}$ field names for l:

$\mathit{repFields}(\sigma, l) = \{f \text{ where } C_l^\sigma.f = \texttt{rep}\,\_\, f\}$

We say that an l and f is *circular* if l is reachable from $l.f$:
$\mathit{circular}(\sigma, l, f)$ iff $l \in \mathit{ROG}(\sigma, \sigma[l.f])$.
We say that an l is *repCircular* if any its $\texttt{rep}$ fields are *circular*:
$\exists f \in \mathit{repFields}(\sigma, l)$ such that $\mathit{circular}(\sigma, l, f)$.

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

We say that an l is *repMutating* if we are in a monitor for l which must have been introduced by CALL MUTATOR:
$\mathit{repMutating}(\sigma, e, l)$ iff $e = \mathcal{E}[\texttt{M(}l\texttt{;}\, e'\texttt{;}\, \_\texttt{)}]$, 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 $\texttt{rep}$ field access:
$\mathit{headNotObservable}(\sigma, e, l)$ iff $e = \mathcal{E}_v[\texttt{M(}l\texttt{;}\, e'\texttt{;}\, \_\texttt{)}]$, and either:
- not $\mathit{reachable}(\sigma, e', l)$, or
- $e' = \mathcal{E}[\texttt{mut}\, l.f]$, $f \in \mathit{repFields}(\sigma, l)$, and not $\mathit{reachable}(\sigma, \mathcal{E}, l)$

Now we formally state the core properties of our $\texttt{rep}$ fields (informally described in **??**):
Theorem 2 (Rep Field Soundness)**.**
If $\mathit{validState}(\sigma, e)$ then $\forall l$ with $\mathit{reachable}(\sigma, e, l)$, we have:
- not $\mathit{repCircular}(\sigma, l, f)$, and
- one of the following holds:
  - $\mathit{repConfined}(\sigma, l)$ and not $\mathit{repMutating}(\sigma, e, l)$, or
  - $\mathit{headNotObservable}(\sigma, e, l)$.

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

In the inductive case, where $m > 0$, we have $\sigma_0|e_0 \rightarrow ... \rightarrow \sigma_{m-1}|e_{m-1} \rightarrow \sigma|e$, for some $\sigma_0, ..., \sigma_{m-1}$ and $e_0, ..., e_{m-1}$, where $\sigma_0|e_0$ is a valid initial memory and expression. Our inductive hypothesis is then that 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) $\sigma'|\mathcal{E}_v[\texttt{new}\ C\texttt{(}\mu_1\,l_1, ..., \mu_n\,l_n\texttt{)}] \rightarrow \sigma|\mathcal{E}_v[e']$,
   where $\sigma = \sigma', l_0 \mapsto C\{l_1, ..., l_n\}$; 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 $\forall l' \in dom(\sigma')$, $ROG(\sigma', l') \subseteq dom(\sigma')$. By our notational conventions for ",", it follows that $l_0 \notin dom(\sigma')$. Now consider each $i \in [1, n]$, since the pre-existing σ' was not modified, it follows that $ROG(\sigma', l_i) = ROG(\sigma, \sigma[l_0.f_i])$. By No Dangling we have that $ROG(\sigma, \sigma[l_0.f_i]) \subseteq dom(\sigma)$, and so $l_0 \notin ROG(\sigma, \sigma[l_0.f_i])$, 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 $ROG(\sigma', l_i)$ cannot be *mutatable* from $\mathcal{E}_v$. Thus, we don't have *mutatable*$(\sigma \setminus l_0, \mathcal{E}_v[e'], l_i)$, and so each of l_0s $\texttt{rep}$ fields is *confined*.
      - We trivially have that l_0 is not *repMutating* since $l_0 \notin dom(\sigma')$, 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 $f' \in repFields(\sigma', l')$, $l'.f'$ 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 $\texttt{mut}$ or $\texttt{rep}$ 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, $l' \in dom(\sigma')$, 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) $\sigma|\mathcal{E}_v[\mu\,l.f] \rightarrow \sigma|\mathcal{E}_v[\mu{::}\kappa\,\sigma[l.f]]$, where $\mathrm{C}^{\sigma}_{l}.f = \kappa\ \_\ f$:
   (a) No *reachable* l' is *repCircular*: this holds by the inductive hypothesis and the fact that we haven't mutated memory.
   (b) If l is *reachable* and it was *repConfined* and not *repMutating*, than it still is:
      - If $\kappa \neq \texttt{rep}$, then we can't have broken *confined* for any $f' \in repFields(\sigma, l)$, since by definition of *repConfined*, $\sigma[l.f']$ can't have been *mutatable* through $\sigma[l.f]$.
      - If $\kappa = \texttt{rep}$, since l' was not *repMutating*, this field access can't have been inside a rep mutator (or else we would be inside a monitor). As fields are instance private, we have $\mu \neq \texttt{mut}$, or else the field access would have come from a rep mutator.
        If $\mu = \texttt{capsule}$, then by Capsule Consistency and *repCircular*, l is not *reachable* from $\mathcal{E}_v[\mu{::}\kappa\,\sigma[l.f]]$, so it is irrelevant if l is no longer *repConfined*. Otherwise, since $\mu \notin \{Kwcapsule, \texttt{mut}\}$, we have $\mu{::}\kappa \not\leq \texttt{mut}$, so $l.f$ is still *confined*. By the above case for $\kappa \neq \texttt{rep}$, every other $f' \in repFields(\sigma, l)$ is *confined*.
      - We can't have made l' *repMutating* since we have introduced any monitor expressions.
   (c) If l was *repMutating* or not *repConfined*, than it is *headNotObservable*: by the inductive hypothesis, l was *headNotObservable* before this reduction, thus $\mathcal{E}_v = \mathcal{E}_v{}'[\texttt{M(}l\texttt{;}\ \mathcal{E}_v{}''\texttt{;}\ \_\texttt{)}]$. As l is clearly *reachable* in $\mathcal{E}_v{}''[\mu\,l.f]$, by definition of *headNotObservable* we must have that l is not *reachable* from $\mathcal{E}_v{}''$, and $\kappa = \texttt{rep}$. By *repCircular*, l is not in the ROG of $\sigma[l.f]$, and so l is not *reachable* from $\mathcal{E}_v{}''[\mu{::}\kappa\,\sigma[l.f]]$, and so it is still *headNotObservable*.

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

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

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

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

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

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

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

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

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

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

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

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

(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 $\mu' \nleq \texttt{mut}$, and by our well-formedness rules on method bodies, e doesn't contain any monitors, thus we can't have caused l_0 to be *repMutating*.
- Otherwise, the rule applied was CALL MUTATOR, and m is a rep mutator, so $e = \texttt{M(}l_0\texttt{;}\ e'\texttt{;}\ \texttt{read}\, l_0\texttt{.invariant())}$. By our rules for rep mutators, m must be a `mut` method with only `imm` and `capsule` parameters, thus by Type Consistency, $\mu_0 \leq \texttt{mut}$, and for each $i \in [1, n]$, $\mu_i \in \{\texttt{imm}, \texttt{capsule}\}$.
  By Imm Consistency and Capsule Consistency, l_0 can't be reachable from any l_i. Since rep mutators use `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 $\mu'\, l'\texttt{.}f \in e$. Moreover, by our well-formedness rules on method bodies, and the CALL/CALL MUTATOR rules, the only monitor that could be in e is a monitor on l_0, thus we can't have made l' *repMutating*.

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

5. (TRY ERROR) $\sigma|\mathcal{E}_v[\texttt{try}^{\sigma'}\texttt{\{}e\texttt{\}}\ \texttt{catch}\ \texttt{\{}e'\texttt{\}}] \rightarrow \sigma|\mathcal{E}_v[e']$, 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.[6]
- 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''.

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

- 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*.
- As with the above case for try error, by the inductive hypothesis, l must have been *headNotObservable*, and so the monitor must have been introduced by CALL MUTATOR.
- Thus, we were previously in a state $\sigma_i|e_i$ where $i \in [1, m-1]$, e_i is of form $\mathcal{E}_v[e']$, and the next state was obtained by said application of the CALL MUTATOR rule to e'.
- Thus, by the inductive hypothesis, in state i, l must have been *repConfined* and not *repMutating*.
- Because l was not *repMutating* in $\sigma_i|\mathcal{E}_v[e']$, and $\mu\, l'$ contains no monitors, l is not *repMutating* in $\mathcal{E}_v[\mu\, l']$.
- Since a rep mutator cannot have any `mut` parameters, by Type Consistency and Non-Mutating, the body of the method can only modify things *mutatable* through l, or a `capsule` parameter.
- By Type Consistency, and Capsule Consistency, every capsule parameter is *encapsulated*, and so anything mutated through such a parameter must have been un*reachable* outside the call.
- Thus, forall $l' \in \mathit{dom}(\sigma_i)$, if $\mathit{reachable}(\sigma_i, \mathcal{E}_v, l')$ and $l' \notin \mathit{MROG}(\sigma_i, l)$, 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*(σ, e) then $\forall l$, if *reachable*(σ, e, l), then *valid*(σ, l) or *monitored*(e, l).

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

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

1. (NEW) $\sigma'|\mathcal{E}_v[\texttt{new}\ C\texttt{(}\_l_1, .., \_l_n\texttt{)}] \rightarrow \sigma', l_0 \mapsto C\{l_1, .., l_n\}|\mathcal{E}_v[\texttt{M(}l_0\texttt{;}\ \texttt{mut}\ l_0\texttt{;}\ \texttt{read}\ l_0\texttt{.invariant())}]$:
   - Clearly the newly created object, l, is *monitored*.
   - This rule does not modify pre-existing memory, introduce pre-existing ls into the main expression, nor remove monitors on other ls, by the inductive hypothesis, every $l' \neq l_0$ is still *valid* (due to Determinism), or *monitored*.
2. (NEW TRUE) $\sigma'|\mathcal{E}_v[\texttt{new True()}] \rightarrow \sigma', l_0 \mapsto \texttt{True}\{\}|\mathcal{E}_v[\texttt{mut}\ l_0]$:

- The `True` class is required to have an invariant of `new True()`, so as with c in the base case above, we have that l_0 is *valid*.
- As in the above case for NEW, since we didn't modify pre-existing memory, introduce pre-existing ls into the main expression, nor remove monitors, by the inductive hypothesis, every $l' \neq l_0$ is still *valid* or *monitored*.

3. (UPDATE) $\sigma'|\mathcal{E}_v[\mu\, l.f = v] \to \sigma|\mathcal{E}_v[e']$, 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, `invariant()` can only accesses `imm` and `rep` fields, thus by Non-Mutating, and Determinism, we must have that l was in the ROG of $\sigma'[l'.f']$, for some $f' \in repFields(\sigma', l')$.

     Since $l \neq l'$, l' can't have been *repConfined*. Thus, by Rep Field Soundness, l' was *headNotObservable*, and so $\mathcal{E}_v[\mu\, l.f = v]$ is of form $\mathcal{E}_v{}'[\texttt{M}(l';\, e'';\, e''')]$:
     - As the ROG of l' has just been mutated, and since e''' must have started off as $\texttt{read}\, l'''\texttt{.invariant()}$, if follows from Determinism, that we cannot currently be inside e'''.
     - Thus, $\mathcal{E}_v = \mathcal{E}_v{}'[\texttt{M}(l';\, \mathcal{E}_v{}'';\, e''')]$, where $\mathcal{E}_v{}''[\mu\, l.f = v] = e''$.
     - 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 $e'' = \mathcal{E}[\texttt{mut}\, l'.f'']$ 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 $\mathcal{E}_v{}'[\texttt{M}(l';\, e'_i;\, \_)]$, immediately after a CALL MUTATOR, where e'_i is of form $\mathcal{E}'[\texttt{mut}\, l'.f''']$.
     - By Rep Field Soundness, l' is not *reachable* through $\sigma'[l'.f''']$,. By the proof for the CALL/CALL MUTATOR case of Rep Field Soundness, we have that l' is not *reachable* through $\mathcal{E}'$. Thus, by Lost Forever, once $\texttt{mut}\, l'.f'''$ has been reduced, l' must be un*reachable*, and it follows that $\texttt{mut}\, l'.f'' = \texttt{mut}\, l'.f'''$
     - 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 $l \in ROG(\sigma', \sigma'[l'.f'])$, 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 $\texttt{mut}\, l'.f''$, 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) $\sigma|\mathcal{E}_v[\texttt{try}^{\sigma'}\texttt{\{}e\texttt{\}}\ \texttt{catch}\ \texttt{\{}e'\texttt{\}}] \to \sigma|\mathcal{E}_v[e']$, where $error(\sigma, e) = \mathcal{E}_v{}'[\texttt{M}(l;\, \_;\, \_)]$:
   - As with the case for TRY ERROR in the proof of Rep Field Soundnes, we were previously in a state $\sigma_i|e_i$, where $e_i = \mathcal{E}_v[\texttt{try}^{\sigma'}\texttt{\{\_\}}\ \texttt{catch}\ \texttt{\{\_\}}]$, and $\sigma_i = \sigma'$.
   - By definition of *error*, we have that l is not *valid* in σ. [Isaac: Because monitors always start of ass invariant calls]
   - Suppose l is still *reachable* in $\sigma|\mathcal{E}_v[e']$, by Strong Exception Safety, we have $l \in dom(\sigma')$. 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.[Isaac: No new bullet point]

- But by our well-formedness rules on rep mutators, the body of the called method cannot mention l except to read a field, as shown in the case for UPDATE above, l will be un*reachable* once the field access has been reduced, which by Lost Forever is a contradiction, as l is *reachable* through e.
- Thus, l can't have been *monitored* in $\sigma'|e_i$, so it must have been *valid*.
- Also by Strong Exception Safety, we have that nothing reachable from l could have been modified, that is $\forall l' \in ROG(\sigma', l)$, we have $\sigma'(l') = \sigma(l')$. By Lost Forever, and our reduction rules, any memory location not *reachable* from a call `read` l`.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) $\sigma|\mathcal{E}_v[$`M(`l`;` v`;` `imm` l'`)`$] \rightarrow \sigma|\mathcal{E}_v[v]$, where $\mathrm{C}^{\sigma}_{l'} =$ `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 `read` l`.invariant()`, and it has now (eventually) been reduced to `imm` l', thus by Determinism l is *valid*.
   - As in the above case for NEW, since we didn't modify any memory, or remove any other monitors, by the inductive hypothesis every *reachable* $l' \neq l$ is still *valid* or *monitored*.
6. (ACCESS, CALL/CALL MUTATOR, AS, TRY ENTER, and TRY OK) these are trivial:
   - As in the above case for NEW, since these rules don't modify memory or remove monitors, by the inductive hypothesis, every *reachable* l is still *valid* or *monitored*. ☐

Proof of Soundness

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

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

Proof. The proof is by induction; obviously the property holds in the initial $\sigma|e$, since $\sigma = c \mapsto$ `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 $\mathrm{C}^{\sigma'}_{l}.f =$ `imm` $\_f$, by Imm Consistency and Non-Mutating, the only way $ROG(\sigma, \sigma[l.f])$ could have changed is if $e' = \mathcal{E}_v[\mu\, l.f = \mu'\, l']$, where $\mu \leq$ `mut`, i.e. we just applied the UPDATE rule. By Type Consistency, $\mu' \leq$ `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 $e = \mathcal{E}_v[$`new` C`(`$\_l_1, \ldots, \_l_n$`)`$]$, and each i with $C.i =$ `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, $monitored(e, l)$, $e = \mathcal{E}[$`M(`l`;` e_1`;` e_2`)`$]$, and either:

- $\mathcal{E}_r = \mathcal{E}[$`M(`$l$`;` $\mathcal{E}'$`;` e_2`)`$]$, that is l was found inside of e_1, but by definition of $\mathcal{E}_r$, we can't have $e_1 = \mu\, l$, this contradicts the definition of *monitored*, or

- $\mathcal{E}_r = \mathcal{E}[\mathtt{M}(l\,;e_1\,;\mathcal{E}')]$, and thus l was found inside e_2. By our reduction rules, all monitor expressions start with $e_2 = \mathtt{read}\,l\mathtt{.invariant()}$; if this has yet to be reduced, then $\mathcal{E}' = \mathcal{E}''[\square\mathtt{.invariant()}]$, thus *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, $\mathcal{E}' = \mathcal{E}''[\square.f]$: 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*$(\mathcal{E}_r, l)$. $\square$

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 $\mathtt{class}\ C\ \mathtt{implements}\ Cs\ \{\_;\_\}$ or $\mathtt{interface}\ C\ \mathtt{implements}\ Cs\ \{\_\}$ and $C' \in Cs$

Then we define subtyping:

$\mu\, C \leq \mu'\, C'$ iff $\mu \leq \mu'$ and $C \leq C'$

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

Now we define a notation that converts `mut` reference capabilities to `read`:

$\widehat{\mathtt{mut}} = \mathtt{read}$ and $\widehat{\mu} = \mu$, if $\mu \neq \mathtt{mut}$

Note that we always have $\mu \leq \widehat{\mu}$ 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 $\widehat{\emptyset} = \emptyset$ and $\widehat{\widehat{\Gamma}} = \widehat{\Gamma}$.

We also extend this to convert all `mut` references in an expression to `read`:

$\widehat{e} = e[\mu_1\, l_1 \coloneqq \widehat{\mu_1}\, l_1, .., \mu_n\, l_n \coloneqq \widehat{\mu_n}\, l_n]$, where $\{\mu_1\, l_1, .., \mu_n\, l_n\} = \{v \in e\}$

Finally, we define a notation to mean that two expressions are identical, except perhaps for reference capability annotations on references:

$e \sim e'$ iff $e[\mu_1\, l_1 \coloneqq \mathtt{read}\, l_1, .., \mu_n\, l_n \coloneqq \mathtt{read}\, l_n] = e'[\mu_1\, l_1 \coloneqq \mathtt{read}\, l_1, .., \mu_n\, l_n \coloneqq \mathtt{read}\, l_n]$,
where $\{\mu_1\, l_1, .., \mu_n\, l_n\} = \{v \in e\} \cup \{v \in e'\}$.

Note that the above requires that the μs of an `as` expression are the same, i.e. $e\ \mathtt{as}\ \mu \sim e'\ \mathtt{as}\ \mu'$ only if $\mu = \mu'$.

[Isaac: Actually I don't think this actually matters, so a simpler definition might be $e \sim e'$ iff:
$e[\mathtt{imm} \coloneqq \mathtt{read}, \mathtt{mut} \coloneqq \mathtt{read}, \mathtt{capsule} \coloneqq \mathtt{read}] = e'[\mathtt{imm} \coloneqq \mathtt{read}, \mathtt{mut} \coloneqq \mathtt{read}, \mathtt{capsule} \coloneqq \mathtt{read}]$]

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 super-type T', in particular this works with our reference capabilities, e.g. an expression of type $\mathtt{imm}\, C$ also has type $\mathtt{read}\, C$.

- 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.

$$(\textsc{TSub})\ \frac{\sigma;\Gamma\vdash e:T}{\sigma;\Gamma\vdash e:T'}\ T\leq T' \qquad (\textsc{TVar})\ \frac{}{\sigma;\Gamma\vdash x:\Gamma(x)} \qquad (\textsc{TRef})\ \frac{}{\sigma;\Gamma\vdash \mu\,l:\mu\,C_l^{\sigma}}$$

$$(\textsc{TNew})\ \frac{\begin{array}{c}\sigma;\Gamma\vdash e_1:\widetilde{\kappa_1}\,C_1\\ \vdots\\ \sigma;\Gamma\vdash e_n:\widetilde{\kappa_n}\,C_n\end{array}}{\sigma;\Gamma\vdash \mathtt{new}\ C\mathtt{(}e_1,\ldots,e_n\mathtt{)}:\mathtt{mut}\ C}\ \begin{array}{l}\mathtt{class}\ C\ \mathtt{implements}\ \_\ \mathtt{\{}Fs\mathtt{;}\_\mathtt{\}}\\ Fs=\kappa_1\,C_1\ \_,\ldots,\kappa_n\,C_n\ \_\end{array}$$

$$(\textsc{TAccess})\ \frac{\sigma;\Gamma\vdash e:\mu\,C}{\sigma;\Gamma\vdash e\mathtt{.}f:\mu{::}\kappa\,C'}\ C.f=\kappa\,C'\,f \qquad (\textsc{TUpdate})\ \frac{\begin{array}{c}\sigma;\Gamma\vdash e:\mathtt{mut}\,C\\ \sigma;\Gamma\vdash e':\widetilde{\kappa}\,C'\end{array}}{\sigma;\Gamma\vdash e\mathtt{.}f\,\mathtt{=}\,e':\mathtt{mut}\,C}\ C.f=\kappa\,C'\,f$$

$$(\textsc{TCall})\ \frac{\begin{array}{c}\sigma;\Gamma\vdash e_0:\mu\,C\\ \sigma;\Gamma\vdash e_1:T_1\\ \vdots\\ \sigma;\Gamma\vdash e_n:T_n\end{array}}{\sigma;\Gamma\vdash e_0\mathtt{.}m\mathtt{(}e_1,\ldots,e_n\mathtt{)}:T'}\ \begin{array}{l}S=\mu\,\mathtt{method}\,T'\,m\mathtt{(}T_1\ \_,\ldots,T_n\ \_\mathtt{)}\\ C.m\in\{S,S\ \_\}\end{array}$$

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

$$(\textsc{TTryCatch1})\ \frac{\begin{array}{c}\sigma;\widehat{\Gamma}\vdash e:T\\ \sigma;\Gamma\vdash e':T\end{array}}{\sigma;\Gamma\vdash \mathtt{try}\ \mathtt{\{}e\mathtt{\}}\ \mathtt{catch}\ \mathtt{\{}e'\mathtt{\}}:T} \qquad (\textsc{TTryCatch2})\ \frac{\begin{array}{c}\sigma;\Gamma\vdash e:T\\ \sigma;\Gamma\vdash e':T\end{array}}{\sigma;\Gamma\vdash \mathtt{try}^{\sigma'}\mathtt{\{}e\mathtt{\}}\ \mathtt{catch}\ \mathtt{\{}e'\mathtt{\}}:T}$$

$$(\textsc{TMonitor})\ \frac{\begin{array}{c}\sigma;\Gamma\vdash e:T\\ \sigma;\Gamma\vdash e':\mu\,\mathtt{Bool}\end{array}}{\sigma;\Gamma\vdash \mathtt{M(}l\mathtt{;}\ e\mathtt{;}\ e'\mathtt{)}:T}\ l\in dom\sigma$$

Figure B.3: Type rules

- TRef types a reference with the given capability by looking up the memory σ to determine the appropriate class. Note that this requires that $l \in dom(\S)$, 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 1 for the definition of $\widetilde{\kappa}$.

- TAccess types a field access expression by checking that the receiver has the given field. The $\mu{::}\kappa$ 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 this 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{G} \vdash e : $ `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 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 $\widehat{\Gamma}$.

  We could also extend our type system with more promotion rules (e.g. `read` to `imm`), but the TAsCapsule rule should be sufficient to demonstrate that our invariant protocol can be supported in a system with non-trivial 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 $\widehat{G}$. 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 to impose special restrictions once that `try–catch` block has begun executing, the check that $\emptyset; \widehat{\Gamma} \vdash e : T$ holds from within a method body is sufficient.
- 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 (if the invariant check succeeds) and the e' will be the `invariant` check itself. 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 $C_0.m = \mu_0$ `method` $T\, m(\mu_1\, C_1\, x_1, .., \mu_n\, C_n\, x_n)\, e$, we require:

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

Finally, we define a 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`;_}`
- $Fs = \_\, C_1\, \_, .., \_\, C_n\, \_$

- $\mathrm{C}^{\sigma}_{l_1} \leq C_1, \ldots, \mathrm{C}^{\sigma}_{l_n} \leq C_n$

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, each each field value has an appropriate class. Note that $\vdash \sigma$ doesn't require the field kinds are respected, this is ensured by our type system requirement proofs below.

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 $tyeeT'$ implies $tyeT'$, 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.

We note that if an expression is well-typed, then each sub-expression must also be well-typed. Note that the proof is non-trivial as we sometimes type a sub-expression 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. [Isaac: Marco, does this proof make sense?] 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 sub-expression 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 $(\textsc{TVar})\ \dfrac{}{\sigma;\widehat{\Gamma} \vdash x : \widehat{\Gamma}(x)}$ derivation with $(\textsc{TSub})\ \dfrac{(\textsc{TVar})\ \dfrac{}{\sigma;\Gamma \vdash x : \Gamma(x)} \quad \Gamma(x) \leq \widehat{\Gamma}(x)}{\sigma;\Gamma \vdash x : \widehat{\Gamma}(x)}$, the side condition trivially holds since we always have $\mu \leq \widehat{\mu}$. 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'. □

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 7 (Substitution)**.**

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

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

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

1. Suppose TVar applied, i.e. $e = x$ and $T = \Gamma(x)$.

- Thus there is some $i \in [1, n]$ with $x_i = x$ and so $e' = \mu_i\, l_i$.
- By the TVar 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 rule we have $\sigma; \emptyset \vdash e' : \Gamma(x_i)$, as required.

2. Suppose TAsCapsule applied, i.e. $e = e_0$ `as capsule` and $T =$ `capsule` C for some e_0 and C, where $\sigma; \widehat{\Gamma} \vdash e_0[$`mut` $c :=$ `read` $c] :$ `mut` C.
   - Thus $e' = e'_0$ `as capsule` where $e'_0 = e_0[x_1 := \mu_1\, l_1, .., x_n := \mu_n\, l_n]$.
   - Note that $dom(\widehat{\Gamma}) = \Gamma$, so consider each $i \in [1, n]$:
     - Let $\mu'_i\, C_i = \Gamma(x_i)$, then $\widehat{\Gamma}(x_i) = \widehat{\mu'_i}\, C_i$ and $\mu_i \leq \mu'_i$ 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 since $\widehat{\emptyset} = \emptyset$, we have $\sigma; \emptyset \vdash e'_0$ `as capsule` $:$ `capsule` C, as required.
3. Suppose TTryCatch1 applied, i.e. $e =$ `try {`e_0`} catch {`e_1`}` for some e_1 and e_1, where $\sigma; \widehat{\Gamma} \vdash e_0 : T$ and $\sigma; \Gamma \vdash e_1 : T$.
   - Thus $e' =$ `try {`e'_0`} catch {`e'_1`}` where $e'_0 = e_0[x_1 := \mu_1\, l_1, .., x_n := \mu_n\, l_n]$ and $e'_1 = e_1[x_1 := \mu_1\, l_1, .., x_n := \mu_n\, l_n]$.
   - By the above TAsCapsule case and the inductive hypothesis we have $\sigma; \emptyset \vdash e'_0 : T$.
   - By the inductive hypothesis, we have $\sigma; \emptyset \vdash e'_1 : T$.
   - Thus by the TTryCatch1 rule we have $\sigma; \emptyset \vdash$ `try {`e'_0`} catch {`e'_1`}` $: T$.
4. Otherwise, the TSub, TRef, TUpdate, TNew, TAccess, TTryCatch2, TMonitor, TCall, or TAs rules applied
   - The side conditions of these rules (if any) do not depend on the Γ 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 $\sigma; \Gamma \vdash e_0 : T_0$ where e_0 is a sub-expression of e.
     - Thus there is a corresponding sub-expression e'_0 of e' such that $e'_0 = e_0[x_1 := \mu_1\, l_1, .., x_n := \mu_n\, l_n]$.
     - 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 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 8 (Method Type).

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

Proof. By Type Rule we have that the TCall rule applied yielding $\sigma; \emptyset \vdash \mu_0\, l_0.m(\mu_1\, l_1, .., \mu_n\, l_n) : T''$ for some $T'' \leq T$.

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

   - By Type Rule and our TRef rule, 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 $C^\sigma_{l_0} = C$.
   - Thus we have $C.m \in \{S, S\,e\}$.
   - By the TCall rule, this means that $\mu = \mu'_0$, thus $\mu_0 \leq \mu'_0$.
3. Now we show that for each $i \in [1, n]$, $\mu_i\, C^\sigma_{l_i} \leq \mu'_i\, C_i$:
   - 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 Type Rule and our TRef rule, we thus have $\mu_i\, C^\sigma_{l_i} \leq \mu'_i\, C_i$.
4. Now we show $\sigma; \emptyset \vdash e[\texttt{this} := \mu'_0\, l_0, x_1 := \mu'_1\, l_1, ..., x_n := \mu'_n\, l_n] : T'$
   - By our well-formedness rules on methods, we have $\emptyset; \Gamma \vdash e : T'$, where $\Gamma = \texttt{this} \mapsto \mu'_0\, C^\sigma_{l_0}, x_1 \mapsto \mu'_1\, C_1, ..., x_n \mapsto \mu'_n\, C_n$.
   - We also have $\sigma; \Gamma \vdash e : T'$, since no typing rule depends on what is *not* contained within the memory.
   - 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 $\sigma; \emptyset \vdash e[\texttt{this} := \mu'_0\, l_0, x_1 := \mu'_1\, l_1, ..., x_n := \mu'_n\, l_n] : T'$.
5. Finally, we have $T' \leq T$ since $C.m \in \{S, S\,e\}$, by the TCall rule, we have $T'' = T'$. □

Now a couple of auxiliary lemmas that are needed to reason over the types of reducing an expression. First a monitor can be trivially added over a well typed expression.

Lemma 9 (Monitor Type)**.**
If $\vdash \sigma$, $l \in dom(\sigma)$, and $\sigma; \emptyset \vdash e : T$ then $\sigma; \emptyset \vdash \texttt{M}(l;\, e;\, \texttt{read}\, l\texttt{.invariant()}) : T$.

Proof. We can construct the following typing derivation:

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

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. □

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

Lemma 10 (Ref Type)**.**
$\sigma; \emptyset \vdash \mu\, l : T$ if and only if $\mu\, C^\sigma_l \leq T$.

Proof. Follows immediately from Type Rule and the TRef and TSub typing rules. □

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 bellow.

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. First we assume that $n = 1$, i.e. $\sigma|e \rightarrow \sigma'|e'$. We note By Type Rule we can obtain a T' such that $\sigma; \emptyset \vdash e : T'$ holds by a rule other than TSub and where $T' \leq T$. We will then show that $\sigma'; \emptyset \vdash e' : T'$ holds by induction on the size of e:

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

  1. Suppose that the new/new true rule was applied, i.e. we have $e = \texttt{new}\ C(\mu_1\, l_1, ..., \mu_n\, l_n)$, $\sigma' = \sigma, l_0 \mapsto C\{l_1, ..., l_n\}$, and $e' \in \{\texttt{mut}\, l_0, \texttt{M}(l_0;\, \texttt{mut}\, l_0;\, \texttt{read}\, l_0\texttt{.invariant()})\}$, and $l_0 = fresh(\sigma)$:

- By the TNew typing rule, we have $T' = \mathtt{mut}\,C$, a declaration $\mathtt{class}\ C\ \mathtt{implements}\ \_\ \mathtt{\{}Fs\mathtt{;}\_\mathtt{\}}$ with $Fs = \kappa_1\,C_1\ \_, ..., \kappa_n\,C_n\ \_$.
- Consider each $i \in [1, n]$, clearly $\mathrm{C}^{\sigma'}_{l_i} = \mathrm{C}^{\sigma}_{l_i}$, moreover 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 $\mathrm{C}^{\sigma'}_{l_i} \leq C_i$.
- Since $l_0 = \mathit{fresh}(\sigma)$, $l_0 \notin \mathit{dom}(\sigma)$.
- Thus, since $\vdash \sigma$, we have $\mathtt{class}\ C\ \mathtt{implements}\ \_\ \mathtt{\{}Fs\mathtt{;}\_\mathtt{\}}$, and the above, we have $\vdash \sigma'$.
- Clearly $\mathrm{C}^{\sigma'}_{l_0} = C$, so by TRef we have $\sigma'; \emptyset \vdash \mathtt{mut}\,l_0 : \mathtt{mut}\,C$.
- If $e' = \mathtt{mut}\,l_0$ then we are done.
- Otherwise, $e' = \mathtt{M(}l_0\mathtt{;}\,\mathtt{mut}\,l_0\mathtt{;}\,\mathtt{read}\,l_0\mathtt{.invariant())}$ so by Monitor Type, we have $\sigma'; \emptyset \vdash e' : \mathtt{mut}\,C$ as required.

2. Suppose the access rule was applied, i.e. we have $e = \mathcal{E}_v[\mu\,l\mathtt{.}f]$, $\sigma' = \sigma$, 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 rules we have $\mu \leq \mu'$ and $\mathrm{C}^{\sigma}_{l} \leq C'$.
   - Since $\vdash \sigma$ we have 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 $\kappa = \mathtt{imm}$ then $\mu{::}\kappa = \mu'{::}\kappa = \mathtt{imm}$ 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 $\mathrm{C}^{\sigma}_{\sigma[l.f]} \leq C$, so since $\mu{::}\kappa \leq \mu'{::}\kappa$, by Ref Type, we have $\sigma; \emptyset \vdash \mu{::}\kappa\,\sigma[l.f] : T'$, as required.
3. Suppose the update rule was applied, i.e. we have $e = \mathcal{E}_v[\mu\,l\mathtt{.}f\,\mathtt{=}\,\mu'\,l']$, $\sigma' = \sigma[l.f = l']$, and $e' = \mathtt{M(}l\mathtt{;}\,\mathtt{mut}\,l\mathtt{;}\,\mathtt{read}\,l\mathtt{.invariant())}$:
   - By the TUpdate typing rule, we have $\sigma; \emptyset \vdash \mu\,l : \mathtt{mut}\,C$ 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 $T' = \mathtt{mut}\,C$.
   - 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 $\vdash \sigma$ and σ' differs from σ only at $l.f$, we thus have $\vdash \sigma'$.
   - By the TRef typing rule we have $\sigma; \emptyset \vdash \mathtt{mut}\,l : \mathtt{mut}\,C$, thus by Monitor Type we have $\sigma; \emptyset \vdash e' : \mathtt{mut}\,C$ as required:
4. Suppose that the call/call mutator rule was applied, i.e. we have $e = \_l_0\mathtt{.}m\mathtt{(}\_l_1, ..., \_l_n\mathtt{)}$, $\sigma' = \sigma$, and $e' \in \{e''\,\mathtt{as}\,\mu', \mathtt{M(}l_0\mathtt{;}\,e''\,\mathtt{as}\,\mu'\mathtt{;}\,\mathtt{read}\,l_0\mathtt{.invariant())}\}$, $e'' = e'''[\mathtt{this} \coloneqq \mu_0\,l_0, x_1 \coloneqq \mu_1\,l_1, ..., x_n \coloneqq \mu_n\,l_n]$, and $\mathrm{C}^{\sigma}_{l_0} = \mu_0\,\mathtt{method}\,\mu'\,C\,m\mathtt{(}\mu_1\,\_x_1, ..., \mu_n\,\_x_n\mathtt{)}\,e'''$:
   - By Method Type we have $\sigma; \emptyset \vdash e'' : \mu'\,C'$ and so $\mu'\,C \leq T'$.
   - Thus, by the TAs typing rule we trivially have $\sigma; \emptyset \vdash e''\,\mathtt{as}\,\mu' : \mu'\,C$, since $\mu' \leq \mu'$.
   - By the TSub typing rule we thus have $\sigma; \emptyset \vdash e''\,\mathtt{as}\,\mu' : T'$.
   - If $e' = e''\,\mathtt{as}\,\mu'$ then we are done.
   - Otherwise, $e' = \mathtt{M(}l_0\mathtt{;}\,e''\,\mathtt{as}\,\mu'\mathtt{;}\,\mathtt{read}\,l_0\mathtt{.invariant())}$ so by Monitor Type, we have $\sigma; \emptyset \vdash e' : T'$ as required.
5. Suppose the as rule was applied, i.e. we have $e = \mu\,l\,\mathtt{as}\,\mu'$, $\sigma' = \sigma$, and $e' = \mu'\,l$:
   - By the TAs and TAsCapsule 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 $\mathrm{C}^{\sigma}_{l} \leq C$, thus by Ref Type we have $\sigma; \emptyset \vdash \mu'\,l : \mu'\,C$ as required.
6. Suppose the try enter rule was applied, i.e. we have $e = \mathtt{try}\ \mathtt{\{}e_1\mathtt{\}}\ \mathtt{catch}\ \mathtt{\{}e_2\mathtt{\}}$, $\sigma' = \sigma$, and $e' = \mathtt{try}^{\sigma}\mathtt{\{}e_1\mathtt{\}}\ \mathtt{catch}\ \mathtt{\{}e_2\mathtt{\}}$:

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

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

- In the inductive case, we assume the base case does not hold, i.e. we have some e_0 and minimal $\mathcal{E}_v \neq \Box$ where $e = \mathcal{E}_v[e_0]$, i.e. e_0 is a direct sub-expression of e.
  - [Isaac: This part is horribly worded...]
  - By the structure of our reduction rules we have $\sigma | e_0 \to \sigma' | e'_0$ where $e' = \mathcal{E}_v[e_0]$.
  - Clearly e' is not of form v, so the typing rule use to obtain $\sigma; \emptyset \vdash \mathcal{E}_v[e_0] : T'$ must not have been TSub, TVar, or TRef.
  - Each such rule will have a premise of form $\sigma; \emptyset \vdash e_0 : T_0$, for some T_0, and since we have $\sigma | e_0 \to \sigma' | e'_0$, by the inductive hypothesis we have $\vdash \sigma$ and $\sigma'; \emptyset \vdash e'_0 : T''$.
  - Now note that regardless of the reduction rule applied to get $\sigma | e_0 \to \sigma' | e'_0$, we have $\forall l \in \mathit{dom}(\sigma)$, $\mathrm{C}^{\sigma}_{l} = \mathrm{C}^{\sigma'}_{l'}$, so for any e_1 and T_1, if we we have $\sigma; \emptyset \vdash e_1 : T_1$ then 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 no altered the value of any C^{σ}_{l}, such a rule is still valid under σ').
  - Now we will modify the rule application that gave us $\sigma; \emptyset \vdash \mathcal{E}_v[e_0] : T'$ as follows:
    * Change the conclusion to be $\sigma'; \emptyset \vdash \mathcal{E}_v[e'_0] : T'$
    * Change the premise for the sub-expression in the hole of $\mathcal{E}_v$, which will be of form $\sigma; \emptyset \vdash e_0 : T_0$, to be $\sigma'; \emptyset \vdash e'_0 : T''$
    * For every other premise, which will be of form $\sigma; \emptyset \vdash e_1 : T_1$, for some e_1 and T_1, change it to be $\sigma'; \emptyset \vdash e_1 : T_1$
  - By looking at each typing rule, it can be seen that the above transformation will respect the rule since:
    * As shown above, each premise is valid
    * The transformation we have applied to the premises is consistent with the transformation of the conclusion
    * Each side-condition is still valid, as they do not depend on the value of σ nor the values of any sub-expressions (this holds as we have preserved the *types* of these sub-expressions)
  - Thus we have $\sigma; \emptyset \vdash \mathcal{E}_v[e'_0] : T'$ as required.
- Now we proceed by induction on n. In the base case, $n = 0$ and the conclusion trivially holds since $\sigma' = \sigma$ and $e' = e$. 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 that $\vdash \sigma_k$ and $\sigma_k; \Gamma \vdash e_k : T$ and so by the above case for $n = 1$, we have $\vdash \sigma'$ and $\sigma'; \emptyset \vdash e' : T$ as required. $\Box$

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

Corollary 2 (Valid Type)**.**
If $\mathit{validState}(\sigma, \mathcal{E}[e])$ 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 $\sigma_0; \emptyset \vdash e_0 : T_0$, $\sigma_0 = c \mapsto$ `Cap`$\{\}$ and $\sigma_0|e_0 \to^* \sigma|\mathcal{E}[e]$. Clearly $\vdash \sigma_0$ and `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'. □

Now 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 `mut` $l' \in e$ or `capsule` $l' \in e$, then $mutatable(\sigma, e, l)$.

Proof.

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

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. First we will assume that $n = 1$. 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 $e_2 \sim e'_2$.

1. 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 $e'_2 = \mu'{::}\kappa\, \sigma_0[l.f]$, and 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.
2. Suppose the TRY ERROR rule applied, i.e. $e_1 =$ `try`$^{\sigma'}$`{`e_3`} catch {`e_4`}`, $\sigma = \sigma_0$, and $e_2 = e_4$, where $error(\sigma, e_3)$:
   - Since $e_1 \sim e'_1$, we have $e'_1 =$ `try`$^{\sigma'}$`{`e'_3`} 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 $error(\sigma, e'_3)$.
   - 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.
3. Suppose the MONITOR EXIT rule applied, i.e. $e_1 =$ `M(`l`;` v`;` $\mu\, l'$`)`, $\sigma = \sigma_0$, and $e_2 = v$, where $\mathrm{C}^{\sigma_0}_{l'} =$ `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 $e'_1 =$ `M(`l`;` v'`;` $\mu'\, l'$`)`, with $v \sim v'$, and we set $e'_2 = v'$.
4. Suppose the TRY ENTER rule applied, i.e. $e_1 =$ `try {`e_3`} catch {`e_4`}`, $\sigma = \sigma_0$, and $e_2 =$ `try`$^{\sigma_0}$`{`e_3`} catch {`e_4`}`:
   - This is similar to the TRY ERROR case above, except that we have $e'_2 =$ `try {`e'_3`} catch {`e'_4`}`, with $e_3 \sim e'_3$ and $e_4 \sim e'_4$, and we set $e'_2 =$ `try`$^{\sigma_0}$`{`e'_3`} catch {`e'_4`}`.
5. Suppose the TRY OK rule applied, i.e. $e_1 =$ `try`$^{\sigma'}$`{`v`} catch {_}`, $\sigma = \sigma_0$, and $e_2 = v'$:
   - This is similar to the TRY ERROR case above, except that we have $e'_2 =$ `try`$^{\sigma'}$`{`v'`} catch {_}`, with $v \sim v'$, and we set $e'_2 = v'$.
6. Otherwise the AS/NEW/NEW TRUE/UPDATE/CALL/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[7] 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. [Isaac: Note that this only works BECAUSE these rules do not do any sanity checking on the μs.]

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 $e' = \mathcal{E}_v'[e'_2]$, then we have $\sigma_0|e'_0 \to \sigma|e'$ and $e \sim e'$, as required.

Now we proceed by induction on n. In the base case, $n = 0$, 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 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 above 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 bellow: 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).

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. [Isaac: Actually check this!] 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:

1. Suppose the NEW rule matches, and so $e = \mathcal{E}_v[\texttt{new}\ C(\_l_1, ..., \_l_n)]$. $fresh(\sigma)$ is well-defined since there is always some $l_0 \notin dom(\sigma)$. Thus, we must have $C = \texttt{True}$. By definition, the `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..
2. Suppose the NEW TRUE rule applies, then as with NEW above, the side condition is satisfiable, a contradiction.
3. Suppose the ACCESS rule matches, and so $e = \mathcal{E}_v[\mu\, l.f]$. 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 $C^\sigma_l \leq C$. 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 $C^\sigma_l = C$. 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.
4. Suppose the UPDATE rule matches, and so $e = \mathcal{E}_v[\_l.f = \_l']$. 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 $C^\sigma_l = C$ 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.

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

5. Suppose the CALL rule matches, and so $e = \mathcal{E}_v[\_\,l_0\texttt{.}m\texttt{(}\_\,l_1, ..., \_\,l_n\texttt{)}]$. By Method Type, we have that $\mathrm{C}^{\sigma}_{l_0}.m = \mu_0\,\texttt{method}\,\mu'\,\_\,m\texttt{(}\mu_1\,\_\,x_1, ..., \mu_n\,\_\,x_n\texttt{)}\;e'$ 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.
6. Suppose the CALL MUTATOR rule matches, and so $e = \mathcal{E}_v[\_\,l_0\texttt{.}m\texttt{(}\_\,l_1, ..., \_\,l_n\texttt{)}]$. As above, by Method Type, we have that $\mathrm{C}^{\sigma}_{l_0}.m = \mu_0\,\texttt{method}\,\mu'\,\_\,m\texttt{(}\mu_1\,\_\,x_1, ..., \mu_n\,\_\,x_n\texttt{)}\;e'$. 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.
7. 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 $\mathcal{E}_v$.
8. 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 $\mathrm{C}^{\sigma}_{s}l' \neq \texttt{true}$. Thus $error(\sigma, e')$. If $\mathcal{E}_v$ is of form $\mathcal{E}_v{}'[\texttt{try}^{\sigma'}\texttt{\{}\mathcal{E}_v{}''\texttt{\}}\;\texttt{catch}\;\texttt{\{\_\}}]$, 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 $\mathrm{C}^{\sigma}_{s}l' \neq \texttt{true}$, we have that $error(\sigma, \mathcal{E}_v[e'])$ holds.
9. 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. ☐

Thus our type system satisfies the conventional soundness notion of Progress + Type Preservation.

Proof of Type System Requirements

Finally we prove each of the requirements from Appendix A.

Requirement 1 (Type Consistency)**.**

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

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. ☐

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. ☐

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 $\sigma|e \to^n \sigma'|e'$, then not $mutatable(\sigma', e', l)$.

Proof. First we will 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 $mutatable$:

1. Suppose the UPDATE rule was applied, i.e. we have some $\mathcal{E}_v$ with $e = \mathcal{E}_v[\mu\, l'.f = \mu'\, l'']$, $\sigma' = \sigma[l'.f = l'']$, and $e' = \mathcal{E}_v[\mathtt{M(}l'\mathtt{;}\ \mathtt{mut}\, l'\mathtt{;}\ \mathtt{read}\, l'\mathtt{.invariant())}]$.
   - By Type Preservation, Type Rule, and our TUPDATE typing rule, we have $\mu \leq \mathtt{mut}$.
   - 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 $\mathtt{mut}\, l_2 \in e'$ or $\mathtt{capsule}\, l_2 \in e'$, and $l_1 \in MROG(\sigma', l_2)$.
   - Since $\sigma' = \sigma[l'.f = v]$ and l was not previously $mutatable$, we must have made $l_1 \in MROG(\sigma', l_2)$ through the fact that $\sigma'(l'.f) = l''$, and so we have that $C^{\sigma'}_{l'}.f = \kappa\, C\, f$ for some $\kappa \in \{\mathtt{mut}, \mathtt{rep}\}$.
   - Thus before the reduction, we had $l_1 \in MROG(\sigma, l'')$ and $l' \in MROG(\sigma, l_2)$. [Isaac: Thus we are extending the $MROG$s by making $l'.f \to l''$]
   - By Type Preservation, Type Rule, and our TUPDATE typing rule, we have that $\mu' \in \{\mathtt{mut}, \mathtt{capsule}\}$.
   - Since $l_1 \in MROG(\sigma, l'')$ and $l_1 \in ROG(\sigma, l)$, we thus have $mutatable(\sigma, e, l)$, a contradiction.
2. 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 $C^{\sigma}_{l'}.f = \kappa\, C\, f$.
   (a) As we have not modified memory, the only way for l to have become $mutatable$ is via v, i.e. we must have $\mu{::}\kappa \leq \mathtt{mut}$ and some $l'' \in ROG(\sigma, l)$ such that $l'' \in MROG(\sigma, \sigma[l'.f])$.
   (b) By definition of $\mu{::}\kappa$ this implies that $\kappa \in \{\mathtt{mut}, \mathtt{rep}\}$ and $\mu \leq \mathtt{mut}$. So we have that $l'' \in MROG(\sigma, l')$, and $\mathtt{mut}\, l' \in e$ or $\mathtt{capsule}\, l' \in e$.
   (c) Thus we must have $mutatable(\sigma, e, l)$, a contradiction.
3. Suppose that the NEW/NEW TRUE rule was applied, i.e. we have some $\mathcal{E}_v$ with $e = \mathcal{E}_v[\mathtt{new}\, C\mathtt{(}\mu_1\, l_1, \ldots, \mu_n\, l_n\mathtt{)}]$, $\sigma' = \sigma, l' \mapsto C\{l_1, \ldots, l_n\}$, and $e' \in \{\mathcal{E}_v[\mathtt{M(}l'\mathtt{;}\ \mathtt{mut}\, l'\mathtt{;}\ \mathtt{read}\, l'\mathtt{.invariant())}], \mathcal{E}_v[\mathtt{mut}\, l']\}$.
   (a) Since no-preexisting part of σ is modified, we must have that l is now $mutatable$ through the $\mathtt{mut}\, l'$ reference in e', i.e. we must have some $l'' \in ROG(\sigma, l)$ with $l'' \in MROG(\sigma', l')$.
   (b) By No Dangling we have $l'' \neq l'$, thus we have that $i \in [1, n]$, $C.i = \kappa\, C'\, f$, $\kappa \in \{\mathtt{mut}, \mathtt{rep}\}$, and $l'' \in MROG(\sigma, l_i)$.
   (c) By Type Preservation, Type Rule, and our TNEW typing rule, we have that $\mu_i \leq \mathtt{mut}$.
   (d) Since $l'' \in MROG(\sigma, l_i)$ and $l'' \in ROG(\sigma, l)$, we thus have $mutatable(\sigma, e, l)$, a contradiction.
4. Suppose the AS rule was applied, i.e. we have some $\mathcal{E}_v$ with $e = \mathcal{E}_v[\mu\, l'\ \mathtt{as}\ \mu']$, $\sigma' = \sigma$, and $e' = \mathcal{E}_v[\mu'\, l']$
   (a) By Type Preservation and Type Rule either the TAS or TASCAPSULE typing rule applied.
   (b) In either case, by Ref Type we have that $\mu' \leq \mathtt{mut}$ only if $\mu \leq \mathtt{mut}$.
   (c) As we haven't introduced any other reference or modified any memory, we must have that l is now $mutatable$ through $\mu'\, l'$.
   (d) But them $\mu' \leq \mathtt{mut}$ and so $\mu \leq \mathtt{mut}$, and hence l was already $mutatable$ through $\mu\, l$, a contradiction.
5. Suppose that the CALL/CALL MUTATOR rule was applied, i.e. we have some $\mathcal{E}_v$ with $e = \mathcal{E}_v[\mu_0\, l_0.m\mathtt{(}\mu_1\, l_1, \ldots, \mu_n\, l_n\mathtt{)}]$, $\sigma' = \sigma$, and $e' \in \{e''\ \mathtt{as}\ \mu'', \mathtt{M(}l_0\mathtt{;}\ e''\ \mathtt{as}\ \mu''\mathtt{;}\ \mathtt{read}\, l_0\mathtt{.invariant())}\}$, $e'' = e'''[\mathtt{this} := \mu'_0\, l_0, x_1 := \mu'_1\, l_1, \ldots, x_n := \mu'_n\, l_n]$, and $C^{\sigma}_{l_0} = \mu'_0\ \mathtt{method}\ T\, m\mathtt{(}\mu'_1\ \_x_1, \ldots, \mu'_n\ \_x_n\mathtt{)}\ e'''$
   (a) As we haven't modified memory, for this reduction to have made l $mutatable$, we must have introduced a $\mathtt{mut}$ or $\mathtt{capsule}$ reference in e''.
   (b) 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 $\mu'_i \leq \mathtt{mut}$.
   (c) By Type Preservation and Method Type, we have that $\mu_i \leq \mu'_i$, and so $\mu_i \leq \mathtt{mut}$ and so e already had a reference, $\mu_i\, l_i$, through which l was $mutatable$, a contradiction.

6. Otherwise, TRY ENTER/MONITOR EXIT/TRY OK/TRY ERROR 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.

Now we proceed by induction on n. In the base case, $n = 0$, which is trivial as $\sigma' = \sigma$ and $e' = e$. 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 $\mathit{mutatable}(\sigma_k, e_k, l)$. We clearly have $l \in \mathit{dom}(\sigma_k)$ as no reduction rules remove from memory, thus by the above case for $n = 1$, we have not $\mathit{mutatable}(\sigma', e', l)$ as required. □

Similarly, we use this Stronger Mut Consistency to prove a stronger version of Non-Mutating.

Corollary 3 (Stronger Non-Mutating)**.**
If $\vdash \sigma$, $\sigma;\emptyset \vdash e : T$, $l \in \mathit{dom}(\sigma)$, not $\mathit{mutatable}(\sigma, e, l)$, 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 $\mathit{validState}(\sigma, \mathcal{E}_v[e])$, $l \in \mathit{dom}(\sigma)$, not $\mathit{mutatable}(\sigma, e, l)$, and $\sigma|e \to^* \sigma'|e'$, then not $\mathit{mutatable}(\sigma', e', l)$.

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 `capsule` can be used where an `imm` is expected, and our TASCAPSULE typing rule allows the use of `imm` local variables.

Theorem 6 (Imm–Capsule Consistency)**.**
If $\mathit{validState}(\sigma, e)$, then $\forall l$:
1. if $\mathit{immutable}(\sigma, e, l)$, then not $\mathit{mutatable}(\sigma, e, l)$, and
2. if $e = \mathcal{E}[\texttt{capsule}\ l]$, then $\mathit{encapsulated}(\sigma, \mathcal{E}, l)$

Proof. As before, we prove this by induction on the number of reduction 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 $\sigma|e$.

Part 1 (Imm Consistency): 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*:

1. (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 $l \in \mathit{ROG}(\sigma, l')$.
   - 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 $l \in \mathit{ROG}(\sigma, l')$, and l was not *immutable*, by Immutable ROG we have $\mathit{mutatable}(\sigma, \mathcal{E}_v[\texttt{capsule}\ l'\ \texttt{as}\ \mu'], l)$.
   - By the inductive hypothesis we have $\mathit{encapsulated}(\sigma, \mathcal{E}_v[\square\ \texttt{as}\ \mu'], l')$, and so it follows that not $\mathit{reachable}(\sigma, \mathcal{E}_v, l)$.
   - 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 $\mathcal{E}_v[\mu'\, l']$.
2. (NEW/NEW TRUE) $\sigma'|\mathcal{E}_v[\texttt{new}\ C(\mu_1\, l_1, \ldots, \mu_n\, l_n)] \to \sigma|e$, where $\sigma = \sigma', l_0 \mapsto C\{l_1, \ldots, l_n\}$, $e = \mathcal{E}_v[e']$, and $e' \in \{\texttt{mut}\ l_0, \texttt{M(}l_0\texttt{;}\ \texttt{mut}\ l_0\texttt{;}\ \texttt{read}\ l_0\texttt{.invariant())}\}$

- By Valid Type, $\texttt{new}\, C\texttt{(}\mu_1\, l_1, ..., \mu_n\, l_n\texttt{)}$ was typed by TNew and so we have $\texttt{class}\ C\ \texttt{implements}\ \_\ \texttt{\{}Fs\texttt{;}\_\texttt{\}}$, where $Fs = \kappa_1\ \_\ f_1, ..., \kappa_n\ \_\ f_n$.
- 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 for some $i \in [1, n]$ with $\kappa_i = \texttt{imm}$, we have $l \in ROG(\sigma, \sigma[l_0.f_i]) = ROG(\sigma, l_i)$.
- By Valid Type and TNew, we have that $\mu_i \leq \widetilde{\kappa_i} = \texttt{imm}$.
- Thus, as with the As case above, we have $\mu_i = \texttt{capsule}$ and by Immutable ROG we have that l was *mutatable*, and so by the inductive hypothesis we have the l was previously *reachable*, only through the $\mu_i\, l_i$ argument of the $\texttt{new}$.
- Thus l is not *reachable* through any $\sigma[l_0.f_j]$ 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 $\texttt{imm}$ field, it follows that l is not *mutatable*.

3. (access) $\sigma|\mathcal{E}_v[\mu\, l'\texttt{.}f] \to \sigma|e$, 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 $\texttt{imm}$ fields that are *reachable* through l'.
   - Thus the only way l is now *immutable* is if we just introduced an $\texttt{imm}$ 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 $\texttt{imm}$ field of $\mu\, l$; either way l must have been *immutable*, a contradiction.
4. (update) $\sigma'|\mathcal{E}_v[\mu\, l'\texttt{.}f = \mu'\, l''] \to \sigma|e$, where $\sigma = \sigma'[l'.f = l'']$ and $e = \texttt{M(}l'\texttt{;}\ \texttt{mut}\, l'\texttt{;}\ \texttt{read}\, l'\texttt{.invariant())}$
   - As with the As case above, since l is now *immutable*, we must have that $C.f = \texttt{imm}\ \_\ f$ and $l \in ROG(\sigma, l'')$.
   - **TODO: FUCK we have $l \in ROG(\sigma, l'')$, HOW DO I PROVE that $l \in ROG(\sigma', l'')$? Does that even hold**
5. (call/call mutator) $\sigma|\mathcal{E}_v[\mu_0\, l_0.m\texttt{(}\mu_1\, l_1, ..., \mu_n\, l_n\texttt{)}] \to \sigma|e$, where $e = \mathcal{E}_v[e']$, $e' \in \{e''\ \texttt{as}\ \mu'', \texttt{M(}l_0\texttt{;}\ e''\ \texttt{as}\ \mu''\texttt{;}\ \texttt{read}\, l_0\texttt{.inv}$
   $e'' = e'''[\texttt{this} := \mu_0'\, l_0, x_1 := \mu_1'\, l_1, ..., x_n := \mu_n'\, l_n]$, 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*, since we have not modified memory, it follows that $l \in ROG(\sigma, l_i)$ for some $i \in [1, n]$ with $\mu_i' = \texttt{imm}$.
   - As with the As case above, by Valid Type and TCall, we have that $\mu_i = \texttt{capsule}$, moreover, as l is not *immutable*, we have $l \in MROG(\sigma, l_i)$.
   - 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 $\mathcal{E}_v[e']$.
6. (try enter/try ok/try error/monitor exit) $\sigma|e' \to \sigma|e$

- 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.

Part 2 (Capsule Consistency): Now suppose $e = \mathcal{E}[\mathtt{capsule}\, l]$, for some $\mathcal{E}$, and $encapsulated(\sigma, \mathcal{E}, l)$ doesn't hold.

Thus we pick an $l' \in ROG(\sigma, l)$ with $mutatable(\sigma, \mathcal{E}[\mathtt{capsule}\, l], l')$ such that $reachable(\sigma, \mathcal{E}, l')$.

We now proceed by cases on the reduction rule we just applied, and show a contradiction, thus proving that l is in fact be *encapsulated*:

1. (NEW/NEW TRUE) $\sigma'|\mathcal{E}_v[e''] \rightarrow \sigma|\mathcal{E}[\mathtt{capsule}\, l]$, where $\sigma = \sigma', l_0 \mapsto C\{ls\}$, $\mathcal{E}[\mathtt{capsule}\, l] = \mathcal{E}_v[e']$, $e' \in \{\mathtt{M(}l_0\mathtt{;\,mut}\, l_0\mathtt{;\,read}\, l_0\mathtt{.invariant()),mut}\, l_0\}$, and $e'' = \mathtt{new}\ C\mathtt{(}vs\mathtt{)}$
   - 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 $\mathtt{capsule}$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 $\mathtt{capsule}\, l \in \mathcal{E}_v$ and $e' \in \mathcal{E}$:
     - As we didn't modify $\mathcal{E}_v$, this $\mathtt{capsule}\, l$ must have been in the previous state, i.e. we have some $\mathcal{E}'$ with $\mathcal{E}_v[e''] = \mathcal{E}'[\mathtt{capsule}\, l]$ 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 $mutatable(\sigma', \mathcal{E}'[\mathtt{capsule}\, l], l')$, and since $l' \in ROG(\sigma, l)$, it follows that not $reachable(\sigma', \mathcal{E}', l')$ in the previous state.
     - Suppose l' is *reachable* through $\mathcal{E}_v$, 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 $l'' \in \mathcal{E}_v$, we have $l'' \in \mathcal{E}'$, and so we had $reachable(\sigma, \mathcal{E}', l')$, a contradiction.
     - Otherwise, l' is *reachable* through e', clearly $l' \in dom(\sigma')$, and so by Lost Forever, we have $reachable(\sigma', \mathtt{new}\ C\mathtt{(}vs\mathtt{)}, l')$. But $\mathtt{new}\ C\mathtt{(}vs\mathtt{)} \in \mathcal{E}'$, 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'' or the reduction rule applied, they only require $validState(\mathcal{E}_v[e''])$, $\sigma'|e'' \rightarrow \sigma|e'$, $ROG(\sigma, l) = ROG(\sigma', l)$, and $\mathcal{E}_v[e'] = \mathcal{E}[\mathtt{capsule}\, l]$, were $\mathcal{E}$ is not of form $\mathcal{E}_v[\mathcal{E}'']$.
2. (ACCESS) $\sigma|\mathcal{E}_v[\mu\, l''\mathtt{.}f] \rightarrow \sigma|\mathcal{E}[\mathtt{capsule}\, l]$, where $\mathcal{E}[\mathtt{capsule}\, l] = \mathcal{E}_v[\mu{::}\kappa\, \sigma[l''.f]]$
   - Suppose $\mathcal{E} = \mathcal{E}_v$, so $\mathtt{capsule}\, l = \mu{::}\kappa\, \sigma[l''.f]$:
     - By definition of $\mu{::}\kappa$, this means that $\mu = \mathtt{capsule}$, and so by the inductive hypothesis we have that $encapsulated(\sigma, \mathcal{E}_v[\square\mathtt{.}f], l'')$.
     - Since $l' \in ROG(\sigma, l)$ and $l = \sigma[l''.f]$, it follows that $l' \in ROG(\sigma, l'')$.
     - Since l' is *mutatable* in $\mathcal{E}_v[\mathtt{capsule}\, l]$, by Mut Consistency, l' is also *mutatable* in $\mathcal{E}_v[\mathtt{capsule}\, l''\mathtt{.}f]$
     - Thus, since l'' was *encapsulated* and $l' \in ROG(\sigma, l'')$, it follows that l' is not *reachable* through $\mathcal{E}_v[\square\mathtt{.}f]$.
     - Clearly this means l' is not *reachable* through $\mathcal{E}_v$, a contradiction.
   - Otherwise, $\mathtt{capsule}\, l \in \mathcal{E}_v$, and so by the NEW/NEW TRUE case above, we have a contradiction.
3. (UPDATE) $\sigma'|\mathcal{E}_v[\mu\, l''\mathtt{.}f\mathtt{=}v] \rightarrow \sigma|\mathcal{E}[\mathtt{capsule}\, l]$, where $\sigma = \sigma'[l''.f = v]$ and $\mathcal{E}[\mathtt{capsule}\, l] = \mathcal{E}_v[\mathtt{M(}l''\mathtt{;\,mut}\, l''\mathtt{;\,read}\, l''\mathtt{.invaria}$
   TODO: Can't use $\sigma[l.f = v]$!
   - Clearly $\mathtt{capsule}\, l \in \mathcal{E}_v$, since the update expression reduced to a monitor over a $\mathtt{mut}$, not a $\mathtt{capsule}$.

- As the reduction didn't modify $\mathcal{E}_v$, have $\mathcal{E}_v[\mu\, l''\texttt{.}f\,\texttt{=}\,v] = \mathcal{E}'[\texttt{capsule}\, l]$, for some $\mathcal{E}'$, with $\mu\ l''\texttt{.}f\,\texttt{=}\,v \in \mathcal{E}'$.
- By the inductive hypothesis, we have $\mathit{encapsulated}(\sigma', \mathcal{E}', l)$.
- By Valid Type and our TUpdate 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 $\mathit{encapsulated}(\sigma', \mathcal{E}', l)$ then 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.

4. (CALL/CALL MUTATOR) $\sigma|\mathcal{E}_v[\mu_0\, l_0\texttt{.}m\texttt{(}\mu_1\, l_1, .., \mu_n\, l_n\texttt{)}] \rightarrow \sigma|\mathcal{E}[\texttt{capsule}\, l]$, where $\mathcal{E}[\texttt{capsule}\, l] = \mathcal{E}_v[e']$, $e' \in \{e''\,\texttt{as}\,\mu'', \texttt{M(}l_0\texttt{;}\ e''\,\texttt{as}\,\mu''\texttt{;}\ \texttt{read}\, l_0\texttt{.invariant())}\}$, $e'' = e'''[\texttt{this} := \mu'_0\, l_0, x_1 := \mu'_1\, l_1, .., x_n := \mu'_n\, l_n]$, and $\mathrm{C}^{\sigma}_{l_0} = \mu'_0\,\texttt{method}\,\mu''\ \_\ m\texttt{(}\mu'_1\ \_\ x_1, .., \mu'_n\ \_\ x_n\texttt{)}\ e'''$
   - 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 $\mu'_i\, l_i = \texttt{capsule}\, l$, for some $i \in [0, n]$.
     - Moreover, this means that $e''' = \mathcal{E}'''[\texttt{this}]$, if $i = 0$, otherwise $e''' = \mathcal{E}'''[x_i]$, for some $\mathcal{E}'''$.
     - By Valid Type and our TCall rule, we have $\mu_i eqmdf'_i$, i.e. $\mu_i = \texttt{capsule}$.
     - Suppose $i = 0$:
       * By the inductive hypothesis we thus have $\mathit{encapsulated}(\sigma, \mathcal{E}_v[\square\texttt{.}m\texttt{(}\mu_1\, l_1, .., \mu_n\, l_n\texttt{)}], l)$.
       * 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 $j \neq i$.
       * 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 $e' = e''\,\texttt{as}\,\mu''$. Thus by our well-formedness rules on method bodies, since l' is *reachable* in $\mathcal{E}_v[\mathcal{E}'']$, we must have that l' is only *reachable* through each occurrence of $\texttt{this} \in e'''$, which have all been substituted with $\mu'_i\, l_i$ (since there are no other references, 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 bodies mentions $\texttt{capsule}$ receivers at most once, it follows that $\texttt{this} \notin \mathcal{E}'''$.
       * Since $\mathcal{E}' = \mathcal{E}'''[\texttt{this} := \mu'_0\, l_0, x_1 := \mu'_1\, l_1, .., x_n := \mu'_n\, l_n]\,\texttt{as}\,\mu''$, it follows that l' is not *reachable* through $\mathcal{E}'$.
     - Otherwise, $i \geq 1$:
       * By the inductive hypothesis, we have $\mathit{encapsulated}(\sigma, \mathcal{E}_v[\mu_0\, l_0\texttt{.}m\texttt{(}\mu_1\, l_1 .. \mu_{i-1}\, l_{i-1}, \square, \mu_{i+1}\, l_{i+1} .. \mu_n\, l_n\texttt{)}], l)$.
       * As in the $i = 0$ case, it follows from Mut Consistency and the definition of *encapsulated* that l' is not *reachable* through $\mathcal{E}_v$,or any $\mu_j\, l_j$ with $j \neq i$. Furthermore, by our well-formedness rules on method bodies, it follows that l' is only *reachable* through each occurrence of $x_i \in e'''$, which have all been substituted with $\mu'_i\, l_i$.
       * Since our type system requires that each method bodies mentions each $\texttt{capsule}$ parameter at most once, it follows that $x_i \notin \mathcal{E}'''$.
       * Since l' is not *reachable* through l_0, and $\mathcal{E}' \in \{\mathcal{E}''''\,\texttt{as}\,\mu'', \texttt{M(}l\texttt{;}\ \mathcal{E}''''\,\texttt{as}\,\mu''\texttt{;}\ \texttt{read}\, l_0\texttt{.invariant())}\}$, where $\mathcal{E}'''' = \mathcal{E}'''[\texttt{this} := \mu'_0\, l_0, x_1 := \mu'_1\, l_1, .., x_n := \mu'_n\, l_n]$, it follows that l' is not *reachable* through $\mathcal{E}'$.
     - Either way, as l' is not *reachable* through $\mathcal{E}'$, and it was not *reachable* through $\mathcal{E}_v$ either, 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.

5. (TRY ENTER/TRY OK/TRY ERROR/MONITOR EXIT) $\sigma|e' \to \sigma|\mathcal{E}[\texttt{capsule}\,l]$
   - These rules do not modify memory, introduce references in the main expression, or change their reference capabilities. Thus it follows that $e' = \mathcal{E}'[\texttt{capsule}\,l]$, for some $\mathcal{E}'$.
   - Thus, from the inductive hypothesis, we have *encapsulated*$(\sigma, \mathcal{E}', l)$, moreover, by Mut Consistency, we have that *mutatable*(σ, e', l'), and so it follows that l' is not *reachable* in $\mathcal{E}'$.
   - But these reduction rules do not introduce any references, nor duplicate them, and since memory hasn't been modified, as l' is *reachable* in $\mathcal{E}$, it follows that l' is *reachable* in $\mathcal{E}'$, a contradiction.

6. (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 $\mu'\, l'' = \texttt{capsule}\,l$.
     - 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 sequence such that $\sigma_0|e_0 \to^* \sigma|\mu\, l\,\texttt{as capsule}$. Thus, $\sigma_0|e_0\,\texttt{as capsule}$ is the state our $\mu\, l\,\texttt{as capsule}$ expression was in before the body of the $\texttt{as capsule}$ expression 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, we will show that $\sigma_0; \emptyset \vdash \widehat{e_0} : \texttt{mut}\,C$ holds for some C:
       * thus there is some σ_0', m, and $l_0..l_n$, where $\sigma_0'|\mu_0\, l_0.m(\_l_1, ..., \_l_n) \to \sigma_0'|\mathcal{E}[e_0\,\texttt{as capsule}] \to^* \sigma_0|\mathcal{E}_v{}'[e_0\,\texttt{as capsule}]$, where $\mathcal{E} = \mathcal{E}_v{}''[\mathcal{E}_v{}']$.
       * 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 $\mathcal{E}'$ with $C_0.m = \_\texttt{method}\,\_\,m(\mu_1\, C_1\, x_1, ..., \mu_n\, C_n\, x_n)$ where $e_0'[x_0 \coloneqq \mu_0\, l_0 .. x_n \coloneqq \mu_n\, l_n] = e_0$.
       * By our well-formedness rules on method bodies, Nested Type and Type Rule, 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, and where the typing rule used was either TAS or TASCAPSULE.
       * Suppose the typing rule applied was TAS, then we have $\emptyset; \Gamma \vdash e_0' : \texttt{capsule}\,C$. So by Valid Type, Method Type, and Substitution we have $\emptyset; \Gamma \vdash e_0 : \texttt{capsule}\,C$, thus as above, by Type Preservation? we have $\mu = \texttt{capsule}$, and by the inductive hypothesis, we have that l is *encapsulated*, a contradiction.
       * Thus the typing rule must have been TASCAPSULE, and since our well-formedness rules on method bodies ensure $c \notin e_0'$, we have $\emptyset; \widehat{\Gamma} \vdash e_0' : \texttt{mut}\,C$.
       * 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'.
       * Consider each $i \in [0, n]$, we have $\widehat{\Gamma}(x_i) = \widehat{\mu_i}\, C_i$, by Valid Type and Method Type we have $\mathrm{C}^{\sigma_0'}_{l_i} \leq C_i$.
       * Thus by Substitution we have $\sigma; \emptyset \vdash \widehat{e_0} : \texttt{mut}\,C$.
     - Now, we show that for all $\texttt{mut}\,l' \in e_0$ and $l'' \in \mathit{ROG}(\sigma_0, l')$, we have $\sigma(l'') = \sigma_0(l'')$ and $l'' \notin \mathit{MROG}(\sigma, l)$:
       * Suppose *mutatable*$(\sigma_0, \widehat{e_0}, l'')$, then since $\widehat{e_0}$ contains no $\texttt{mut}$ references, it follows that $e_0 = \mathcal{E}[\texttt{capsule}\,l''']$ for some $\mathcal{E}$ and l''' with $l'' \in \mathit{MROG}(\sigma_0, l''')$.
       * By the inductive hypothesis, we have *encapsulated*$(\sigma_0, \mathcal{E}, l''')$. Since l'' is clearly *mutatable* in $\mathcal{E}$, it follows that l'' is not *reachable* in $\mathcal{E}$.
       * But $\texttt{mut}\,l' \in \mathcal{E}$, and l'' is *reachable* through l', a contradiction. Thus not *mutatable*$(\sigma_0, \widehat{e_0}, l'')$.
       * Clearly $e_0 \sim \widehat{e_0}$, and since $\sigma_0|e_0 \to^* \sigma|\mu\, l$, by Bisimulation, there is some μ' such that $\sigma_0|\widehat{e_0} \to^* \sigma|\mu'\, l$.
       * Then, since not *mutatable*$(\sigma_0, \widehat{e_0}, l'')$, by Stronger Non-Mutating we have $\sigma(l'') = \sigma_0(l'')$.

    - ∗ Suppose $l'' \in \mathit{MROG}(\sigma, l)$, then $\sigma_0; \emptyset \vdash \widehat{e_0} : \mathtt{mut}\, C$, by Type Preservation and Type Rule, it follows that $\mu' \leq \mathtt{mut}$ and hence $\mathit{mutatable}(\sigma, \mu'\, l, l'')$.
    - ∗ But $\sigma_0|\widehat{e_0} \rightarrow^* \sigma|\mu'\, l$ and not $\mathit{mutatable}(\sigma_0, \widehat{e_0}, l'')$, so by Stronger Mut Consistency we have not $\mathit{mutatable}(\sigma, \mu'\, l, l'')$, a contradiction.
- – We also show that for all l'', if $\mathit{reachable}(\sigma_0, \mathcal{E}_v, l'')$, then $\sigma_0(l'') = \sigma(l'')$:
    - ∗ If l'' is in the ROG of some $\mathtt{mut}\, l''' \in e_0$, then this holds by the above.
    - ∗ Otherwise, by Non Mutating, we must have that l'' is in the MROG of some $\mathtt{capsule}\, l''' \in e_0$, i.e. $e_0 = \mathcal{E}'[\mathtt{capsule}\, l''']$, for some $\mathcal{E}'$.
    - ∗ By the inductive hypothesis, we have that $\mathit{encapsulated}(\sigma_0, \mathcal{E}_v[\mathcal{E}'], l''')$, since l'' is *mutatable* through l''', it follows that we can't have $\mathit{reachable}(\sigma_0, \mathcal{E}_v[\mathcal{E}'], l'')$, a contradiction.
- – Since $\mathit{reachable}(\sigma, \mathcal{E}_v, l')$, since nothing *reachable* from $\mathcal{E}_v$ has been modified, it follows that we must have initially had $\mathit{reachable}(\sigma_0, \mathcal{E}_v, l')$, and so $l' \in \mathit{dom}(\sigma_0)$.
- – Since $\mathit{mutatable}(\sigma, \mathcal{E}_v[\mathtt{capsule}\, l], l')$ and $l' \in \mathit{dom}(\sigma_0)$, by Mut Consistency, we have $\mathit{mutatable}(\sigma_0, \mathcal{E}_v[e_0], l')$.
- – 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 $l' \in \mathit{ROG}(\sigma_0, l'')$.
- – Now consider the possible values of μ':
    - ∗ If $\mu' = \mathtt{capsule}$:
        - · By the inductive hypothesis, we have $\mathit{encapsulated}(\sigma_0, \mathcal{E}_v[\mathcal{E}'], l'')$, where $\mathcal{E}'[\mathtt{capsule}\, l''] = e_0$.
        - · Since $\mathit{mutatable}(\sigma_0, \mathcal{E}_v[e_0], l')$, it thus follows that not $\mathit{reachable}(\sigma_0, \mathcal{E}_v[\mathcal{E}'], l')$, and hence not $\mathit{reachable}(\sigma_0, \mathcal{E}_v, l')$.
        - · **TODO: THERE IS A MISSING STEP HERE**
        - · By the above, it follows that we can't have mutated anything *reachable* from $\mathcal{E}_v$, thus we can't have made $\mathit{reachable}(\sigma, \mathcal{E}_v, l')$ hold, a contradiction.
    - ∗ If $\mu' = \mathtt{mut}$:
        - · By the above, we have $l' \notin \mathit{MROG}(\sigma, l)$, since $l' \in \mathit{ROG}(\sigma, l)$, it follows that l' must be in the ROG of an `imm` field (since we do not have `read` fields).
        - · Thus $\mathit{immutable}(\sigma, \mathcal{E}_v[\mathtt{capsule}\, l], l')$, and by the Imm Consistency part of the proof above, we have not $\mathit{mutatable}(\sigma, \mathcal{E}_v[\mathtt{capsule}\, l], l')$, a contradiction.
    - ∗ If $\mu' = \mathtt{imm}$:
        - · Thus we have $\mathit{immutable}(\sigma_0, \mathcal{E}_v[e_0\, \mathtt{as\, capsule}], l')$, and so by the inductive hypothesis we have not $\mathit{mutatable}(\sigma_0, \mathcal{E}_v[e_0\, \mathtt{as\, capsule}], l')$
        - · Since $\sigma_0|\mathcal{E}_v[e_0\, \mathtt{as\, capsule}] \rightarrow^* \sigma|\mathcal{E}_v[\mathtt{capsule}\, l]$, by Mut Consistency, we have not $\mathit{mutatable}(\sigma, \mathcal{E}_v[\mathtt{capsul}$
          a contradiction.
    - ∗ Otherwise, $\mu' = \mathtt{read}$:
        - · If l' is in the ROG of any non-`read` reference in e_0, then one of the above cases would apply, and we would have a contradiction.
        - · If l' was in the ROG of any `imm` field in the ROG of l'', then $\mathit{immutable}(\sigma_0, \mathcal{E}_v[e_0\, \mathtt{as\, capsule}], l')$ would hold, and by the case for $\mu' = \mathtt{imm}$ above, we have a contradiction.
        - · Thus, we assume that l' is only *reachable* through `read` references in e_0, but not through any `imm` fields.
        - · Consider the reduction sequence $\sigma_0|e_0 \rightarrow^* \sigma|\mu\, l\, \mathtt{as\, capsule}$: by Valid Type and our typing rules, it follows that a `read` reference cannot change reference capabilities (because our TAs, TAsCapsule, and TCall rules prohibit this), `read` reference can 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).
        - · However, we just assumed that l' isn't in the ROG of an `imm` field, so if a field access on a `read` reference returns an `imm`, then l' is not *reachable* through the result of said access (by the Access rule).

  - Thus we have that at each step of reduction: either l' is not *reachable* in the body of the as expression, or it is *reachable* only through `read` references.
  - But by Valid Type and our TAs and TAsCapsule rules, we have that $\mu \notin \{$`read`, `imm`$\}$, hence l' cannot be *reachable* through $\mu\, l$.
  - But we assumed that $l' \in ROG(\sigma, l)$, a contradiction.
- Otherwise, `capsule` $l \in \mathcal{E}_v$, and so by the NEW/NEW TRUE case above, we have a contradiction. □

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

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

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

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

Proof.

- By definition of *validState* and our well-formedness rules on method bodies, we must have some $\mathcal{E}_v$, e_0, and e'_0 with $validState(\sigma, \mathcal{E}_v[$`try {`$e_0$`} catch {`$e'_0$`}`$])$ and $\sigma|$`try {`e_0`} catch {`e'_0`}` $\to \sigma|$`try`$^{\sigma}$`{`e_0`} catch {`e'_0`}` $\to^* \sigma' \mid$ `try`$^{\sigma}$`{`e`} catch {`e'`}`.

- By our grammar for $\mathcal{E}_v$ and our reduction rules we also have $\sigma|e_0 \to^* \sigma'|e$ and $e'_0 = e'$.

- By Valid State and our typing rules we have that the TTryCatch1 rule applied, and hence $\sigma; \emptyset \vdash$ `try {`e`} catch {`e'`}` $: T$, $\sigma; \emptyset \vdash e_0 : T$, and $\sigma; \emptyset \vdash e' : T$, for some T.

- By definition of *validState* and our reduction rules we must have had that the `try {`e_0`} catch {`e'_0`}` expression was introduced by a method call. We will show that $\sigma; \emptyset \vdash \widehat{e_0} : T'$ holds for some T':

  – Thus there is some σ'_0, m, and $l_0..l_n$, where $\sigma''|\mu_0\, l_0$`.`m`(`$\_l_1, .., \_l_n$`)` $\to \sigma''|\mathcal{E}[$`try {`$e_0$`} catch {`$e'_0$`}`$] \to^* \sigma|\mathcal{E}_v{}'[$`try {`$e_0$`} catch {`$e'_0$`}`$]$ where $\mathcal{E}_v = \mathcal{E}_v{}''[\mathcal{E}_v{}']$ for some $\mathcal{E}_v{}''$.

  – Let $x_0 =$ `this` and $C_0 = \mathrm{C}^{\sigma''}_{l_0}$, then by our CALL/CALL MUTATOR rules we have some e_1, e'_1, and $\mathcal{E}'$ with $C_0.m =$ `_method _`m`(`$\mu_1\, C_1\, x_1, .., \mu_n\, C_n\, x_n$`)` $\mathcal{E}'[$`try {`e_1`} catch {`e'_1`}`$]]$ where $e'_1[x_0 := \mu_0\, l_0..x_n := \mu_n\, l_n] = e_1$.

  – By our well-formedness rules on method bodies and Nested Type we have that $\sigma; \Gamma \vdash$ `try {`e_1`} catch {`e'_1`}` $:$ T' holds by TTryCatch1, for some T'.

  – By our well-formedness rules on method bodies, $c \notin e_1$, thus we have we have $\sigma; \widehat{\Gamma} \vdash e_1 : T'$.

  – As with the AS case for the Capsule Consistency part of the Imm–Capsule Consistency proof above, we have $e_1[x_0 := \widehat{\mu_0}\, l_0, .., x_n := \widehat{\mu_n}\, l_n] = \widehat{e_0}$ where for each $i \in [0, n]$ we have $\widehat{\Gamma}(x_i) = \widehat{\mu_i}\, C_i$, and by Valid Type and Method Type, we we have $\mathrm{C}^{\sigma'_0}_{l_i} \leq C_i$.

  – Thus by Substitution we have $\sigma; \emptyset \vdash \widehat{e_0} : T'$.

- Now let $l \in dom(\sigma)$ with $reachable(\sigma, \mathcal{E}_v[e'], l)$.

- If we don't have $\mathit{reachable}(\sigma, e_0, l)$ then by Lost Forever, the reduction $\sigma|e_0 \to^* \sigma'|e$ cannot involve a UPDATE on l, i.e. we must have $\sigma'(l) = \sigma(l)$.
- Suppose l is *mutatable* through a **`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 $\mathit{mutatable}(\sigma, \mathcal{E}_v[\texttt{try}^{\sigma}\texttt{\{}e_0\texttt{\}}\ \texttt{catch}\ \texttt{\{}e'_0\texttt{\}}], l)$, and since $\mathit{validState}(\sigma, \mathcal{E}_v[\texttt{try}^{\sigma}\texttt{\{}e_0\texttt{\}}\ \texttt{catch}\ \texttt{\{}e'_0\texttt{\}}])$, by Capsule Consistency we have not $\mathit{reachable}(\sigma, \mathcal{E}_v[\texttt{try}^{\sigma}\texttt{\{}\mathcal{E}\texttt{\}}\ \texttt{catch}\ \texttt{\{}e'_0\texttt{\}}], l)$.
  - But this implies not $\mathit{reachable}(\sigma, \mathcal{E}_v, l)$ and since $e'_0 = e'$, not $\mathit{reachable}(\sigma, e', l)$. Thus we have not $\mathit{reachable}(\sigma, \mathcal{E}_v[e'_0])$, a contradiction.
- By the above, l is not *mutatable* through any **`capsule`** reference in $\widehat{e_0}$ either, as such a reference would be in e_0 as well.
- Since $\widehat{e_0}$ has no **`mut`** references, it follows that not $\mathit{mutatable}(\sigma, \widehat{e_0}, l)$.
- Clearly $e_0 \sim \widehat{e_0}$, and since $\widehat{e_0}$ $\sigma|e_0 \to^* \sigma'|e$, by Bisimulation, there is some e'' such that $\sigma|\widehat{e_0} \to^* \sigma'|e''$.
- 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. $\square$