Using Capabilities for Strict Runtime Invariant Checking

Isaac Oscar Gariano, Marco Servetto*, Alex Potanin

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

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

1. Formal Language Model

[Isaac: REMBER TO TALK ABOUT READ FIELDS WITH MARCO!!!!] [Isaac: Note i'm using "$\square$" instead of "[]" for the "hole"s as it's more standard] 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 **??** the requirements we make on such a type system, so that any language satisfying them can soundly support our invariant protocol. In **??** 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 Pierce [?] and Featherweight Java [?], which is a turing complete minimalistic subset of Java. [Isaac: check this is in the citation]. [Isaac: What's the relavence of "Pierce" here?] 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 $C\texttt{(}\kappa_1\, C_1\, x_1, \ldots, \kappa_n\, C_n\, x_n\texttt{)\{}\textbf{this}.f_1\texttt{=}x_1\texttt{;}\ldots\texttt{;}\textbf{this}.f_n\texttt{=}x_n\texttt{\}}$, where the fields of C are $\kappa_1\, C_1\, f_1, \ldots, \kappa_n\, C_n\, f_n$, and $\widetilde{\kappa_i}$ gives the reference capability required for the field kind κ_i, see below for the definition. Note that we do not model variable updates or traditional subclassing, since this would make the proofs more involved without adding any additional insight.

Notational Conventions

We use the following notational conventions:

- Class, method, parameter, and field names are denoted by C, m, x, and f, respectively.
- We use "vs" as a metavariable denoting a sequence of form $v_1, \ldots, v_n$, and "$v?$" to denote an optional v, similarly with other metavariables.
- We use "_" to stand for any piece of syntax, including the empty string.
- Memory locations are denoted by l.

*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)

- We assume an implicit program/class table; we use the notation $C.m$ to get the method declaration for m within class C, similarly we use $C.f$ to get the declaration of field f, and $C.i$ to get the declaration of the i^{th} field.
- Memory, denoted by $\sigma : l \to C\{vs\}$, is a finite map from locations, l, to annotated tuples, $C\{vs\}$, representing objects; here C is the class name and vs are the field values, without any reference capabilities. We use the notation C^{σ}_{l} get the class name of l, $\sigma[l.f = v]$ 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 $\Gamma : x \to T$.

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 '`imm true`', and
- all other methods in the `Cap` class must require a `mut` receiver; such methods can be declared with the same signature multiple times, thus a call to them will non-deterministically chose one of the implementations.

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.

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

Grammar

The grammar is defined in Figure 1.

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

We use v to represent a raw value, either a location or a boolean. We use w, of form $\mu\, v$, 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. For simplicity, the boolean literals `true` and `false` have a reference capability, but these imposes no constraints (e.g. you can have two "`capsule true`"s simultaneously).

Our expressions (e), include variables (x), boolean literals (μ `true` and μ `false`), object creations (`new` C`(`es`)`), field accesses (`this`$.f$ and $\mu\, l.f$), field updates (`this`$.f$ `=` e and $\mu\, l.f$ `=` e), and method calls $e.m$`(`es`)`. Note that these are sufficient to model standard constructs like sequencing and let-expressions [Isaac: give a citation]. 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:

e	$::=$	$x \mid \mu\,\texttt{true} \mid \mu\,\texttt{false} \mid \texttt{new}\ C\texttt{(}es\texttt{)} \mid \texttt{this.}f \mid \texttt{this.}f\texttt{=}e \mid e.m\texttt{(}es\texttt{)}$	expression
		$\mid \mu\ \texttt{promote}\ \texttt{\{}e\texttt{\}} \mid \texttt{try}\ \texttt{\{}e\texttt{\}}\ \texttt{catch}\ \texttt{\{}e'\texttt{\}}$	
		$\mid \mu\,l \mid \mu\,l.f \mid \mu\,l.f\texttt{=}e \mid \texttt{try}^{\sigma}\texttt{\{}e\texttt{\}}\ \texttt{catch}\ \texttt{\{}e'\texttt{\}} \mid \texttt{M(}l\texttt{;}\ e\texttt{;}\ e'\texttt{)}$	runtime expr.
v	$::=$	$l \mid \texttt{true} \mid \texttt{false}$	raw value
w	$::=$	$\mu\,v$	value
$\mathcal{E}_v$	$::=$	$\square \mid \texttt{new}\ C\texttt{(}ws, \mathcal{E}_v, es\texttt{)} \mid \mu\,l.f\texttt{=}\mathcal{E}_v \mid \mathcal{E}_v.m\texttt{(}es\texttt{)} \mid w.m\texttt{(}ws, \mathcal{E}_v, es\texttt{)}$	eval. context
		$\mid \mu\ \texttt{promote}\ \texttt{\{}\mathcal{E}_v\texttt{\}} \mid \texttt{try}^{\sigma}\texttt{\{}\mathcal{E}_v\texttt{\}}\ \texttt{catch}\ \texttt{\{}e\texttt{\}} \mid \texttt{M(}l\texttt{;}\ \mathcal{E}_v\texttt{;}\ e\texttt{)} \mid \texttt{M(}l\texttt{;}\ w\texttt{;}\ \mathcal{E}_v\texttt{)}$	
$\mathcal{E}$	$::=$	$\square \mid \texttt{new}\ C\texttt{(}es, \mathcal{E}, es'\texttt{)} \mid \mathcal{E}.f \mid \mathcal{E}.f\texttt{=}e \mid e.f\texttt{=}\mathcal{E} \mid \mathcal{E}.m\texttt{(}es\texttt{)}$	full context
		$\mid e.m\texttt{(}es, \mathcal{E}, es'\texttt{)} \mid \mu\ \texttt{promote}\ \texttt{\{}\mathcal{E}\texttt{\}} \mid \texttt{try}^{\sigma?}\texttt{\{}\mathcal{E}\texttt{\}}\ \texttt{catch}\ \texttt{\{}e\texttt{\}}$	
		$\mid \texttt{try}^{\sigma?}\texttt{\{}e\texttt{\}}\ \texttt{catch}\ \texttt{\{}\mathcal{E}\texttt{\}} \mid \texttt{M(}l\texttt{;}\ \mathcal{E}\texttt{;}\ e\texttt{)} \mid \texttt{M(}l\texttt{;}\ e\texttt{;}\ \mathcal{E}\texttt{)}$	
CD	$::=$	$\texttt{class}\ C\ \texttt{implements}\ Cs\ \texttt{\{}Fs\texttt{;}Ms\texttt{\}} \mid \texttt{interface}\ C\ \texttt{implements}\ Cs\ \texttt{\{}As\texttt{\}}$	class decl.
F	$::=$	$\mu\,C\,f$	field
A	$::=$	$\mu\,\texttt{method}\,T\,m\texttt{(}Ps\texttt{)}$	abs. method
M	$::=$	$\mu\,\texttt{method}\,T\,m\texttt{(}Ps\texttt{)}\ e$	method
P	$::=$	$T\,x$	parameter
T	$::=$	$\mu\,C$	type
μ	$::=$	$\texttt{mut} \mid \texttt{imm} \mid \texttt{read} \mid \texttt{capsule}$	reference capability
κ	$::=$	$\texttt{mut} \mid \texttt{imm} \mid \texttt{read} \mid \texttt{rep}$	field kind
$\mathcal{E}_r$	$::=$	$\mathcal{E}_v[\texttt{new}\ C\texttt{(}ws, \square, ws'\texttt{)}] \mid \mathcal{E}_v[\square.f] \mid \mathcal{E}_v[\square.f\texttt{=}v] \mid \mathcal{E}_v[v.f\texttt{=}\square]$	redex context
		$\mid \mathcal{E}_v[\square.m\texttt{(}ws\texttt{)}] \mid \mathcal{E}_v[w.m\texttt{(}ws, \square, ws'\texttt{)}] \mid \mathcal{E}_v[\mu\ \texttt{promote}\ \texttt{\{}\square\texttt{\}}]$	
error	$::=$	$\mathcal{E}_v[\texttt{M(}l\texttt{;}\ w\texttt{;}\ \texttt{imm}\,\texttt{false)}]$, where $\mathcal{E}_v$ not of form $\mathcal{E}_v{}'[\texttt{try}^{\sigma?}\texttt{\{}\mathcal{E}_v{}''\texttt{\}}\ \texttt{catch}\ \texttt{\{\_\}}]$	validation error

Figure 1: Grammar

- Promote expressions ($\mu\ \texttt{promote}\ \texttt{\{}e\texttt{\}}$), these evaluate e and change the reference capability of the result to μ. This is similar to Pony's promote expressions. It is up to the typesystem what promotions it accepts as valid, provided they cannot be used to violate our requirements in ??. Using an explicit promotion makes the proofs much simpler, but a typesystem could simply wrap expressions with a $\mu\ \texttt{promote}\ \texttt{\{\_\}}$ when necessary.

- 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 $\texttt{imm}\,\texttt{true}$ (i.e. l is valid), then the whole monitor expression will return the value fo e, otherwise if it evaluates to $\texttt{imm}\,\texttt{false}$, the monitor expression is an *error*, and evaluation will proceed with to nearest enclosing `catch` block, if any.

- Standard try-catch expressions ($\texttt{try}\ \texttt{\{}e\texttt{\}}\ \texttt{catch}\ \texttt{\{}e'\texttt{\}}$) evaluate e, and if successful return its result, otherwise if e is an *error*, e' will be evaluated and returned. This differs from Java's try-catch *statements* as they do not return a value; our expression-based form can be emulated by using "$T\ x$`; try {`x` = `e`;} catch(Throwable `t`) {`x` = `e'`;}`", for an appropriate type T and fresh variable names t and x, x can then be used in place of the `try-catch` expression. [Isaac: alternatively, put the `try-catch` in a method so you can `return` e?] During reduction, `try-catch` expressions will be annotated as $\texttt{try}^{\sigma}\texttt{\{}e\texttt{\}}\ \texttt{catch}\ \texttt{\{}e'\texttt{\}}$, 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`.

An *error* represents an uncaught invariant failure, i.e. a runtime-injected invariant check that has failed and is not enclosed in a `try` block; this ensures that a `try` block will only contain an *error* if there is no inner `try-catch` that should catch it instead.

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

We provide several expression contexts [Isaac: give a citation that describes them], $\mathcal{E}$, $\mathcal{E}_v$, and $\mathcal{E}_r$. As is standard, an $\mathcal{E}$ represents an expression with a single *hole* ($\square$) in place of a sub-expression. We use the notation $\mathcal{E}[e]$ to fill in the hole, i.e. $\mathcal{E}[e]$ returns $\mathcal{E}$ but with the single occurrence of '$\square$ replaced by e. For example, if $\mathcal{E} = \square.m\texttt{()}$ then $\mathcal{E}[\texttt{new}\ C\texttt{()}] = \texttt{new}\ C\texttt{()}.m\texttt{()}$.

An evaluation context, $\mathcal{E}_v$, represents the standard left-to-right evaluation order, an $\mathcal{E}_v$ is like an $\mathcal{E}$, but all the expression to the left of the hole will be fully evaluated. This is used to model the standard left to right evaluation order: the hole denotes the location of the next expression to be evaluated.

The context $\mathcal{E}_r$ instead has a hole in an argument to a *redex* (i.e. an expression that is about to be reduced). Thus $\mathcal{E}_r[w]$ represents an expression whose very next reduction step involves the value w.

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

Reference Capability Operations

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

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

  [Isaac: is this right? Or should a $\texttt{capsule}::\texttt{rep}$ and $\texttt{capsule}::\texttt{mut}$ return $\texttt{capsule}$?]

The $\le$ notation and $\widetilde{\kappa}$ notations are used latter in **??** and **??**.

Well-Formedness Criteria

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

- $\texttt{invariant()}$ methods must follow the requirements of Section **??**, except that for simplicity method calls on $\texttt{this}$ are not allowed.[1]. This means that for every non-interface class C, $C\texttt{.invariant} = \texttt{read method imm Bool invariant()}\ e$, where e can only use $\texttt{this}$ as the receiver o a $\texttt{imm}$ or $\texttt{rep}$ field access. Formally, this means that if forall $\mathcal{E}$ where $e = \mathcal{E}[\texttt{this}]$, we have:
  - $\mathcal{E} = \mathcal{E}'[\square.f]$, for some $\mathcal{E}'$

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

  - $C.f = \kappa\ \_\ f$
  - $\kappa \in \{\texttt{imm}, \texttt{rep}\}$

- Rep mutators must also follow the requirements in **??**, except that a `mut` method that reads a `rep` field is *always* considered a rep mutator, even if it only needs to use the field value as `read`[2] Such methods must 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 are type checked against their declared return type, under the assumption that their parameters and receiver have the appropriate type' we also require that they do not contain runtime expressions. Formally, for all C_0 and m with $C_0.m = \mu_0\, \texttt{method}\, T\, m\texttt{(}\mu_1\, C_1\, x_1, ..., \mu_n\, C_n\, x_n\texttt{)}\ e$, we have:
  - $\emptyset; \texttt{this} \mapsto \mu_0\, C_0, x_1 \mapsto \mu_1\, C_1, ..., x_n \mapsto \mu_n\, C_n \vdash e : T$
  - e contains no l, $\texttt{M(}\_;\ \_;\ \_\texttt{)}$, 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 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 $\texttt{class}\ C\ \texttt{implements}\ \_\ \texttt{\{}\_; Ms\texttt{\}}$, and $\texttt{interface}\ C'\ \texttt{implements}\ \_\ \texttt{\{}As\texttt{\}}$, where C' is reachable through the `implements` clauses starting from C, then for all $\mu\, \texttt{method}\, m\, T\texttt{(}Ps\texttt{)} \in As$, there is some e with $\mu\, \texttt{method}\, m\, T\texttt{(}Ps\texttt{)}\ e \in Ms$.

Reduction Rules

Our reduction rules are defined in Figure 2. [Isaac: Note: I've decided to try and not make substiutition behave specially for promote expressions as it breaks part of the invariant protocol proof] The rules use $\mathcal{E}_v$ to ensure that the sub-expression to be reduced is the left-most unevaluated one:

- NEW creates a new object, and returns a monitor expression that will check the new object's invariant, and if that succeeds, return a `mut` reference to the object. Note the use of $\sigma, l \mapsto C\{\_\}$ implies that $l \notin dom(\sigma)$, since $\sigma, l \mapsto C\{\_\}$ would be undefined otherwise.

- ACCESS looks up the value of a field in the memory and returns it, annotated with the appropriate reference capability (see above for the definition of μ::κ).

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

[2]This restriction is merely for simplicity, as you could simply write a getter of form $\texttt{read method read}\, C\, m\texttt{()}\ \texttt{this}.f$, where $C.f = \texttt{rep}\, C\, f$, and then call the getter on a `mut this`.

(NEW) $\sigma|\mathcal{E}_v[\texttt{new}\ C(\_v_1, ..., \_v_n)] \to \sigma, l \mapsto C\{v_1, ..., v_n\}|\mathcal{E}_v[\texttt{M}(l;\ \texttt{mut}\ l;\ \texttt{read}\ l.\texttt{invariant()})]$

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

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

(CALL) $\sigma|\mathcal{E}_v[\_l.m(\_v_1, ..., \_v_n)] \to \sigma|\mathcal{E}_v[e'[\texttt{this} \coloneqq \mu_0\, l, x_1 \coloneqq \mu_1\, v_1, ..., x_n \coloneqq \mu_n\, v_n]]$, where:
$\mathrm{C}^{\sigma}_{l}.m = \mu_0\ \texttt{method}\ \_\ m(\mu_1\ \_x_1, ..., \mu_n\ \_x_n)\ e$
if $\mu_0 = \texttt{mut}$ and $\exists f$ such that $\mathrm{C}^{\sigma}_{l}.f = \texttt{rep}\ \_\ f$ and $e = \mathcal{E}[\texttt{this}.f]$,
then $e' = \texttt{M}(l;\ e;\ \texttt{read}\ l.\texttt{invariant()})$,
otherwise $e' = e$

(PROMOTE) $\sigma|\mathcal{E}_v[\mu\ \texttt{promote}\ \{\_v\}] \to \sigma|\mathcal{E}_v[\mu\, v]$

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

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

(MONITOR EXIT) $\sigma|\mathcal{E}_v[\texttt{M}(l;\ w;\ \texttt{imm}\,\texttt{true})] \to \sigma|\mathcal{E}_v[w]$

Figure 2: Reduction rules

- CALL 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. In the case of a rep mutator (a `mut` method that accesses a `rep` field), the method body is wrapped 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` can have multiple definitions of the same method, this reduction rule allows for non-determinism, but only if the receiver is of class `Cap` and the method is a `mut` method.

- PROMOTE simply changes the reference capability to the one indicated. Note that our requirements on the type-system, given in **??**, 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 **??**, we require the type system to ensure strogn 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 `imm false`, 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, in particular, our proofs say nothing about such ill-defined operations. 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 **??** 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 we implement them by statically injecting calls to `invariant()` at the end of setters (for `imm` and `rep` fields), factory methods, and capsule mutators; this works as L42 follows the uniform access principle, so it does not have primitive expression forms for field updates and constructors, rather they are uniformly represented as method calls.

Statement of Soundness

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

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

We say that an object is *valid* when calling its `invariant()` method would deterministically produce `imm true` in a finite number of steps, i.e. assuming the typesystem is sound, this means it does not evaluate to `imm` $Kwfalse$, 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 $\sigma|$`read` l`.invariant()` $\Rightarrow^{+} \sigma, \sigma'|$`imm 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:

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

The idea being that the $\mathcal{E}_r$ is like an $\mathcal{E}_v$ but it has a hole where a reference can be, thus $\mathit{trusted}(\mathcal{E}_r, l)$ holds when the very next reduction we are about to perform is $\mu\, l$`.invariant()` or $\mu\, l.f$. As we discuss in our proof of Soundness, any such $\mu\, l.f$ expression came from the body of the `invariant()` method itself, since l can not occur in the *rog* of any of its fields mentioned in the `invariant()` method.[3]

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 $c \mapsto$ `Cap`$\{\}|e_0 \rightarrow^{*} \sigma|e$, for some e_0 such that:

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

By restricting which initial expressions are well-typed, the type-system (such as the one presented in **??**) can ensure the required properties of our reference-capabilities (see **??**); 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. To make the type system and proofs presented in **??** simpler, we require that c can only be referenced as `mut` and that there are no promote expressions. 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 promote 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.

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

This is a very strong statement because $valid(\sigma, l)$ 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. Thus 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.