More Reasoning about Risk and Trust in an Open Word (Appendix)

Sophia Drossopoulou[1], James Noble[2,1], Toby Murray[4], Mark Miller[3]

[1]Imperial College London, [2]Victoria University Wellington, [3]Google Inc, [3]NICTA and UNSW.

1. Introduction

This is the companion appendix to our work *"Reasoning about Risk and Trust in an Open World"*. We give here the full definitions of $\mathcal{F}ocal$, $\mathcal{C}hainmail$, our Hoare logic, prove soundness of our Hoare logic, and then prove that our escrow exchange implementation establishes mutual trust while managing risk.

Status There are lots of rough edges. But what we want to concentrate on at the moment is the introduction of the oracles, $\mathcal{O}$, in order to give meaning to the **obeys** predicate, and to express the soundness of the Hoare logic.

On the other hand, some other (smaller) things are inconsistent, and will be updated once we are certain about the circularity issue. These include

1. How many specifications may a class declare to implement? Currently in some places it is one, it others it is many
2. Constructors – do we allow constructors with bodies, and explicit pre- snf post- conditions in the specifications? We introduced them later in the day, and need to overhaul the oper semantics, $\mathcal{R}each$ etc, or drop them altogether (ie no pre-post conditions in the Hoare logic).
3. The treatment of *unknown* as a possible interpretation and value of an expression, or an assertion. I think this does not pose problems, but I am not sure it is all consistent. Also, unsure whether we need the distinction between *unknown* and *undefined* – bit SD thinks that we can make all this to work - after all similar problems appear in other systems (while the orcale-related issues are novel).
4. How many parameters to a method or constructor? I now propose that we write the syntax with any number thereof, but already ope semantics onwards we require exactly one, and say this simplifies the exposition.
5. Is it `stmt` or `stmts`?

[Copyright notice will appear here once 'preprint' option is removed.]

2. Formal Definition of the language $\mathcal{F}ocal$

2.1 Modules and Classes

$\mathcal{F}ocal$ modules map class identifiers to class descriptions, and specification identifiers to specification descriptions.

Definition 1 (Modules). *We define modules $M \in Module$ as follows*

$$\begin{aligned} \mathit{Module} \ &= \mathit{ClassId} \ \cup \mathit{SpecId} \\ &\longrightarrow \\ &(\ \mathit{ClassDescr} \ \cup \mathit{Specification}\ ) \end{aligned}$$

such that for any any module M, class identifier $\mathtt{C}$, and specification identifier $\mathtt{C}$, we have $M(\mathtt{C}) \in ClassDescr$ or undefined, and $M(\mathtt{C}) \in Specification$ or undefined.

Classes We define the syntax of class definitions below. Class definitions serve two purposes: They describe how a class contributes to the runtime behaviour of a program: for this we use its field and method declarations. They also describe how a class contributes to the semantics of its module: which specifications it claims to obey, how functions and predicates are to be interpreted when applied to its objects[1], and how ghost fields are to be interpreted.[2] Note that ghost fields are interpreted according to the syntax of expressions - Expr - which will be given later.

Note that the language is untyped. Method bodies consist of sequences of statements; these can be field read or field assignments (only allowed if the object is `this` – i.e. as in Smalltalk), conditionals, and method calls. All else can be encoded.

The keyword **private** indicates that a class, say `C`, is private to its module. This means that even though anybody who has access to objects of class `C` may call any methods on these objects, only classes from the same module as `C` may construct objects of class `C`. In terms of the capabilities

[1] TODO: say earlier that functions and predicates are interpreted depending on the class of receiver

[2] TODO: perhaps use model field and not ghost field.

literature, this means that the capability to create new objects of that class is hidden within this module. This form of privacy is enforced through the linking operator defined in Definition 6.

Definition 2 (Classes, Methods, Args)**.** *We define the syntax of modules below.*

ClassDescr	::=	[**private**] **class** *ClassId* **implements** *SpecId**
		{ (**field** *FieldId*)*
		(*methBody*)*
		(*FunDescr*)*
		(*PredDescr*)*
		(**ghost** *GhFldId* = *Expr*)* }
methBody	::=	**method** *m* **(** *ParId** **)**
		{ *Stmts* **; return** *Arg* }
Stmts	::=	*Stmt* | *Stmt* **;** *Stmts*
Stmt	::=	**var** *VarId* **:=** *Rhs*
	|	*VarId* **:=** *Rhs*
	|	**this**.*FieldId* **:=** *Rhs*
	|	**if** *Arg* **then** *Stmts* **else** *Stmts*
	|	**skip**
Rhs	::=	*Arg***.***MethId*(*Arg**) | *Arg*
	|	**new** *ClassId*(*Arg**)
Arg	::=	*ParId* | *VarId* | **this**
	|	**this**.*FieldId*

Note that $\mathcal{F}ocal$ supports a limited form of protection: the syntax supports reading of fields of any object, but restricts each object to being able to modify only its *own* fields.

Lookup We define method lookup function, $\mathcal{M}$ which returns the corresponding method definition given a class and a method identifier, specification lookup function $\mathcal{S}$ which returns the set of specification identifiers which a class is claiming to be implementing, and $\mathcal{P}$ .... and $\mathcal{G}$ ...

Definition 3 (Lookup)**.** *The lookup functions $\mathcal{M}$, $\mathcal{S}$, and $\mathcal{P}$ are defined as follows, where we assume that* `C` *is a class identifier,* `m` *a method identifier, P a predicate identifier, and* `f` *a function identifier :*

$\mathcal{M}(\mathtt{M}, \mathtt{C}, \mathtt{m}) = \mathbf{method}\, m\, (\, p_1, ...p_n)\, \{\, stms;\ \mathbf{return}\ a\}$
iff $M(\mathtt{C}) = p\,\mathbf{class}\ \mathtt{C}\ ....\{\ ...$
$\quad\mathbf{method}\, m\, (\, p_1, ...p_n)\, \{\, stms;\ \mathbf{return}\ a\}$
$\quad ...\}$.
undefined, otherwise.

$\mathcal{S}(M, \mathtt{C})\ =\ \{\ S_1, ...S_n\ \}$
iff $M(\mathtt{C})\ =\ p\,\mathbf{class}\ \mathtt{C}\ \mathbf{implements}\ S_1, ...S_n\{\ ...\}$
undefined, otherwise.

$\mathcal{P}(M, \mathtt{C}, Q)\ =\ \mathbf{predicate}\, Q\, (\, p_1, ...p_n)\, \{\ \ A\ \}$
iff $M(\mathtt{C})\ =\ p\,\mathbf{class}\ \mathtt{C}\ ....\{\ ...$
$\quad\mathbf{predicate}\, Q\, (\, p_1, ...p_n)\, \{\ \ A\ \}$
$\quad ...\}$.
undefined, otherwise.

$\mathcal{G}(M, C, gf)\ = expr$
iff $M(C) =\ p\,\mathbf{class}\ C\ ....\{\ ...$
$\quad\mathbf{ghost}\ gf = expr$
$\quad ...\}$.
undefined, otherwise.

In all the above, p is either empty, or the annotation **private***.*

2.2 Execution in $\mathcal{F}ocal$

In order to define execution in $\mathcal{F}ocal$, we define runtime states, σ, the interpretation of arguments in such states, $\lfloor a \rfloor_\sigma$, and finally a relation of the form $M, \sigma, stms \leadsto \sigma'$.[3].

Runtime state The runtime state σ consists of a stack frame ϕ, and a heap χ. A stack frame is a mapping from receiver (**this**) to its address, and from the local variables ($VarId$) and parameters ($ParId$) to their values. Values are integers, the booleans **true** or **false**, addresses, or **null**. Addresses are ranged over by ι. The heap maps addresses to objects. Objects are tuples consisting of the class of the object, and a mapping from field identifiers onto values.

$\sigma \in$ state	=	frame $\times$ heap
$\phi \in$ frame	=	StackId $\longrightarrow$ val
$\chi \in$ heap	=	addr $\longrightarrow$ object
v $\in$ val	=	{ `null`, **true**, **false** } $\cup$ *addr* $\cup\ \mathbb{N}$
object	=	ClassId $\times$ (FieldId $\longrightarrow$ val)
$\iota, \iota', ..$	$\in$	*addr*
StackId	=	{ **this** } $\cup$ VarId $\cup$ ParId

The Operational Semantics of $\mathcal{F}ocal$ We define $\lfloor a \rfloor_\sigma$, the *interpretation* of an argument $a \in Arg$ in a state σ as follows.

Definition 4 (Interpretation)**.** *For a state $\sigma\ =\ (\phi, \chi)$ we define the partial function*
$\quad\lfloor\_\rfloor_{\_} : state \times Path \rightarrow Value$
as follows:

$$
\begin{array}{lcll}
\lfloor \mathbf{null} \rfloor_\sigma & = & \mathbf{null} & \\
\lfloor \mathbf{true} \rfloor_\sigma & = & \mathbf{true} & \\
\lfloor \mathbf{false} \rfloor_\sigma & = & \mathbf{false} & \\
\lfloor x \rfloor_\sigma & = & \phi(x) & \text{(for x} \in StackId\text{)}, \\
 & & \textit{undefined,} & \textit{otherwise.} \\
\lfloor p.f \rfloor_\sigma & = & \chi(\lfloor p \rfloor_\sigma)(f) & \textit{if} \lfloor p \rfloor_\sigma \textit{ is defined,} \\
 & & & \textit{and f is a field of} \lfloor \mathbf{this} \rfloor_\sigma \\
 & & \textit{undefined,} & \textit{otherwise}
\end{array}
$$

where
$\quad p \in path\quad ::=\quad x\ \mid\ p.f$
We also define the lookup of the class of an expression,
$\quad\mathcal{C}lass(e)_\sigma\ =\ (\sigma \downarrow_2 (\lfloor e \rfloor_\sigma)) \downarrow_1\quad$ *if* $\lfloor e \rfloor_\sigma$ *defined*
$\quad\quad$ *undefined, otherwise*

Execution uses module M, and maps a runtime state σ and statements stmts (respectively a right hand side rhs) onto a new state σ' (respectively a new heap χ' and a value).

[3] TODO: rename private to internal; rename ghost field to model field

(methCall_OS)

$$\frac{\begin{array}{l}\lfloor a\rfloor_{\phi\cdot\chi}=\iota\\ \mathcal{M}(\mathtt{M},\chi(\iota)\downarrow_1,\mathtt{m})\ =\\ \quad\textbf{method}\ m(\,par_1,\dots par_n)\ \{\ stms\textbf{;}\ \textbf{return}\ a'\}\\ \phi''=\textbf{this}\mapsto\iota, par_1\mapsto\lfloor a_1\rfloor_{\phi\cdot\chi},\dots par_n\mapsto\lfloor a_n\rfloor_{\phi\cdot\chi}\\ M,\ \phi''\cdot\chi, stmts\ \leadsto\ \phi'\cdot\chi'\end{array}}{M,\ \phi\cdot\chi,\ a.m(\,a_1,\dots a_n)\ \leadsto\ \chi', \lfloor a'\rfloor_{\phi'\cdot\chi'}}$$

(Arg_OS)

$$\frac{}{M,\ \phi\cdot\chi,\ a\ \leadsto\ \chi, \lfloor a\rfloor_{\phi\cdot\chi}}$$

(New_OS)

$$\frac{\begin{array}{l}\iota\text{ is new in }\chi\\ \mathtt{f}_1,\dots\mathtt{f}_n\text{ are the fields defined in }C\end{array}}{\begin{array}{l}M,\ \phi\cdot\chi,\ \textbf{new}\,C(\,a_1,\dots a_n\textbf{)}\\ \leadsto\ \chi[\iota\mapsto(C,\mathtt{f}_1\mapsto\lfloor a_1\rfloor_{\phi,\sigma}..\mathtt{f}_n\mapsto\lfloor a_n\rfloor_{\phi,\sigma})]\end{array}}$$

(varAsg-1_OS)

$$\frac{M,\ \phi\cdot\chi,\ e\ \leadsto\ \chi',\ val}{M,\ \phi\cdot\chi,\ \textbf{var}\,v{:=}e\ \leadsto\ \phi[v\mapsto val]\cdot\chi'}$$

(varAsg-2_OS)

$$\frac{M,\ \phi\cdot\chi,\ e\ \leadsto\ \chi',\ val}{M,\ \phi\cdot\chi,\ v{:=}e\ \leadsto\ \phi[v\mapsto val]\cdot\chi'}$$

(fieldAsg_OS)

$$\frac{M,\ \phi\cdot\chi,\ e\ \leadsto\ \phi\cdot\chi',\ val}{M,\ \phi\cdot\chi,\ \textbf{this}.f{:=}\ e\ \leadsto\ \phi\cdot\chi'[\phi(\textbf{this}), f\mapsto val]}$$

(sequence_OS)

$$\frac{\begin{array}{l}M,\ \sigma, stmt\ \leadsto\ \sigma''\\ M,\ \sigma'',\ stmts\ \leadsto\ \sigma'\end{array}}{M,\ \sigma,\ stmt\textbf{;}\ stmts\ \leadsto\ \sigma'}$$

(cond-True_OS)

$$\frac{\begin{array}{l}\lfloor a\rfloor_{\sigma}=\textbf{true}\\ M,\ \sigma,\ stmts_1\ \leadsto\ \sigma'\end{array}}{M,\ \sigma,\ \textbf{if}\ a\ \textbf{then}\ stmts_1\ \textbf{else}\ stmts_2\ \leadsto\ \sigma'}$$

(cond-False_OS)

$$\frac{\begin{array}{l}\lfloor a\rfloor_{\sigma}=\textbf{false}\\ M,\ \sigma,\ stmts_2\ \leadsto\ \sigma'\end{array}}{M,\ \sigma,\ \textbf{if}\ a\ \textbf{then}\ stmts_1\ \textbf{else}\ stmts_2\ \leadsto\ \sigma'}$$

(skip_OS)

$$\frac{}{M,\ \sigma,\ \textbf{skip}\ \leadsto\ \sigma}$$

Figure 1. Operational Semantics

We therefore do not give execution rules for things like null-pointer-exception, or stuck execution. This allows us to keep the system simple; it will be easy to extend the semantics to a fully-fledged language.

Definition 5. *Execution of $\mathcal{F}ocal$ statements and expressions is defined in figure 2.2, and has the following shape:*

$$\begin{array}{lll}\leadsto & : & \textit{Module}\times\textit{state}\times\textit{Stmts}\ \longrightarrow\ \textit{state}\\ \leadsto & : & \textit{Module}\times\textit{state}\times\textit{Rhs}\ \longrightarrow\ \textit{heap}\times\textit{val}\end{array}$$

Note that execution is undefined – will be stuck – if we try to access fields or call methods which are not part of the object, and also, if we execute a conditional where the condition is not a boolean (neither **true** nor **false**). Such situations are possible, since $\mathcal{F}ocal$ is untyped. Wrt to security the question arises whether the language then guards against denial of service attacks - pass the wrong kind of object, and get stuck. The answer is: yes in theory. Even though we could guard against these by throwing exceptions, we cannot guard against denial of service attacks when passed an object on which "our" code will execute a method which will loop forever. All this does not affect the validity of our approach, since we are only claiming partial correctness.[4].

Linking We now define linking of modules, $M * M'$, to be the union of their respective mappings, provided that the domains of the two modules are disjoint wrt to classes, that the modules give the same definitions for specification identifiers, and that linking preserves the privacy of the constructors of any classes which had been declared as private.

Definition 6 (Linking)**.** *Linking of modules M and M' is*

$$*\ :\ Module\times Module\ \longrightarrow\ Module$$

$$M*M' = \left\{\begin{array}{ll}M *_{aux} M', & \textit{if } WFL(M,M')\\ \bot & \textit{otherwise.}\end{array}\right.$$

$$(M*_{aux}M')(c) = \left\{\begin{array}{ll}M(id), & \textit{if } M(id) \textit{ is defined}\\ M'(id) & \textit{otherwise.}\end{array}\right.$$

$$\begin{array}{l}WFL(M,M')\ \equiv\\ \quad dom(M)\cap dom(M')\cap ClassId{=}\emptyset\ \wedge\\ \quad\forall \textit{S}\in \textit{SpecId}\cap dom(M)\cap dom(M').\ M(\textit{S})=M(\textit{S}')\\ \quad WFP(M,M')\ \wedge\ WFP(M',M)\\ WFP(M,M')\ \ \equiv\\ \forall \textit{C}.\ M(\textit{C}) = \textbf{private....}\ \ \rightarrow\ \ \textbf{new}\ \textit{C... does not appear in } M'\end{array}$$

In the above, the predicate $WFL(M,M')$ asserts that linking of the modules M and M' is well-defined. It requires that 1) classes are not defined more than once, 2) specifications may have been defined more than once, but then their bodies must be identical[5], and 3) no module can call private

[4] TODO: Add similar disclaimer in the Soundness theorem

[5] We need to expand this, to also require that such "duplicate" specifications do not mention classes

constructors[6] from another module. This means, that the runtime system enforces this form of privacy. For example, if in module M_{mp} we define `Purse` as **private** and `Mint` as not private, the call of the `Purse` constructor is restricted to only insider the module M_{mp}, while the call of `Mint` is unrestricted. This means, that only objects of classes defined in the module M_{mp} may create `Purses`, while clients of M_{mp} may create objects of class `Mint`. In effect, the creation of `Mint` is publicly available, but the creation of `Purses` is restricted

The following lemma says that lookup is preserved by linking more modules.

Lemma 1. *For modules M and M' such that and $M' * M$ is defined, and class identifier $\texttt{C}$. method identifier $\texttt{m}$, predicate identifier $\texttt{Q}$, and ghost field identifier $\texttt{gf}$, we have*

- *If $\mathcal{M}(M, \texttt{C}, \texttt{m})$ is defined, then $\mathcal{M}(M, \texttt{C}, \texttt{m}) = \mathcal{M}(M' * M, \texttt{C}, \texttt{m})$.*
- *If $\mathcal{S}(M, \texttt{C})$ is defined, then $\mathcal{S}(M, \texttt{C}) = \mathcal{S}(M' * M, \texttt{C})$.*
- *If $\mathcal{P}(M, \texttt{C}, \texttt{Q})$ is defined, then $\mathcal{P}(M, \texttt{C}, \texttt{Q}) = \mathcal{P}(M' * M, \texttt{C}, \texttt{Q})$.*
- *If $\mathcal{G}(M, \texttt{C}, \texttt{gf})$ is defined, then $\mathcal{G}(M, \texttt{C}, \texttt{gf}) = \mathcal{G}(M' * M, \texttt{C}, \texttt{gf})$.*

Proof. By application of the definitions. □

The following lemma says that execution is preserved by linking more modules.

Lemma 2. *For all state σ and σ', modules M and M', and statements $stmts$, such that $M' * M$ is defined:*
If $M, \sigma, stmts \rightsquigarrow \sigma'$, then $M' * M, \sigma, stmts \rightsquigarrow \sigma'$.

Proof. By structural induction on the execution $\rightsquigarrow$ and lemma 1. □

Arising Configurations Policies need to be satisfied in all configurations which may arise during execution of some program. This leads us the concept of *arising* configuration. Arising configurations allow us to restrict the set of configurations we need to consider. For example, in a program where a class does not export visibility to a field, the constructor initialises the field to say 0, and all method calls increment that field, the arising configurations will only consider states where the field is positive.

We can now define $\mathcal{A}rising(M)$ as the set of runtime configurations which may be reached during execution of some initial context (σ_0,$\texttt{stmts}_0$), with the module Mexpanded with *any* module M'. A context is initial if its heap contains only objects of class `Object`.

Definition 7 (Arising and Initial configurations). *We define the mappings*

$$\begin{array}{lcl} \mathcal{I}nit & : & Module \longrightarrow \mathcal{P}(state \times Stmt) \\ \mathcal{A}rising & : & Module \longrightarrow \mathcal{P}(state \times Stmts) \end{array}$$

as follows:

$$\begin{array}{rl} \mathcal{I}nit(M) = & \{ \; (\; \sigma_0, \mathbf{new}\; c.m(\; \mathbf{new}\; c'\;) \;) \mid c, c' \in dom(M) \\ & \quad where\; \sigma_0 \;=\; ((\iota, \mathbf{null}), \chi_0), \\ & \quad and\; \chi_0(\iota) = (\texttt{Object}, \emptyset) \; \} \\ \mathcal{A}rising(M) = & \bigcup_{(\sigma, stmts) \in \mathcal{I}nit(M)} \mathcal{R}each(M, \sigma, \texttt{stmts}) \end{array}$$

Initial configuration should be as "minimal" as possible, We therefore construct a heap which has only one object, and execute a method call on a newly created object, with another newly created object as argument.

Reachable Configurations A configuration is reachable from another configuration, if the former may be required for the evaluation of the latter after any number of steps.

$$\begin{array}{rl} \mathcal{R}each \;:\; & Module \times state \times Stmts \\ & \longrightarrow \mathcal{P}(Stmts \times state) \end{array}$$

In figure 2 we define the function $\mathcal{R}each$ by cases on the structure of the expression, and depending on the execution of the statement. The set $\mathcal{R}each(M, \sigma, \texttt{stmts})$ collects all configurations reachable during execution of $\sigma, \texttt{stmts}$. Note that the function $\mathcal{R}each(M, \sigma, \texttt{stmts})$ is defined, even when the execution should diverge; of course then it may be an infinite set. The definedness of $\mathcal{R}each(M, \sigma, \texttt{stmts})$ is important, because it allows us to give meaning to capability policies without requiring termination.

Lemma 3 ($\mathcal{R}each$ and $\rightsquigarrow$)**.** *For all M, M', σ, σ', σ',and $\texttt{stmt}$, $\texttt{stmt}'$, and $\texttt{stmt}''$:*

- *If $M, \sigma, \texttt{stmt} \rightsquigarrow \sigma'$, then $(\_, \sigma') \in \mathcal{R}each(M, \sigma, \texttt{stmt})$.*
- *If $(\texttt{stmt}', \sigma') \in \mathcal{R}each(M, \sigma, \texttt{stmt})$, and $(\texttt{stmt}'', \sigma'') \in \mathcal{R}each(M, \sigma', \texttt{stmt}')$, then $(\texttt{stmt}''', \sigma'') \in \mathcal{R}each(M, \sigma, \texttt{stmt})$.*
- *If $M{*}M'$ is defined, and $(\texttt{stmt}', \sigma') \in \mathcal{R}each(M, \sigma, \texttt{stmt})$, then $(\texttt{stmt}', \sigma') \in \mathcal{R}each(M * M', \sigma, \texttt{stmt})$.*
- *If $M * M'$ is defined, then $\mathcal{A}rising(M) \subseteq \mathcal{A}rising(M * M')$.*

Proof By structural induction on $\rightsquigarrow$ and the definition of $\mathcal{R}each$ and $\mathcal{A}rising$. □

Notation We shall use $\sigma' \in \mathcal{R}each(M, \sigma, \texttt{stmt})$ as a shorthand for $(\_, \sigma') \in \mathcal{R}each(M, \sigma, \texttt{stmt})$ and $\sigma' \in \mathcal{A}rising(M)$ as a shorthand for $(\_, \sigma') \in \mathcal{A}rising(M)$.

3. The Specification Language Chainmail

Our specifications and policies are expressed in terms of one-state as well as two-state assertions. To express the state in which an expression is evaluated, we annotate it with a t-subscript. For example, given σ and σ' where $\lfloor x \rfloor_{sigma}$=4, and $\lfloor x \rfloor_{sigma}$=3, we have $M, \sigma, \sigma' \models_{\mathcal{O}} x_{pre} - x_{post} = 1$.

Expressions We first define expressions, $\texttt{e} \in Expr$, which depend on *one state* only. We allow the use of mathematical operators, like $+$ and $-$, and we use the identifier f to indicate functions whose value depends on the state (eg

[6] Why nor also private methods?

$$
\begin{array}{lcl}
\mathcal{R}each(M,\sigma,\texttt{v:=}\mathbf{new}\,\texttt{c(}\,\texttt{a}_1,...\texttt{a}_n\texttt{)}\,) & = & \{\,(\texttt{v:=}\mathbf{new}\,\texttt{c(}\,\texttt{a}_1,...\texttt{a}_n\texttt{)}\,,\sigma),\ (\mathbf{skip},\sigma')\} \\
 & & \text{where } M,\sigma,\texttt{v:=}\mathbf{new}\,\texttt{c(}\,\texttt{a}_1,...\texttt{a}_n\texttt{)}\ \rightsquigarrow\sigma' \\
\mathcal{R}each(M,\sigma,\texttt{stmt};\,\texttt{stmts}) & = & \mathcal{R}each(M,\sigma,\texttt{stmt})\cup\mathcal{R}each(M,\sigma',\texttt{stmts}) \\
 & & \text{where } M,\sigma,\texttt{stmt}\rightsquigarrow\sigma' \\
\mathcal{R}each(M,\sigma,\texttt{v:=a}) & = & \{(\texttt{v:=a},\sigma),\ (\mathbf{skip},\sigma')\} \\
 & & \text{where } M,\sigma,\texttt{v:=a}\rightsquigarrow\sigma' \\
\mathcal{R}each(M,\sigma,\texttt{v:=a.m(}\,\texttt{a}_1,...\texttt{a}_n\texttt{)}\,) & = & \{\,(\texttt{v:=a.m(}\,\texttt{a}_1,...\texttt{a}_n\texttt{)}\,,\sigma),\ (\mathbf{skip},\sigma''')\,\}\ \cup\ \mathcal{R}each(M,\sigma',\texttt{stmts}) \\
 & & \text{where } \sigma=\phi\cdot\chi \text{ and} \lfloor a\rfloor_{\phi\cdot\chi}=\iota \text{ and} \\
 & & \mathcal{M}(\texttt{M},\chi(\iota)\downarrow_1,\texttt{m})\ = \\
 & & \quad \mathbf{method}\ m(\,par_1,\dots par_n)\,\{\ stms;\ \mathbf{return}\ a'\}\ \text{ and} \\
 & & \phi'=\mathbf{this}\mapsto\iota, par_1\mapsto\lfloor a_1\rfloor_{\phi\cdot\chi},\dots par_n\mapsto\lfloor a_n\rfloor_{\phi\cdot\chi} \text{ and} \\
 & & M,\ \phi'\cdot\chi, stmts\ \rightsquigarrow\sigma'' \text{ and } \sigma'''=(\sigma\downarrow_1[\texttt{v}\mapsto\lfloor a'\rfloor_{\sigma''}],\sigma''\downarrow_2) \\
\mathcal{R}each(M,\sigma,\mathbf{skip}) & = & \{\,(\mathbf{skip},\sigma)\,\} \\
\mathcal{R}each(M,\sigma,\mathbf{if}\ \texttt{a}\ \mathbf{then}\ \texttt{stmts}_1\ \mathbf{else}\ \texttt{stmts}_2\,) & = & \{\,(\mathbf{if}\ \texttt{a}\ \mathbf{then}\ \texttt{stmts}_1\ \mathbf{else}\ \texttt{stmts}_2,\sigma),\ \}\ \cup\ \mathcal{R}each(M,\sigma,\texttt{stmts}'') \\
 & & \text{where } \texttt{stmts}''=\texttt{stmts}_1 \text{ if } \lfloor a\rfloor_\sigma=\mathbf{true},\ \text{ otherwise } \texttt{stmts}''=\texttt{stmts}_2
\end{array}
$$

Figure 2. Reachable Configurations

the function `length` of a list). We use identifier Q, or P to range over predicate identifiers.

The difference between expressions $\texttt{e}$, and arguments ar, as from the earlier section, is that expressions may depend ghost information, which is not stored explicitly in the state σ but can be deduced from it — e.g. the length of a list that is not stored with the list. Thus, expressions may read the fields from any object reachable from **this**, or any argument or local variable; they may access ghost fields, and they may call functions.

Definition 8 (Expressions)**.**

$$
\begin{array}{lcl}
\textit{Path} & ::= & \textit{ParId}\ \mid\ \textit{VarId}\ \mid\ \mathbf{this}\ \mid \textit{Path}.\,\textit{FieldId} \\
\textit{Expr} & ::= & \textit{Arg}\ \mid\ \textit{Path}\ \mid\ \textit{Val} \\
 & \mid & \mid\ \textit{Expr}+\textit{Expr}\ \mid\ ...\ \mid\ \textit{f(Expr}^*) \\
 & \mid & \mathbf{if}\ \textit{Expr}\ \mathbf{then}\ \textit{Expr}\ \mathbf{else}\ \textit{Expr} \\
\textit{funDescr} & ::= & \mathbf{function}\,\textit{f(}\ \textit{ParId}^*\ )\ \{\ \ \textit{Expr}\ \ \}
\end{array}
$$

We now define the value of such expressions through the interpretation function $\lfloor e\rfloor_{M,\sigma}$. We distinguish between the case where the function returns a value, is undefined, or unknown. The latter case arises only when the expression mentions functions which are unknown in the current module.

Definition 9 (Interpretations)**.** *We define the interpretation of expressions, as a* partial *function:*
$\lfloor\cdot\rfloor\ :\ Expr\times Module\times state\rightarrow Value$
using the notation $\lfloor\cdot\rfloor_{M,\sigma}$*:*

- $\lfloor val\rfloor_{M,\sigma}$ *= val, for all values* $val\in Val$*.*
- $\lfloor a\rfloor_{M,\sigma}$ = $\lfloor a\rfloor_\sigma$*, for all arguments* $a\ \in Arg$*.*
- $\lfloor p.f\rfloor_{M,\sigma}$ = $\chi(\lfloor p\rfloor_\sigma)(f)$ *if* $\lfloor p\rfloor_\sigma$ *is defined,*
  and f *is a field of* $\lfloor p\rfloor_\sigma$
  undefined, otherwise.
- $\lfloor p.gf\rfloor_{M,\sigma}$ = $\lfloor e[p/\texttt{this}]\rfloor_{M,\sigma}$ *if* $\lfloor p\rfloor_{M,\sigma}$ *is defined,*
  and e= $\mathcal{G}(M,\mathcal{C}lass(\lfloor p\rfloor_\sigma,\sigma),gf)$*),*
  unknown, if $\lfloor p\rfloor_\sigma,\sigma$ *defined,*
  and $M(\mathcal{C}lass(\lfloor p\rfloor_\sigma,\sigma))$ *undefined*
  undefined, otherwise.
- $\lfloor \texttt{e}_1+\texttt{e}_2\rfloor_{M,\sigma}$ = $\lfloor\texttt{e}_1\rfloor_{M,\sigma}+\lfloor\texttt{e}_2\rfloor_{M,\sigma}$,
  if $\lfloor\texttt{e}_1\rfloor_{M,\sigma}$ *and* $\lfloor\texttt{e}_2\rfloor_{M,\sigma}$ *are numbers,*
  undefined, if $\lfloor\texttt{e}_1\rfloor_{M,\sigma}$ *or* $\lfloor\texttt{e}_2\rfloor_{M,\sigma}$ *are undefined,*
  or not numbers
  unknown, otherwise.
- $\lfloor f(\texttt{e}_1,...\texttt{e}_n)\rfloor_{M,\sigma}$ = $\lfloor e[\texttt{e}_1/\texttt{this},\texttt{e}_2/p_2,...\texttt{e}_n/p_n]\rfloor_{M,\sigma}$
  where $M(f,\mathcal{C}lass(\texttt{e},\sigma))=$ **function** f **(** $p_2...p_n$**)** **{** $\texttt{e}$ **}** ,
  undefined, if $\exists i.\ \lfloor\texttt{e}_i\rfloor_{M,\sigma}$ *undefined,*
  unknown, if $\exists i.\ \lfloor\texttt{e}_i\rfloor_{M,\sigma}$ *unknown,*
  or if $M(\mathcal{C}lass(\texttt{e}_1,\sigma),f)$ *undefined.*
- $\lfloor\mathbf{if}\ \texttt{e}_0\ \mathbf{then}\ \texttt{e}_1\ \mathbf{else}\ \texttt{e}_2\rfloor_{M,\sigma}$
  =$\lfloor\ \texttt{e}_1\rfloor_{M,\sigma}$*, if* $\lfloor\ \texttt{e}_0\rfloor_{M,\sigma}$=**true***,*
  =$\lfloor\ \texttt{e}_2\rfloor_{M,\sigma}$*, if* $\lfloor\ \texttt{e}_0\rfloor_{M,\sigma}$=**false***.*
  undefined, if $\lfloor\ \texttt{e}_0\rfloor_{M,\sigma}$ *undefined,*
  or $\lfloor\ \texttt{e}_0\rfloor_{M,\sigma}$ *defined but not* **true** *or* **false***.*
  unknown, if $\lfloor\ \texttt{e}_0\rfloor_{M,\sigma}$ *unknown.*

Thus, the interpretation $\lfloor e\rfloor_{M,\sigma}$ is undefined if e refers to non-existing variables, or fields, or calls a infinite recursive function. For example, $\lfloor length(x)\rfloor_{M,\sigma}$ is undefined if x is not defined in σ, or if $\lfloor x\rfloor_\sigma$ points to a cyclic structure, and the $length$ function is defined with the obvious meaning in the class of x. On the other hand, the interpretation is unknown, if any subterm of e refers objects whose class is not defined in M. For example, if the class of x is not defined in M, then $\lfloor length(x)\rfloor_{M,\sigma}$ is unknown. Undefined expressions remain undefined in the context of larger modules, but unknown expressions may become known. For example, consider a module M' whichj contains a definition for t the class of x, then $\lfloor length(x)\rfloor_{M'*M,\sigma}$ is no longer unknown; it may be defined or undefined.

The value of a defined expression is preserved by linking. The same applies to undefinedness.

Lemma 4. *For any state σ, expression e, and modules M and M', where $M' * M$ is defined:*

- *If $\lfloor e \rfloor_{M,\sigma}$ is defined, then $\lfloor e \rfloor_{M,\sigma} = \lfloor e \rfloor_{M'*M,\sigma}$.*
- *If $\lfloor e \rfloor_{M,\sigma}$ is undefined, then $\lfloor e \rfloor_{M'*M,\sigma}$ is undefined.*

Proof. By structural induction on e and lemma 1. □

The opposite implication does not hold. It is possible to have, say $\lfloor e \rfloor_{M2*M1,\sigma}$=3, but $\lfloor e \rfloor_{M1,\sigma}$ is undefined. Also, it is possible to have $\lfloor e \rfloor_{M,\sigma}$ unknown, but $\lfloor e \rfloor_{M'*M,\sigma}$ defined (or also undefined).

One-state assertions We now define the syntax of one-state assertions, i.e. assertions whose validity depends on *one* state. These assertions include the standard assertions, such as expressions e, comparisons $e > e'$, and the usual operators on expressions. They also support the usual boolean combinators, ie implication, negation etc.[7] One state assertions also include predicates specific to our concerns: the assertions $\mathcal{M}ay\mathcal{A}ffect$ and $\mathcal{M}ay\mathcal{A}ccess$ which we use to model risk, the assertion `e : C` which expresses class membership, and the assertion `e` **obeys** `S` which expresses adherence to a specification. The two former predicates are *hypothetical*, in that they talk about the potential effect of execution of code, or of the existence of paths to connect two objects. In particular, the predicate $\mathcal{M}ay\mathcal{A}ffect$ ascertains whether its first parameter may execute code which affects the second one, while the predicate $\mathcal{M}ay\mathcal{A}ccess$ ascertains whether its first parameter has *any* path to the second one.

Definition 10 (One-state Assertions)**.**

$$
\begin{array}{lcl}
A & ::= & \textit{Expr} \ \mid\ \textit{Expr} \geq \textit{Expr} \ \mid\ ... \\
 & \mid & \textit{Q(Expr}^*\textit{)} \\
 & \mid & A \land A \ \mid\ A \rightarrow A \\
 & \mid & \exists x.A \ \mid\ \forall x.A \\
 & \mid & \textit{Expr}\textbf{:}\textit{ClassId} \\
 & \mid & \mathcal{M}ay\mathcal{A}ffect\ (\ \textit{Expr},\textit{Expr}) \\
 & \mid & \mathcal{M}ay\mathcal{A}ccess(\ \textit{Expr},\textit{Expr}) \\
 & \mid & \textit{Expr}\ \textbf{obeys}\ \textit{SpecId} \\
 & & \\
\textit{PredDescr} & ::= & \textbf{predicate}\ Q(\ \textit{ParId}^*\ )\ \{\ \ A\ \ \}
\end{array}
$$

We next defined validity of one-state assertions through a judgement of the form $M, \sigma \models_{\mathcal{O}} A$, which expresses that the assertion A holds for module M, state σ and using the oracle $\mathcal{O}$.[8] We first define what is an oracle.

Definition 11. *A function*
$\mathcal{O} \subseteq Module \times ClassId \times SpecId \ \longrightarrow\ \{true, false\}$ *is called an* oracle*, if for all modules M and M', class identifiers C and specification identifiers S,*

- *$C \notin dom(M)$, or $S \notin dom(M)$, then $\mathcal{O}(M, C, S)$=false*[9]
- *if $C, S \in dom(M)$, and $M*M'$ is defined, then $\mathcal{O}(M, C, S) = \mathcal{O}(M * M', C, S)$.*

The oracle $\mathcal{O}$ is needed to give meaning to the assertion $M, \sigma \models_{\mathcal{O}} e$ **obeys** `S`. We appeal to `C`, the class of the object indicated by e, and appeal to the oracle as to whether $\mathcal{O}(M, \texttt{C}, \texttt{S})$ holds. Later in this paper, cf. Def. 17, we define soundness of such oracles to means that whenever the oracle judges that a class `C` indeed adheres to a specification `S`, then all objects of this class behave according to `S`.

Note that whether an object obeys a spec depend on the state σ only insofar as the class of the object is concerned, and not on the contents of the object's fields or any other fields. This may seem strange - surely an object cannot fullfill its obligations unless its state is well-formed? However, we are only concerned with reachable states, and in these, if the class has been implemented in a robust manner, the object's state will be appropriate[10]

Note also, that judging that $M, \sigma \not\models_{\mathcal{O}}$ `e` **obeys** `S` does not imply that there exists some witness that the object will have behaviour contradicting `S`. In fact, it is even possible to have that a class's behaviour indeed adheres to the spec at a semantic level, but the oracle still returns $false$. [11]This treatment allows us to avoid circularities in the definitions with relatively light-weight means, as opposed to using notion like ... as in Devries [12]

Definition 12 (Validity of one-state assertions)**.** *Given an oracle $\mathcal{O} \subseteq Module \times ClassId$, the validity of an assertion A, is defined through the* partial *judgments:*

$$
\begin{array}{l}
\models\ \subseteq\ Module \times state \times Oracle \times Assertion \\
\not\models\ \subseteq\ Module \times state \times Oracle \times Assertion
\end{array}
$$

using the notations $M, \sigma \models_{\mathcal{O}} A$ and $M, \sigma \not\models_{\mathcal{O}} A$:

- $M, \sigma \models_{\mathcal{O}} \texttt{e}$, *if* $\lfloor \texttt{e} \rfloor_{M,\sigma} = \textbf{true}$,
  $M, \sigma \not\models_{\mathcal{O}} \texttt{e}$, *if* $\lfloor \texttt{e} \rfloor_{M,\sigma} = \textbf{false}$,
  undefined, otherwise.
- $M, \sigma \models_{\mathcal{O}} \texttt{e}_1 \geq \texttt{e}_2$, *if* $\lfloor \texttt{e}_1 \rfloor_{M,\sigma} \geq \lfloor \texttt{e}_2 \rfloor_{M,\sigma}$,
  $M, \sigma \not\models_{\mathcal{O}} \texttt{e}$, $\lfloor \texttt{e}_1 \rfloor_{M,\sigma} < \lfloor \texttt{e}_2 \rfloor_{M,\sigma}$,
  undefined, if $\lfloor \texttt{e}_1 \rfloor_{M,\sigma}$ or $\lfloor \texttt{e}_1 \rfloor_{M,\sigma}$ is undefined, or not a number.
- $M, \sigma \models_{\mathcal{O}} Q(\texttt{e}_0, \texttt{e}_1, ...\texttt{e}_n)$, *if*
  $M, \sigma \models_{\mathcal{O}} A[\texttt{e}_0/this, \texttt{e}_1/p_1, ...\texttt{e}_n/p_n]$
  $M, \sigma \not\models_{\mathcal{O}} Q(\texttt{e}_0, \texttt{e}_1, ...\texttt{e}_n)$, *if*
  $M, \sigma \not\models_{\mathcal{O}} A[\texttt{e}_0/this, \texttt{e}_1/p_1, ...\texttt{e}_n/p_n]$
  if $\mathcal{P}(M, \lfloor \texttt{e}_0 \rfloor_\sigma \downarrow_1, Q) = \textbf{predicate}\ Q\ (\ p_1...p_n)\ \{\ \ A\ \}$,

[7] All these may be encode through $\rightarrow$, the value **false** and $\land$, so we only defined these.

[8] want better name than oracle.

[9] perhaps undefined instesad? SD thinks it does not matter

[10] TODO explain this betterm as it is crucial.

[11] TODO expand.

[12] TODO expand. TODO perhaps move some of this to the section ojn Hoare Logic.

undefined, if $\mathcal{P}(M, \lfloor e_0 \rfloor_\sigma \downarrow_1, Q)$ undefined, or if $M, \sigma \models_{\mathcal{O}} A(e_0, e_1, ... e_n)$ undefined.

- *$M, \sigma \models_{\mathcal{O}} A_1 \wedge A_2$, if $M, \sigma \models_{\mathcal{O}} A_1$ and $M, \sigma \models_{\mathcal{O}} A_2$,*
  $M, \sigma \not\models_{\mathcal{O}} A_1 \wedge A_2$, if $M, \sigma \not\models_{\mathcal{O}} A_1$ or $M, \sigma \not\models_{\mathcal{O}} A_2$
  undefined, if $M, \sigma \models_{\mathcal{O}} A_1$ or $M, \sigma \models_{\mathcal{O}} A_2$ is undefined.
- *$M, \sigma \models_{\mathcal{O}} A_1 \rightarrow A_2$, if $M, \sigma \models_{\mathcal{O}} A_1$ and $M, \sigma \models_{\mathcal{O}} A_2$, or $M, \sigma \not\models_{\mathcal{O}} A_1$.*
  $M, \sigma \not\models_{\mathcal{O}} A_1 \rightarrow A_2$, if $M, \sigma \models_{\mathcal{O}} A_1$ and $M, \sigma \not\models_{\mathcal{O}} A_2$,
  undefined, if $M, \sigma \models_{\mathcal{O}} A_1$ or $M, \sigma \models_{\mathcal{O}} A_2$ is undefined.
- *$M, \sigma \models_{\mathcal{O}} \exists x.A$ iff for some address ι and some fresh variable $z \in VarId$, we have $M, \sigma[z \mapsto \iota] \models_{\mathcal{O}} A[z/x]$.*
  $M, \sigma \not\models_{\mathcal{O}} \exists x.A$ iff for all address ι and fresh variable $z \in VarId$, we have $M, \sigma[z \mapsto \iota] \not\models_{\mathcal{O}} A[z/x]$.
  undefined, otherwise.
- *$M, \sigma \models_{\mathcal{O}} \forall x.A$ iff for all addresses $\iota \in dom(\sigma)$, and fresh variable z, we have $M, \sigma[z \mapsto \iota] \models_{\mathcal{O}} A[z/x]$.*
  $M, \sigma \not\models_{\mathcal{O}} \forall x.A$ iff there exists an address $\iota \in dom(\sigma)$, and fresh variable z, such that $M, \sigma[z \mapsto \iota] \not\models_{\mathcal{O}} A[z/x]$.
  undefined, otherwise.
- *$M, \sigma \models_{\mathcal{O}} \mathtt{e}{:}\mathtt{C}$, if $\sigma(\lfloor \mathtt{e} \rfloor_{M,\sigma}) \downarrow_1 = \mathtt{C}$.*
  $M, \sigma \not\models_{\mathcal{O}} \mathtt{e}{:}\mathtt{C}$, if $\sigma(\lfloor \mathtt{e} \rfloor_{M,\sigma}) \downarrow_1 \neq \mathtt{C}$.
  undefined, if $\sigma(\lfloor \mathtt{e} \rfloor_{M,\sigma}) \notin dom(\sigma \downarrow_2)$.
- *$M, \sigma \models_{\mathcal{O}} \mathcal{M}ay\mathcal{A}ffect(\mathtt{e}, \mathtt{e}')$, if $\lfloor \mathtt{e} \rfloor_{M,\sigma}$ and $\lfloor \mathtt{e}' \rfloor_{M,\sigma}$ are defined, and there exists a method $\mathtt{m}$, arguments $\bar{\mathtt{a}}$, state σ', identifier z, such that $M, \sigma[z \mapsto \lfloor \mathtt{e} \rfloor_{M,\sigma}], z.\mathtt{m}(\bar{\mathtt{a}}) \leadsto \sigma'$, and $\lfloor \mathtt{e}' \rfloor_{M,\sigma} \neq \lfloor \mathtt{e}' \rfloor_{M,\sigma \downarrow_1, \sigma' \downarrow_1}$.*
  $M, \sigma \models_{\mathcal{O}} \mathcal{M}ay\mathcal{A}ffect(\mathtt{e}, \mathtt{e}')$, undefined if $\lfloor \mathtt{e} \rfloor_{M,\sigma}$ or $\lfloor \mathtt{e}' \rfloor_{M,\sigma}$ are undefined.
  $M, \sigma \not\models_{\mathcal{O}} \mathcal{M}ay\mathcal{A}ffect(\mathtt{e}, \mathtt{e}')$, otherwise.
- *$M, \sigma \models_{\mathcal{O}} \mathcal{M}ay\mathcal{A}ccess(\mathtt{e}, \mathtt{e}')$, if $\lfloor \mathtt{e} \rfloor_{M,\sigma}$ and $\lfloor \mathtt{e}' \rfloor_{M,\sigma}$ are defined, and there exist fields $\mathtt{f}_1, ... \mathtt{f}_n$, such that $\lfloor z.\mathtt{f}_1 ... \mathtt{f}_n \rfloor_{M, \sigma[z \mapsto \lfloor \mathtt{e} \rfloor_{M,\sigma}]} = \lfloor \mathtt{e}' \rfloor_{M,\sigma}$.*
  $M, \sigma \models_{\mathcal{O}} \mathcal{M}ay\mathcal{A}ccess(\mathtt{e}, \mathtt{e}')$, undefined if $\lfloor \mathtt{e} \rfloor_{M,\sigma}$ or $\lfloor \mathtt{e}' \rfloor_{M,\sigma}$ are undefined.
  $M, \sigma \not\models_{\mathcal{O}} \mathcal{M}ay\mathcal{A}ccess(\mathtt{e}, \mathtt{e}')$, otherwise.
- *$M, \sigma \models_{\mathcal{O}} \mathtt{e}$ **obeys** S, undefined, if $\lfloor \mathtt{e} \rfloor_{M,\sigma}$ undefined, unknown, if $\lfloor \mathtt{e} \rfloor_{M,\sigma}$ unknown, or $\mathcal{C}lass(\mathtt{e}, \sigma) \notin dom(M)$.*
  *$M, \sigma \models_{\mathcal{O}} \mathtt{e}$ **obeys** S, if $\mathcal{O}(M, C, S) = true$*
  *$M, \sigma \not\models_{\mathcal{O}} \mathtt{e}$ **obeys** S, if $\mathcal{O}(M, C, S) = false$*
  where $\mathtt{C}{=}\mathcal{C}lass(\mathtt{e}, \sigma)$.

In the above, the notation $\sigma[v \mapsto \iota]$ is shorthand for $(\phi[v \mapsto \iota], \chi)$ for a state $\sigma = (\phi, \chi)$.

Note that the definition of $M, \sigma \models_{\mathcal{O}} \mathtt{e}$ **obeys** $\mathtt{S}$ only depends on whether $\mathtt{C}$, the *class* of the object denoted by $\mathtt{e}$, is considered by the oracle $\mathcal{O}$ to satisfy $\mathtt{S}$. In particular, $M, \sigma \models_{\mathcal{O}} \mathtt{e}$ **obeys** $\mathtt{S}$ does not imply that $\mathtt{e}$ satisfies the policies from $\mathtt{S}$. In order to have this guarantee we also need to know that $\mathcal{O}$ is sound - as defined in Def 17.

Some examples to do with undefined assertions and preservation under linking can be found in section 5.1. In particular, in a state σ where $\mathtt{x}$ points to an object of class $\mathtt{c}$, and module M where class $\mathtt{c}$ is undefined, and any specification identifier $\mathtt{S}$, we have that $M, \sigma \models_{\mathcal{O}} \mathtt{x}$ **obeys** $\mathtt{S}$ is undefined, and $M, \sigma \models_{\mathcal{O}} \mathtt{x}$ **obeys** $\mathtt{S} \rightarrow \mathbf{true}$ is also undefined.

However, the situation that the class of an object in the state is undefined in a module cannot happen in states that have been reached by execution of a module:

Lemma 5. *For all modules M, all states σ, and specification identifiers S:*

- *If $\sigma \in \mathcal{A}rising(M)$ and $\mathtt{x} \in dom(\sigma)$, then $M, \sigma \models_{\mathcal{O}} \mathtt{x}$ **obeys** S is defined.*

Proof. We first prove that in any $\sigma \in \mathcal{A}rising(M)$, any object in σ has a class that is defined in M. This can be proven by induction on the execution. It holds because objects can only be crated if M contains a class declaration for the respective class. We then argue that under this condition the judgment $M, \sigma \models_{\mathcal{O}} \mathtt{x}$ **obeys** $\mathtt{S}$ is well-defined for any $\mathtt{S}$, provided that □

Therefore, for any $\mathcal{O}$, if $\sigma \in \mathcal{A}rising(M)$ and $\mathtt{x} \in dom(\sigma)$, then either $M, \sigma \models_{\mathcal{O}} \mathtt{x}$ **obeys** $\mathtt{S}$ or $M, \sigma \not\models_{\mathcal{O}} \mathtt{x}$ **obeys** $\mathtt{S}$. This is so, because we always either $\mathcal{O}(M, \mathtt{C}, \mathtt{S}) = true$, or $\mathcal{O}(M, \mathtt{C}, \mathtt{S}) = false$.

Validity of one-state assertions is preserved by linking, as expressed by the lemma below:

Lemma 6. *For any state σ, assertion A, and modules M and M', where $M' * M$ is defined:*

- *If $M, \sigma \models_{\mathcal{O}} A$ holds, then $M' * M, \sigma \models_{\mathcal{O}} A$.*
- *If $M, \sigma \not\models_{\mathcal{O}} A$ holds, then $M' * M, \sigma \not\models_{\mathcal{O}} A$.*

Proof. By structural induction on A and lemmas 1 and 6. In several inductive cases we use the fact that valictydity of the subterms must be defined, For example, in the case of implication, i.e. when A has the form $A_1 \rightarrow A_2$ we use the fact that validity of A_1 must be defined. □

TODO Say somewhere that even though it is possible to construct sound $\mathcal{O}$s, the $\mathcal{O}$ is not unique.

Two state assertions Two-state assertions allow us to compare properties of two different states, and thus say, e.g. that $\mathtt{p.balance}_{post} = \mathtt{p.balance}_{pre} + 10$. To differentiate between the two states we use the subscripts pre and post.

Definition 13 (Two-state Assertions)**.**

$$
\begin{array}{lcl}
t & ::= & \mathit{pre} \mid \mathit{post} \mid \epsilon \\
B & ::= & A_t \\
 & \mid & Expr_t \geq Expr_t \mid ... \\
 & \mid & \mathcal{N}ew(Expr) \\
 & \mid & B \wedge B \mid ... \\
 & \mid & \exists x.B \mid \forall x.B .
\end{array}
$$

Given the syntax from above, we can express assertions like $\forall \mathtt{p.p} :_{pre} \mathtt{Purse}$.

$[\,\texttt{p.bank}_{pre} =_{pre} \texttt{RBS} \rightarrow$
$\quad \texttt{p.balance}_{pre} = \texttt{p.balance}_{post}\,]$,
to require that the balance of any Purse belonging to RBS is immutable across the to states.

Validity of one-state, two-state assertions, and policies We now define validity of two state assertions, ...

Definition 14 (Validity of Two-state assertions)**.** *We define the judgment*

$$\models \;\subseteq\; Module \times state \times state \times Oracle \times TwoStateAssertion$$

using the notation $M, \sigma, \sigma' \models_{\mathcal{O}} B$ *as follows*

- $M, \sigma, \sigma' \models_{\mathcal{O}} A_t$ *iff* $M, \sigma'' \models_{\mathcal{O}} A$, *where* $\sigma'' = \sigma$ *if* t=pre*, and* $\sigma'' = \sigma'$ *otherwise.*
- $M, \sigma, \sigma' \models_{\mathcal{O}} e_t \geq e'_{t'}$*, iff* $\lfloor e \rfloor_{M,\sigma_1} \geq \lfloor e' \rfloor_{M,\sigma_2}$*, where* $\sigma_1 = \sigma$ *if* t=pre*, and* $\sigma_1 = \sigma'$ *otherwise, and* $\sigma_2 = \sigma$ *if* t'=pre*, and* $\sigma_2 = \sigma'$ *otherwise.*
- $M, \sigma, \sigma' \models_{\mathcal{O}} \mathcal{N}ew(e)$ *iff* $\lfloor e \rfloor_{M,\sigma'} \in dom(\sigma') \setminus dom(\sigma)$
- $M, \sigma, \sigma' \models_{\mathcal{O}} B_1 \wedge B_2$ *iff* $M, \sigma, \sigma' \models_{\mathcal{O}} B_1$ *and* $M, \sigma, \sigma' \models_{\mathcal{O}} B_2$.
- $M, \sigma, \sigma' \models_{\mathcal{O}} \exists x.B$ *iff for some address* ι *and fresh variable* z*, we have* $M, \sigma[z \mapsto \iota], \sigma'[z \mapsto \iota] \models_{\mathcal{O}} B[z/x]$.
- $M, \sigma, \sigma' \models_{\mathcal{O}} \forall x.B$ *iff for all addresses* ι *and fresh variable* z*, we have* $M, \sigma[z \mapsto \iota], \sigma'[z \mapsto \iota] \models_{\mathcal{O}} B[z/x]$.

For example, for states σ_1, σ_2 where $\lfloor \texttt{x.balance} \rfloor_{\sigma_1} = 4$ and $\lfloor \texttt{x.balance} \rfloor_{\sigma 2} = 14$, we have $M, \sigma_1, \sigma_2 \models_{\mathcal{O}} \texttt{x.balance}_{post} = \texttt{x.balance}_{pre} + 10$.

Policies are expressed in terms of one-state assertions A, A', etc. and two state assertions B, B'' etc.

Policies can have one of the three following forms: 1) invariants of the form A, which require that A holds at all visible states of a program; or 2) $A\,\{$ code $\}\,B$, which require that execution of code in any state which satisfies A will lead to a state which satisfies B wrt the original state; or 3) $A\,\{$any_code$\}\,B$ which, similar to two state invariants, requires that execution of *any* code in a state which satisfies A will lead to a state which satisfies B.

Definition 15 (Policies)**.**

$$\begin{array}{lcl} Policy & ::= & A \mid A\,\{\texttt{res = this.m(par)}\}\,B \\ & \mid & A\,\{\texttt{res = new(x1, ..xn)}\}\,B \\ & \mid & A\,\{\textbf{any\_code}\}\,B \\ PolSpec & ::= & spec\; SpcId\{\; Policy^*\;\} \end{array}$$

[13] .

3.1 Connectivity Properties

A large amount of research into is concerned with properties of connectivity. For this we will first prove lemmas about connectivity.

[13] Do we want to policies for constructors A {res = new(x1, ..xn)} B and do we still want to policy A {any_code} B – we can add both; the next chapter is incomplete wrt the two latter ones

Lemma 7 (Only Connectivity Begets Connectivity for Existing Objects)**.** *For any module* M*, oracle* $\mathcal{O}$*, frame* ϕ*,*

- *If* $M, \phi \cdot \chi, \texttt{stmts} \rightsquigarrow \phi' \cdot \chi'$ *and if* $\{x, z\} \subseteq dom(\chi)$*, and* $M, \phi' \cdot \chi' \models_{\mathcal{O}} \mathcal{M}ay\mathcal{A}ccess(x, z)$*, then* $M, \phi \cdot \chi \models_{\mathcal{O}} \mathcal{M}ay\mathcal{A}ccess(x, z)$ *or* $\exists w, y \in dom(\chi) \cap dom(\phi).\; M, \phi \cdot \chi \models_{\mathcal{O}} \mathcal{M}ay\mathcal{A}ccess(w, x) \wedge \mathcal{M}ay\mathcal{A}ccess(y, z)$.
- *If* $M, \phi \cdot \chi, rhs \rightsquigarrow \chi'$ *and if* $\{x, z\} \subseteq dom(\chi)$*, and* $M, \phi \cdot \chi' \models_{\mathcal{O}} \mathcal{M}ay\mathcal{A}ccess(x, z)$ *then* $M, \phi \cdot \chi \models_{\mathcal{O}} \mathcal{M}ay\mathcal{A}ccess(x, z)$ *or* $\exists w, y \in dom(\chi) \cap dom(\phi).\; M, \phi \cdot \chi \models_{\mathcal{O}} \mathcal{M}ay\mathcal{A}ccess(w, x) \wedge \mathcal{M}ay\mathcal{A}ccess(y, z)$.
- *If* $\phi' \cdot \chi' \in \mathcal{R}each(M, \phi \cdot \chi, \texttt{stmts})$ *and if* $\{x, z\} \subseteq dom(\chi)$*, and* $M, \phi' \cdot \chi' \models_{\mathcal{O}} \mathcal{M}ay\mathcal{A}ccess(x, z)$ *then* $M, \phi \cdot \chi \models_{\mathcal{O}} \mathcal{M}ay\mathcal{A}ccess(x, z)$ *or* $\exists w, y \in dom(\chi) \cap dom(\phi).\; M, \phi \cdot \chi \models_{\mathcal{O}} \mathcal{M}ay\mathcal{A}ccess(w, x) \wedge \mathcal{M}ay\mathcal{A}ccess(y, z)$.

This lemma expresses a basic property of object-capability systems that “only connectivity begets connectivity” [1], and says that the only way that some existing object z can become accessible by some existing x through executing statements stmts or expression *rhs* is if each is already accessible to some y and w respectively that are both in the current stack frame ϕ.

Proof. The first two bullets we prove by structural induction over the mutual inductive definitions of $\rightsquigarrow$ for statements and expressions. The final bullet is then proved by structural induction over statements using the definition of $\mathcal{R}each$. □

The following lemma expresses the only connectivity begets connectivity property for newly created objects.

Lemma 8 (Only Connectivity Begets Connectivity for New Objects)**.**

- *If* $M, \phi \cdot \chi, \texttt{stmts} \rightsquigarrow \phi' \cdot \chi'$ *and if* $x \in dom(\chi)$ *and* $z \notin dom(\chi)$*, and* $M, \phi' \cdot \chi' \models_{\mathcal{O}} \mathcal{M}ay\mathcal{A}ccess(x, z) \vee \mathcal{M}ay\mathcal{A}ccess(z, x)$*, then* $\exists w \in dom(\chi) \cap dom(\phi).\; M, \phi \cdot \chi \models_{\mathcal{O}} \mathcal{M}ay\mathcal{A}ccess(w, x)$.
- *If* $M, \phi \cdot \chi, rhs \rightsquigarrow \chi'$ *and if* $x \in dom(\chi)$ *and* $z \notin dom(\chi)$*, and* $M, \phi \cdot \chi' \models_{\mathcal{O}} \mathcal{M}ay\mathcal{A}ccess(x, z) \vee \mathcal{M}ay\mathcal{A}ccess(z, x)$*, then* $\exists w \in dom(\chi) \cap dom(\phi).\; M, \phi \cdot \chi \models_{\mathcal{O}} \mathcal{M}ay\mathcal{A}ccess(w, x)$.
- *If* $\phi' \cdot \chi' \in \mathcal{R}each(M, \phi \cdot \chi, \texttt{stmts})$ *and if* $x \in dom(\chi)$ *and* $z \notin dom(\chi)$*, and* $M, \phi' \cdot \chi' \models_{\mathcal{O}} \mathcal{M}ay\mathcal{A}ccess(x, z) \vee \mathcal{M}ay\mathcal{A}ccess(z, x)$*, then* $\exists w \in dom(\chi) \cap dom(\phi).\; M, \phi \cdot \chi \models_{\mathcal{O}} \mathcal{M}ay\mathcal{A}ccess(w, x)$.

It says that if an existing object x gains access to an object z by executing stmts or *rhs* in which z is newly created, then x must have already been accessible to some objecct w in the initial stack frame ϕ.

Proof. Analogous to Lemma 7. □

(VARASG)

$$\frac{}{\begin{array}{l}\vdash\ \textbf{true}\,\{\,\textbf{var}\ \texttt{v:=a}\,\}\ \texttt{v} = \texttt{a}_{pre}\ \bowtie\ \textbf{true}\\ \vdash\ \textbf{true}\,\{\,\texttt{v:=a}\,\}\ \texttt{v} = \texttt{a}_{pre}\ \bowtie\ \textbf{true}\end{array}}$$

(FIELDASG)

$$\frac{}{\vdash\ \textbf{true}\,\{\,\textbf{this}.\texttt{f:=a}\,\}\ \textbf{this}.\texttt{f} = \texttt{a}_{pre}\ \bowtie\ \textbf{true}}$$

(COND-1)

$$\frac{\begin{array}{l}A \to_M \texttt{cond}\\ \vdash\ A\,\{\,\texttt{stmts}_1\,\}\,B\ \bowtie\ B'\end{array}}{\vdash\ A\,\{\,\textbf{if}\ \texttt{cond}\ \textbf{then}\ \texttt{stmts}_1\ \textbf{else}\ \texttt{stmts}_2\,\}\,B\ \bowtie\ B'}$$

(COND-2)

$$\frac{\begin{array}{l}A \to_M \neg\texttt{cond}\\ \vdash\ A\,\{\,\texttt{stmts}_2\,\}\,B\ \bowtie\ B'\end{array}}{\vdash\ A\,\{\,\textbf{if}\ \texttt{cond}\ \textbf{then}\ \texttt{stmts}_1\ \textbf{else}\ \texttt{stmts}_2\,\}\,B\ \bowtie\ B'}$$

(SKIP)

$$\frac{}{\vdash\ A\,\{\,\textbf{skip}\,\}\,A\ \bowtie\ \textbf{true}}$$

(NEWOBJ)

$$\frac{M(\mathcal{S}(M,\texttt{C}))\ =\ \textbf{spec}\ S\,\textbf{\{}\ \ldots.,\ A\textbf{\{}\ \textbf{new}(\texttt{x}_1,\ldots\texttt{x}_n)\,\textbf{\}}\ B,\ \ldots.\ \textbf{\}}}{\vdash\ A[\texttt{a}_1/\texttt{x}_1,\ldots\texttt{a}_n/\texttt{x}_n]\,\{\,\texttt{x} := \textbf{new(}\ \texttt{a}_1\ldots\texttt{a}_n\textbf{)}\ \}\ B[\texttt{a}_1/\texttt{x}_1,\ldots\texttt{a}_n/\texttt{x}_n,\texttt{x}/\texttt{res}]\ \wedge\ \texttt{x}\,\textbf{obeys}\,\mathcal{S}(M,\texttt{C})\ \bowtie\ \textbf{true}}$$

(METH-CALL-1)

$$\frac{M(S)\ =\ \textbf{spec}\ S\,\textbf{\{}\ \overline{\mathit{Policy}},A\textbf{\{}\ \texttt{this.m(par)}\ \textbf{\}}\ B,\overline{\mathit{Policy}'}\ \textbf{\}}}{\vdash\ \texttt{x}\,\textbf{obeys}\,S\wedge A[\texttt{x}/\texttt{this},\texttt{y}/\texttt{par}]\,\{\,\texttt{v} := \texttt{x.m(y)}\ \}\,B[\texttt{x}/\texttt{this},\texttt{y}/\texttt{par},\texttt{v}/\texttt{res}]\ \bowtie\ \textbf{true}}$$

(METH-CALL-2)

$$\frac{\begin{array}{l}B \equiv \forall\texttt{z}:_{pre}\texttt{Object}.\,[\,\mathcal{MayAccess}(\texttt{v},\texttt{z}) \to (\,\mathcal{MayAccess}_{pre}(\texttt{x},\texttt{z})\ \vee\ \mathcal{MayAccess}_{pre}(\texttt{y},\texttt{z})\,)\ \,]\\ B' \equiv \forall\texttt{z},\texttt{u}:_{pre}\texttt{Object}.\,[\ \mathcal{MayAccess}(\texttt{u},\texttt{z}) \to\\ \qquad\qquad\qquad\qquad\qquad\qquad (\mathcal{MayAccess}_{pre}(\texttt{u},\texttt{z})\ \ \vee\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad [\,(\mathcal{MayAccess}_{pre}(\texttt{x},\texttt{z}) \vee \mathcal{MayAccess}_{pre}(\texttt{y},\texttt{z}))\wedge\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ (\mathcal{MayAccess}_{pre}(\texttt{x},\texttt{u}) \vee \mathcal{MayAccess}_{pre}(\texttt{y},\texttt{u}))\,]\,)\ \,]\\ \qquad\wedge\\ \qquad\forall\texttt{z}:_{pre}\texttt{Object}.\forall\texttt{u}:\texttt{Object}.\\ \qquad\qquad\qquad\qquad\quad [\ \mathcal{New}(\texttt{u}) \wedge (\ \mathcal{MayAccess}(\texttt{z},\texttt{u}) \vee \mathcal{MayAccess}(\texttt{u},\texttt{z})\ )\quad\to\\ \qquad\qquad\qquad\qquad\qquad\qquad\quad (\ \mathcal{MayAccess}_{pre}(\texttt{x},\texttt{z}) \vee \mathcal{MayAccess}_{pre}(\texttt{y},\texttt{z})\ )\ \ ]\end{array}}{\vdash\ \texttt{true}\ \{\texttt{v} := \texttt{x.m(y)}\,\}\,B\ \bowtie\ B'}$$

(FRAME-METHCALL)

$$\frac{\begin{array}{l}\vdash\ A\wedge\texttt{e}=\texttt{e}\,\{\,\texttt{x.m(y)}\,\}\,\textbf{true}\ \bowtie\ \forall\texttt{z}.(\ \mathcal{MayAffect}(\texttt{z},\texttt{e}) \to B'(\texttt{z})\ )\ \wedge\\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \forall\texttt{z}.(\ (\mathcal{MayAccess}_{pre}(\texttt{x},\texttt{z}) \vee \mathcal{MayAccess}_{pre}(\texttt{y},\texttt{z}) \vee \mathcal{New}(\texttt{z})\ )\ \to\ \neg B'(\texttt{z})\end{array}}{\vdash\ A\wedge\texttt{e}=\texttt{e}\,\{\,\texttt{x.m(y)}\,\}\,\texttt{e}=\texttt{e}_{pre}\ \bowtie\ \textbf{true}}$$

(SEQUENCE)

$$\frac{\vdash\ A\,\{\,\texttt{stmts}_1\,\}\,B_1\ \bowtie\ B' \qquad \vdash\ A_2\,\{\,\texttt{stmts}_2\,\}\,B_2\ \bowtie\ B' \qquad A,B_1 \to_{M,\mathcal{O}} \textbf{true},A_2 \qquad B_1,B_2 \to_{M,\mathcal{O}} B}{\vdash\ A\,\{\,\texttt{stmts}_1;\ \texttt{stmts}_2\,\}\,B\ \bowtie\ B'}$$

Figure 3. Hoare Logic – Basic rules of the language – we assume that the module M is globally given

4. Hoare Logic

We define the Hoare Logic that allows us to prove adherence to policies. In order to reflect that the code to be verified is executed in an open system, and that it calls code whose specification and trustworthiness is unknown to the code being verified, we augment the Hoare triples, so that not only do they guarantee some property to hold *after* execution of the code, but also guarantee that some property is preserved *during* execution of the code.

A Hoare tuple in our system has the format

$M \vdash A \{ \texttt{stms} \} B' \bowtie B$,

and promises that execution of stms in any state which satisfies A will lead to a state where the relation of the old and new state is described by B. It also promises that the relation between the initial state, and any of the the intermediate states reached by execution of stms will be described by B.

Notice that the judgement is expressed in terms of a module M. This is needed, because execution of stmts may call methods defined in M, and also because the assertions appearing in A, B, and B', may use predicates defined in M. When the module M is implicit from the context we use the shorthand

$\vdash A \{ \texttt{stms} \} A' \bowtie B$.

4.1 Preliminaries

Before discussing the Hoare logic, we need to define entailment and disjointnes.

4.1.1 Entailment

As usual in Hoare logics, we have rules of consequence and for these we need a notion of entailment. Since we have both one- and two-state assertions, we support several versions of entailment:

The standard entailment, *i.e.* $A \rightarrow_M A'$, guarantees that any state which satisfies A also satisfies A'. We extend the notion to cater for two state assertions, and have three new forms of entailment, described in Definition 18. The entailment $A, B_1 \rightarrow_M \mathbf{true}, A_2$ guarantees that for any pair of states if the former states satisfies A and the two together satisfy B_1, then the second state will also satisfy A_2, *c.f.* Definition 18.3. The entailment $B_1, B_2 \rightarrow_M B$ guarantees for any three states, if the first two together satisfy B_1, and the second and third together satisfy B_2, then the first and third will satisfy B, *c.f.* Definition 18.5. For example, according to Definition. 18.3 we have

$\texttt{x} = 5, \texttt{x}_{post} = \texttt{x} + 2 \rightarrow_M \mathbf{true}, \texttt{x} = 7$,

while with Definition 18.5 we have

$\texttt{x}_{post} = \texttt{x} + 4, \texttt{x}_{post} = \texttt{x} + 2 \rightarrow_M \texttt{x}_{post} = \texttt{x} + 6$

for any module M.

The entailment $\rightarrow_M$, needs to preserve validity ($\models$). The latter is defined wrt oracles, therefore $\rightarrow_M$ needs to preserve validity with respect to oracle ($\models_{\mathcal{O}}$). But only for *sound* oracles. Therefore, we first define sound oracles: An oracle is *sound for a particular class* C *and module* M, if it either classifies the class badly formed (ie $\mathcal{O}(M, \texttt{C}) = true$), or it classifies it as well-formed (ie $\mathcal{O}(M, \texttt{C}) = true$) and the class behaves according to its specification in all possible reachable contexts:

We define adherence to an invariant as follows

Definition 16 (Invariants). *Given an oracle $\mathcal{O}$, a module M and class* C*, we define:*

- $M \models_{\mathcal{O}} \texttt{C} : A$ *iff*
  for all M', and all $\sigma \in \mathcal{A}rising(M' * M)$.[14]
  $M * M', \sigma \models_{\mathcal{O}} x : \texttt{C} \rightarrow A[x/\texttt{this}]$
- $M \models_{\mathcal{O}} \texttt{C} : A\{ \texttt{res=this.m(x)} \}B$ *iff*
  for all M, all a1, a2, ... $\sigma \in \mathcal{A}rising(M' * M)$.*:*
  $\mathcal{C}lass(\texttt{a1})_\sigma = \texttt{C}$, *and*
  $M * M', \sigma \models_{\mathcal{O}} A[\texttt{a1}/\texttt{this}, \texttt{a2}/x]$, *and*
  $M * M', \sigma, \texttt{u=a1.m(a2)} \rightsquigarrow \sigma'$, *implies*
  $M * M', \sigma, \sigma' \models_{\mathcal{O}} B[\texttt{u}/\texttt{res}, \texttt{a1}/\texttt{this}, \texttt{a2}/x]$.
- *TODO: add similar requirement for constructors - if indeed we decide to add them to specs $M \models \texttt{C} : A$ iff for all possible oracles $\mathcal{O}$, we have that $M \models_{\mathcal{O}} \texttt{C} : A$*

The judgment $M, \texttt{C} \models_{\mathcal{O}} A$ expresses something similar to A being a class or object invariant [15], with the difference that we it is with respect to the oracle , and that we require A to hold in *all*, rather just in *visible* states.[16] We do not define such a sound judgment in our paper, but we claim that it is possible to develop such judgments and aim to do that in further work. We give example arguments that lead to such a judgment in appendix ???[17]

Definition 17 (Sound Oracle). *For oracle $\mathcal{O}$ we define:*

- $\mathcal{O}$ *is* sound for module M and class C, *if $\mathcal{O}(M, \texttt{C}) = false$, or for all policies from* C*'s specification $Policy \in \mathcal{S}(M, \texttt{C})$ we have $M \models_{\mathcal{O}} \texttt{C} : Policy$.*
- $\mathcal{O}$ *is* sound *if it sound for all classes and modules.*

Based on this, we now define entailment of assertions as follows:

Definition 18 (Entailment). *We define five entailment relations, below*

1. $A \rightarrow_M A'$ *iff*
   $\forall M'.\forall(\_, \sigma) \in \mathcal{A}rising(M * M').\forall$*sound* $\mathcal{O}$.
   $M * M', \sigma \models_{\mathcal{O}} A \rightarrow A'$.
2. $B \rightarrow_M B'$ *iff*
   $\forall M'.\forall(\_, \sigma), (\_, \sigma') \in \mathcal{A}rising(M * M').\forall$*sound* $\mathcal{O}$.
   $M * M', \sigma, \sigma' \models_{\mathcal{O}} B$ *implies* $M * M', \sigma, \sigma' \models_{\mathcal{O}} B'$.

[14] Note that if $M * M'$ is undefined, then the set $\mathcal{A}rising(M * M')$ is empty, and the assertion is trivially satisfied.

[15] TODO:cites

[16] in further work we will refine this to require that A holds only in "observable" or "external" states. Also, if we have constructors, do the invariants hold from the creation of the object, or at the end of constructor? We can deal with this through an extra bit, "ready", and writing he A so that it uses the"ready" bit.

[17] TODO: write this appendix

3. $A, B \rightarrow_M A', A''$ *iff*
$\forall M'.\forall(\_,\sigma),(\_,\sigma') \in \mathcal{A}rising(M*M').\forall sound\ \mathcal{O}.$
$M*M',\sigma \models_{\mathcal{O}} A \ \wedge\ M*M',\sigma,\sigma' \models_{\mathcal{O}} B$ *implies*
$M*M',\sigma \models_{\mathcal{O}} A' \ \wedge\ M*M',\sigma' \models_{\mathcal{O}} A''$
4. $A, A' \rightarrow_M B$ *iff*
$\forall M'.\forall(\_,\sigma),(\_,\sigma') \in \mathcal{A}rising(M*M').\forall sound\ \mathcal{O}.$
$M*M',\sigma \models_{\mathcal{O}} A \ \wedge\ M*M',\sigma' \models_{\mathcal{O}} A'$ *implies* $M*M',\sigma,\sigma' \models_{\mathcal{O}} B.$
5. $B, B' \rightarrow_M B''$ *iff*
$\forall M'.\forall(\_,\sigma),(\_,\sigma') \in \mathcal{A}rising(M*M').\forall sound\ \mathcal{O}.$
$M*M',\sigma,\sigma' \models_{\mathcal{O}} B \ \wedge\ M*M'\sigma',\sigma'' \models_{\mathcal{O}} B'$ *implies*
$M*M',\sigma,\sigma'' \models_{\mathcal{O}} B''$

[18]

The following lemma is an example entailment.

Lemma 9. *For all modules M:*
$\mathcal{M}ay\mathcal{A}ccess(x,y) \wedge \mathcal{M}ay\mathcal{A}ccess(y,z) \rightarrow_M \mathcal{M}ay\mathcal{A}ccess(x,z)$.

We first demonstrate that judgments made in the context of a module are preserved when we link a larger module. In lemma 10, we state that entailment is preserved by linking:

Lemma 10.

- $A \rightarrow_M A'$ *implies that* $A \rightarrow_{M*M'} A'$.
- $B \rightarrow_M B'$ *implies that* $B \rightarrow_{M*M'} B'$
- $A, A' \rightarrow_M B$ *implies that* $A, A' \rightarrow_{M*M'} B$
- $B, B' \rightarrow_M B''$ *implies that* $B, B' \rightarrow_{M*M'} B''$

4.1.2 Disjointness

Definition 19 (Disjointness)**.**

- $M,\sigma \models_{\mathcal{O}} \texttt{stms} \mathbin{\#\!\#} A$ *iff*
$\forall\sigma'.\,[\,M,\sigma \models_{\mathcal{O}} A \ \wedge \sigma' \in \mathcal{R}each(M, \texttt{stmts}, \sigma)$
$\longrightarrow\ M,\sigma' \models_{\mathcal{O}} A\,].$
- $M,\sigma \models_{\mathcal{O}} \texttt{stms} \# A$ *iff*
$\forall\sigma'.\,[\,M,\sigma \models_{\mathcal{O}} A \ \wedge\ M,\sigma,\texttt{stms} \rightsquigarrow \sigma'$
$\longrightarrow\ M,\sigma' \models_{\mathcal{O}} A.\,]$

Note that the first notion is stronger than the second:

Lemma 11. *If $M,\sigma \models_{\mathcal{O}} \texttt{stms} \mathbin{\#\!\#} A$ then $M,\sigma \models_{\mathcal{O}} \texttt{stms} \# A$.*

Proof Follows from Lemma 3. □

For example `x=7` $\#$ `x:=x+1; x:=x-1` holds for all states and modules, but `x=7` $\#\#$ `x:=x+1; x:=x-1` never holds. In general, framing is an undecidable problem, but we can prove some very basic properties, eg that assignment to a variable does not affect all other variables, nor other paths. Note, that in order to express this property we are making use of logical variables.

Lemma 12. *For all modules M, and states σ, for all modules M and oracles $\mathcal{O}$:*

- *If `x` and `y` are textually different variables, then,*
$M,\sigma \models_{\mathcal{O}} \texttt{x=a} \mathbin{\#\!\#} \texttt{y} := \texttt{a'}$.
- *If `x` is not a prefix of the path `p`, then*
$M,\sigma \models_{\mathcal{O}} \texttt{p.f=a} \mathbin{\#\!\#} \texttt{x} := \texttt{a'}$.

4.2 Hoare Rules

We define the Hoare rules in figure 3 for the language constructs, while in figure 4 we give the rules for framing, the rules for consequence, and rules about invariants preserved during execution of a statement.[19]

4.2.1 Hoare Rules – Structural Rules

[20] We first consider the rules from figure 3: The rules (VARASG) and (FIELDASG) are not surpising. The annotations $\_{pre}$ and $\_{post}$ explain the use of $\texttt{a}_{pre}$, and allow us to talk in the postcondition about values in the pre-state. For example, we would obtain

true
  $\{\texttt{this.f=this.f+3}\}$
$\texttt{this.f} = \texttt{this.f}_{pre} + 3$.
$\bowtie$
true

The rules (COND-1) and (COND-2) describe conditional statements, and are standard.

The rule (NEWOBJ) describes the call of a constructor for class `C`[21] . It requires that the class is defined in module M. This means that verified code can only construct objects from the same module. Also, the rule says that is the constructors specification mandates that A needs to hold before the call, and guarantees that B holds after, then indeed, if A holds before the call (where A has been adjusted to hold for the current arguments), then B will hold after the call (again with some adjustments expressed though renamings of the variables).

The rule (METH-CALL-1) describes method call. [22]

(METH-CALL-2) also describes method calls, but is unusual in a Hoare logic setting; it expresses that "only connectivity begets connectivity" . The terms was coined by Mark Miller and is used widely in the capabilities literature. To our knowledge, this property has not been expressed in a Hoare logic. The reason, is, we believe, that Hoare logics so far have been developed with the closed world assumption, in the sense that all methods (or functions) called come from code which has a specification, and which has been verified.

The rule (FRAME-METHCALL) is also unusual; note that its precondition is **true**. This means that we make no as-

[18] TODO: Can we express the definitions more elegantly, as we did for the first one? Also, we we need all four forms?

[19] Notice that we have no rule for object creation; these would like rules for method calls; while they do not pose special challenges, they would increase the size of our system and we leave this to further work.

[20] TODO: is it the right term?

[21] TODO clarify whether the constructor has a body, and where in the spec it has pre- and post- conditions, also whether the constructors have only one parameter - like methods?

[22] We have no invariant part in the spec of a method, but it would not be difficult to extend the system to support this.

$$\frac{\vdash A\,\{\,\texttt{stmts}\,\}\,B \bowtie B' \qquad A \to_M \texttt{stmts}\,\#\,A' \qquad A \to_M \texttt{stmts}\,\#\!\#\,A''}{\vdash A \wedge A'\,\{\,\texttt{stmts}\,\}\,B \wedge A' \bowtie B' \wedge A''}\ (\textsc{frame-general})$$

$$\frac{\vdash A_1\,\{\,\texttt{stmts}\,\}\,B_1 \bowtie B_3 \qquad \vdash A_2\,\{\,\texttt{stmts}\,\}\,B_2 \bowtie B_4}{\vdash A_1 \wedge A2\,\{\,\texttt{stmts}\,\}\,B_1 \wedge B_2 \bowtie B_3 \wedge B_4}\ (\textsc{Conj})$$

$$\frac{\vdash A\,\{\,\texttt{stmts}\,\}\,B \bowtie B' \qquad A' \to_M A \qquad B \to_M B'' \qquad B' \to_M B'''}{\vdash A'\,\{\,\texttt{stmts}\,\}\,B'' \wedge B' \bowtie B'''}\ (\textsc{Cons-1})$$

$$\frac{\vdash A\,\{\,\texttt{stmts}\,\}\,B \bowtie B'' \qquad A', B' \to_M A, \mathbf{true}}{\vdash A'\,\{\,\texttt{stmts}\,\}\,B' \to B \bowtie B''}\ (\textsc{Cons-2})$$

$$\frac{M(S) \equiv \ \mathbf{spec}\ S\ \{\ \overline{\mathit{Policy}}, P, \overline{\mathit{Policy}'}\ \}}{\vdash \mathbf{true}\,\{\,\texttt{stmts}\,\}\,\mathbf{true} \bowtie \texttt{x}\,\mathbf{obeys}\,S \to P[\texttt{this}/\texttt{x}]}\ (\textsc{code-invar-1})$$

$$\frac{}{\vdash e\,\mathbf{obeys}\,S\,\{\,\texttt{stmts}\,\}\,\mathbf{true} \bowtie e_{pre}\,\mathbf{obeys}\,S}\ (\textsc{code-invar-2})$$

$$\frac{x \text{ does not appear in } A, \text{ or } B, \text{ or } \texttt{stmts} \qquad \vdash A\,\{\,\texttt{stmts}\,\}\,B' \bowtie B}{\vdash A\,\{\,\texttt{stmts}\,\}\,\forall x.B' \bowtie B}\ (\textsc{quantifiers-1})$$

$$\frac{x \text{ does not appear in } A, \text{ or } B \text{ or } \texttt{stmts} \qquad \vdash A\,\{\,\texttt{stmts}\,\}\,B \bowtie B'}{\vdash A\,\{\,\texttt{stmts}\,\}\,B \bowtie \forall x.B'}\ (\textsc{quantifiers-2})$$

Figure 4. Hoare Logic – we assume that the module M is globally given

sumptions about the receiver of the method call; this allows us to reason in an *open* setting. Even though we do not know what the behaviour method `m` will be, we still have some conditions which can guarantee that A' will be preserved. These conditions are that anything that was accessible from the receiver `x` or argument of `z` at the time of the method call, or anything that is newly created during execution of the method body, does not satisfy the prerequisites necessary to affect A'.[23]

The last rule in figure 3 is (SEQUENCE). It requires that the precondition and the postcondition of the first statements, *i.e.* A and B_1, imply the precondition of the second statements, ie A_2, and that the combined effects described by the two-state assertion in the postconditions of $\texttt{stmts}_1$ and $\texttt{stmts}_2$, B_1 followed by B_2, imply the postcondition of the sequence, *i.e.* B.

4.2.2 Hoare Rules – Substructural Rules

24

We now turn our attention to the substructural rules from figure 4.

Rule (FRAME-GENERAL) allows us to frame onto a tuple any assertion that has not been affected by the code. The rule (CONJ) allows us to combine different Hoare tuples for the same code, and follows standard Hoare logics.

Interestingly, our system has *two* rules of consequence. The fist rule, (CONS-1), is largely standard, as it allows us to strengthen the precondition A, and weaken the postcondition Bas well as the invariant B'. A novelty of this rule, however, is that it allows the invariant to be conjoined to the postcondition; this is sound, because the invariant is guaranteed to hold throughout execution of the code, and thus also after it.

The next rule, (CONS-2), is unusual, in that it allows us to *weaken* the precondition, while adding a hypothesis B' to the postcondition, such that the original postcondition, B, is only guaranteed if B' holds. The rule is sound, because we also require that the new precondition A' together with the new postcondition B' guarantee that the original precondition holds in the pre-state. The judgment $A, B \to_M A', A''$ is defined in in Definition 18. For example, we can use this rule to take

$\texttt{p1}$ **obeys** Purse
  $\{\,\texttt{p2:=p1.sprout}\}$
$\texttt{p2}$ **obeys** Purse
$\bowtie$
true

and deduce that

true
  $\{\,\texttt{p2:=p1.sprout}\}$
$\texttt{p1}_{pre}$ **obeys** $\mathit{Purse} \to \texttt{p2}$ **obeys** Purse .
$\bowtie$
true

The two last rules in 4 are concerned with adherence to specification.

The rule (CODE-INVAR-1) expresses that throughout execution of any code, in all intermediate states, for any variable `x` for which we know that it **obeys** a specification S, we know that it satisfies any of S's stated policies.

[23] Notes that $\sigma' \in \mathcal{R}each(M, \sigma, \texttt{stmts})$ is a shorthand for $\sigma'.(\sigma', \_') \in \mathcal{R}each(M, \sigma, \texttt{stmts})$.

[24] TODO: is it the right term?

The rule (CODE-INVAR-2) guarantees that any term e which has been shown to be pointing to an object which **obeys** a specification S will continue satisfying the specification throughout execution of any `stms`.

4.3 Soundness

In lemma 13 we state that derivability of Hoare tuples is preserved for larger modules.[25]

Lemma 13 (Linking preserves derivations). *For all modules M, M', if $M * M'$ is defined, then:*

- *If $M \vdash A\,\{\,\texttt{stms}\,\}\,A' \bowtie B$, then $M{*}M' \vdash A\,\{\,\texttt{stms}\,\}\,A' \bowtie B$.*

We will now state and prove soundness of the Hoare logic. A prerequisite for the derivation of a Hoare tuple to be semantically valid is that all the methods in the module used to derive the tuple have been checked by the Hoare logic to satisfy their specifications. Thus we come to the concept of a "checked" module, expressed as $\vdash M$. We say that a module has been checked if all classes in M have been checked to satisfy all the policies from their specification.

Definition 20 (Checked modules and classes). *We define the following judgments:*

- *$M, C \vdash A\{\ res = this.m(par)\ \}\ B$ iff we can prove that*
  *$M \vdash A \wedge$ **this** $: C\,\{\,stmts\,\}\,B[a/res] \bowtie$ **true***
  where
  *$\mathcal{M}(M, C, m) =$ **method** $m($ par $)$ $\{$ $stmts$; **return** a $\}$.*
- *A similar requirement to be added if we decide to include constructors, ie something like*
  $M, C \vdash A\{\ res = \mathbf{new}(x_1, ...x_n)\ \}\ B$ iff
  we can prove that
  *$M \vdash A\,\{\,stmts\,\}\,B[a/res] \bowtie$ **true***
  where
  *$\mathcal{M}(M, C, new) =$ **new**$($ $x_1, ...x_n)$ $\{$ $stmts$; **return** a $\}$*
- *$M \vdash C : S$ iff for all $Pol \in \mathcal{S}(M, C)$[26]*
  - *If Pol has the form $A\{\ res = this.m(par)\ \}\ B$, or $A\{\ res = \mathbf{new}(x_1, ...x_n)\ \}\ B$ then $M, C \vdash Pol$.*
  - *If Pol has the form A, then for all sound $\mathcal{O}$, we have $M \models_{\mathcal{O}} C : A$.*
- *$\vdash M$ iff*
  $M \vdash C : S$ for all C, S with $S \in \mathcal{S}(M, C)$.

Notice that in the above the policies that pertain to method or constructor calls are checked with the Hoare logic, while the invariants A are required to hold semantically. This is so, because we do not – yet – have a Hoare logic for proving invariants. But we are planning to construct one, and in the appendix ??? we show how such reasoning about invariants can work – also mention the works on reasoning about invariants.

We will now define and prove the soundness of our Hoare logic.

Theorem 1 (Soundness of the Hoare Logic). *Take any modules $M$and M', any $\mathcal{O}$, code $\texttt{stmts}$, assertions A, A' and B and B', and any oracle $\mathcal{O}$. Then, if*
1. $\vdash M$, and
2. $\mathcal{O}$ is sound for all classes defined outside module M, and $\mathcal{O}(M, C, S) = true$ for any C, S with $S \in \mathcal{S}(M, C)$
3. $M \vdash A\,\{\,\texttt{stms}\,\}\,B \bowtie B'$, and
4. $(\texttt{stmts}, \sigma) \in \mathcal{A}rising(M' * M)$
5. $M' * M, \sigma \models_{\mathcal{O}} A$,
6. $M' * M, \sigma, \texttt{stms} \rightsquigarrow \sigma'$
then
1. $M' * M, \sigma, \sigma' \models_{\mathcal{O}} B$, and
2. $\forall \sigma'' \in \mathcal{R}each(M' * M, \sigma, \texttt{stmts}).\ M{*}M', \sigma, \sigma'' \models_{\mathcal{O}} B'$

Note that while requirement 1. and 3. above only talk of module M which has been checked, and which is used to prove the tuple $M \vdash A\,\{\,\texttt{stms}\,\}\,B' \bowtie B$, the other requirements (4.. 5. and 6.) and the conclusions (1. and 2.) are made in the context of the larger module $M' * M$. This reflects the open setting[27] of our work, where modules are so robust, t hat they can guarantee properties in the context of unknown, and possibly malicious other code.

Requirement 2. above requires that the oracle $\mathcal{O}$ is sound for all classes except those defined in module M. This does not mean that it should be unsound for M – merely that we will not be using the soundness of $\mathcal{O}'$ to prove the method calls of methods defined in M. In fact, our theorem implies that for any oracle $\mathcal{O}$ such that $\mathcal{O}(M, C, S) = true$ for any C,S with $S \in \mathcal{S}(M, C)$, if the module M has been checked, ie $\vdash M$, then $\mathcal{O}$ is sound.

Proof. THE PROOF NEEDS REVISION. BUT I THINK IT GOES THROUGH. We fix the modules M and M'.

The proof proceeds by well-founded induction. We define a well-founded ordering $\prec$ which orders tuples of states, statements, one-state assertions, and two two-state assertions, ie

$$\prec\ \subseteq\ (\ state\ \times\ Stmts\ \times\ OneStateAssert\ \times\ TwoStateAssert\ \times\ TwoStateAssert\ )^2$$

This ordering $\prec$ is the smallest relation which satisfies the following two requirements[28]

For all σ, σ' `stmts`, `stmts'` ,A, B, B', A', B'', B''':

If $M * M', \sigma, \texttt{stmts} \rightsquigarrow \sigma''$ in fewer steps than $M * M', \sigma', \texttt{stmts'} \rightsquigarrow \sigma'''$[29], then

$$(\sigma, \texttt{stmts}, A, B, B') \prec (\sigma', \texttt{stmts'}, A', B'', B''')$$

If the proof of $M \vdash A\,\{\,\texttt{stms}\,\}\,B \bowtie B'$ requires the proof of $M \vdash A'\,\{\,\texttt{stms}\,\}\,B'' \bowtie B'''$ through one of the steps

[25] SD thinks that this lemma is unnecessary now what we have $\mathcal{O}$'s. But it is still a "sanity check".

[26] Here we are using S wrongly – we need somethoign that looks up the policies in the spec – easy to fix

[27] good term?

[28] need to express better

[29] This should be expressed better, but is clear

from Figure 4[30], then

$(\sigma, \texttt{stmts}, A, B, B') \prec (\sigma, \texttt{stmts}, A', B'', B''')$

We now argue that the relation is well-founded, ie there are no cycles. *** some work here ***

We proceed by case analysis on the last step in the derivation of $M \vdash A\,\{\,\texttt{stms}\,\}\,B' \bowtie B$.

Case (VARASG), (FIELDASG), (FIELDASG), (COND-1) and (COND-2) all follow from the operational semantics of $\mathcal{F}ocal$; the latter two cases also require application of the induction hypothesis.

Case (METH-CALL-1). This gives that

5. $\texttt{stmts}$ has the form $\texttt{v:=x.m(y)}$, and that

6. $A \equiv x\,\textbf{obeys}\,\texttt{S} \wedge A'[\texttt{x/this}, \texttt{y/par}]$, where

7. $M(S) = \textbf{specification}\,\texttt{S}\,\{\,...,A'\{\,\texttt{this.m(par)}\,B'',...\}$, and where

8. $B \equiv B''[\texttt{x/this}, \texttt{y/par}, \texttt{v/res}]$, and

9. $B' \equiv \textbf{true}$.

From 5. and the operational semantics we obtain that

10. $M * M', \phi' \cdot \chi, \texttt{stmts'} \rightsquigarrow \sigma''$, where

11. $\mathcal{M}(\texttt{M},\texttt{C},\texttt{m}) = ..\{\,\texttt{stmts'};\textbf{return}\,a\,\}$, and

12. $\sigma' = \sigma''[\texttt{v} \mapsto \lfloor a \rfloor_{\sigma''}]$ . ... More steps here ...

From 6. and 3., and by definition ???, we obtain that

yy. $\sigma(x) \downarrow_1 = \texttt{C}$, and

vv. $\texttt{C}$ is defined to satisfy $\texttt{S}$.

From 7, vv, and because of 1. we also obtain that

zz. $M \vdash \texttt{this : C} \wedge A\,\{\,\texttt{stms'}\,\}\,B''[\texttt{a/res}] \bowtie \textbf{true}$

From 10, and .. we obtain that

uu. $(\phi' \cdot \chi, \texttt{stms''}, x\,\textbf{obeys}\,\texttt{S} \wedge A'[\texttt{x/this}, \texttt{y/par}], ..., ...) \prec (\sigma, \texttt{stms}, A, B', B)$.

Therefore, by application of inductive hypothesis, we obtain ... more here ..

Case (METH-CALL-2) Follows from lemmas 7 and 8.

Case (FRAME-METHCALL) needs work ***FIXME:*** ***Are we missing the entailment between $\mathcal{M}ay\mathcal{A}ffect$ and $\mathcal{M}ay\mathcal{A}ccess$?*** I think we need a lemma which says, what you, Toby had written earlier, ie that expresses a basic axiom of object-capability languages, namely that in order to cause some visible effect, one must have access to an object able to perform the effect. But we need to formulate this lemma, you are right!

Case (SEQUENCE) follows from the definition of $\mathcal{R}each(M, \sigma, \texttt{code}_1\texttt{;}\ \texttt{code}_2)$ and the definition of validity of Hoare tuples (**??**).

Case (FRAME-GENERAL) Follows by the definition of $\#$ and $\#\!\#$.

Case (CONS-1) follows from the definition of entailment (Definition 18) and the fact that $(\sigma, \texttt{stms}) \in \mathcal{R}each(M, \sigma, \texttt{stms})$.

Case (CONS-2) follows because $\sigma, \sigma' \models_{\mathcal{O}} Q' \rightarrow Q$ if and only iff $\sigma, \sigma' \models_{\mathcal{O}} Q$ assuming $\sigma, \sigma' \models_{\mathcal{O}} Q'$.

Case (CONS-3) and (CONS-4) follow straightforwardly from the definition of entailment and Hoare tuple validity.

Case (CODE-INVAR-1) follows because the definition of policy satisfaction for one-state-assertions A requires that A holds for all internally-reachable states σ' via $\mathcal{R}each$.

Case (CODE-INVAR-2) follows straightforwardly from the definition of Hoare tuple validity and 2-state-assertion validity.

□

5. Illuminating Examples

Here we consider a suite of examples that shed light on the various subtle aspects of our approach.

5.1 Unknown Assertions

Consider the following module definition.

```
module M₁
     ≡
specification S₁{
  }
class C₁ satisfies S₁{
    }
module M₂
     ≡
class C₂ satisfies S₁{
    }
specification S₁{
  }
}
```

Consider now some mappings $\mathcal{O}_1$, $\mathcal{O}_2$ such that

$\mathcal{O}_1(\texttt{M}_1, \texttt{C}_1, \texttt{S}_1) = true$, and

$\mathcal{O}_1(\texttt{M}_1 * \texttt{M}_2, \texttt{C}_1, \texttt{S}_1) = false$, and

$\mathcal{O}_2(\texttt{M}_1, \texttt{C}_1, \texttt{S}_1) = false$, and

$\mathcal{O}_2(\texttt{M}_1 * \texttt{M}_2, \texttt{C}_1, \texttt{S}_1) = true$.

Neither $\mathcal{O}_1$, nor $\mathcal{O}_2$ are oracles, since they do not satisfy the requirements from Definition 11.

Now onsider now some mappings $\mathcal{O}_3$, $\mathcal{O}_4$ such that

$\mathcal{O}_3(\texttt{M}_1 * M', \texttt{C}_1, \texttt{S}_1) = true$ for all M';[31]

$\mathcal{O}_3(\texttt{M}_2 * M', \texttt{C}_2, \texttt{S}_1)$ for all M';

$\mathcal{O}_3(\_,\_,\_) = false$, otherwise.

$\mathcal{O}_4(\texttt{M}_1 * M', \texttt{C}_1, \texttt{S}_1) = false$ for all M';

$\mathcal{O}_3(\texttt{M}_2 * M', \texttt{C}_2, \texttt{S}_2) = true$ for all M';

$\mathcal{O}_4(\_,\_,\_) = false$, otherwise.

Both $\mathcal{O}_3$ and $\mathcal{O}_4$ are oracles. Note that $\mathcal{O}_4$ treats $\texttt{M}_1$, $\texttt{C}_1$, and $\texttt{S}_1$ differently from $\texttt{M}_2$, $\texttt{C}_2$, and $\texttt{S}_2$, even though they are structurally isomorphic. Nevertheless, $\mathcal{O}_4$ is an oracle. Moreover, both $\mathcal{O}_3$ and $\mathcal{O}_4$ are *sound* oracles.

We now consider the application of these oracles to judge validity. Assume a state σ_1 such that $\mathcal{C}lass(x)_{\sigma_1} = \texttt{C}_1$, and $\mathcal{C}lass(x)_{\sigma_1} = \texttt{C}_2$. Then we have that

$M_1, \sigma_1 \models_{\mathcal{O}_4} x\,\textbf{obeys}\,\texttt{S}_1$, while

[30] DANGEROUS, need to know that we cannot introduce cycles! but should be doable

[31] We assume that there also exist ab empty module, so as to also obtain $\mathcal{O}_3(\texttt{M}_1, \texttt{C}_1, \texttt{S}_1) = true$

$M_1, \sigma_1 \models_{\mathcal{O}_4} y$ **obeys** $\mathtt{S}_1$ is unknown. [32]
On the other hand:
$M_2 * M_1, \sigma_1 \models_{\mathcal{O}_3} x$ **obeys** $\mathtt{S}_1 \rightarrow y$ **obeys** $\mathtt{S}_1$, but
$M_2 * M_1, \sigma_1 \not\models_{\mathcal{O}_4} x$ **obeys** $\mathtt{S}_1 \rightarrow y$ **obeys** $\mathtt{S}_1$.
Therefore, we have that
$\models x$ **obeys** $\mathtt{S}_1 \not\longrightarrow_{M_2 * M_1} y$ **obeys** $\mathtt{S}_1$.

5.2 Infinite execution

Consider the following module definition.

```
module M₁
      ≡
specification S₁{
   policy P₁:
      true
         { this.m() }
      false
}
class C₁ satisfies S₁{
    method m(){
        this.m()
    }
}
```

Specification $\mathtt{S}_1$ is clearly nonsensical, as it asserts the unsatisfiable postcondition `false`. However, perhaps surprisingly, a mapping $\mathcal{O}_1$ such that $\mathcal{O}_1(\mathtt{M}_1 * M', \mathtt{C}_1, \mathtt{S}_1) = \mathit{true}$ for all modules M', and $\mathcal{O}_1(\_,\_,\_) = \mathit{false}$, otherwise, is an oracle, and moreover is a *sound* oracle! This circularity is not a soundness issue, since `m()` never terminates. Our logic only requires *partial correctness*.

5.3 Contradiction in the Specification

Our specification language allows one to express apparently contradictory specifications, such as the following.

```
module M₂
      ≡
specification S₂{
   policy P₂:
      ¬ this obeys S₂
}
class C₂ satisfies S₂{
    ...
}
```

Note that while class $\mathtt{C}_2$ claims to satisfy $\mathtt{S}_2$, the contradiction in policy $\mathtt{P}_2$ means that we would be very surprised if there was a possibility to satisfy it. Consider any oracle $\mathcal{O}_1$ such that $\mathcal{O}_1(\mathtt{M}_2, \mathtt{C}_2, \mathtt{S}_2) = \mathit{true}$ – such an oracle is *not* sound.

Here is why: Assume a state σ_1 such that $\mathcal{C}lass(x)_{\sigma_1} = \mathtt{C}_2$, and assume that $\mathcal{O}_1$ was sound. Then, by definition of $\models$, we obtain that $M_2, \sigma_1 \models_{\mathcal{O}_1} x$ **obeys** $\mathtt{S}_2$. Because $\mathcal{O}_1$ is sound, we can unfold the definition of $\mathtt{S}_2$, and ontain $M_2, \sigma_1 \not\models_{\mathcal{O}_1} x$ **obeys** $\mathtt{S}_2$. This is a contradiction, and therefore, $\mathcal{O}_1$ is not sound.

Note that we do not obtain a contradiction if we have an oracle $\mathcal{O}_2$ such that $\mathcal{O}_2(\mathtt{M}_2, \mathtt{C}_2, \mathtt{S}_2) = \mathit{false}$. Namely, we are not allowed the fold the definition, and thus we cannot obtain that $M_2, \sigma_1 \models_{\mathcal{O}_2} x$ **obeys** $\mathtt{S}_2$.

5.4 Monotonicity and Invariants

Consider the following module $\mathtt{M}_3$ that declares a single specification $\mathtt{S}_3$ and class $\mathtt{C}_4$ that claims to satisfy $\mathtt{S}_3$.

```
module M₃
      ≡
specification S₃{ }
specification S₄
  policy Pol_1
       ∀ o. o:Object. o obeys S₃
}
class C₃ satisfies S₃{
    ...
}
```

In the context of this module *alone* all objects will be of class $\mathtt{C}_4$, and so each will obey $\mathtt{S}_3$. However, the assertion $\mathtt{M}_3 \models_{\mathcal{O}} \mathtt{C}_4 : \forall o : Object.\ o$ **obeys** $\mathtt{S}_3$ does *not* hold for *any* oracle $\mathcal{O}$. This is so, because Definition 16 requires that the assertion holds in all configuration arising from all possible extensions of $\mathtt{M}_3$, by linking $\mathtt{M}_3$ against all possible M'. Naturally some of these M' not contain the definition of $\mathtt{S}_3$, and by Definition 11 the oracle $\mathcal{O}$ will say that $\mathcal{O}(M, \mathtt{C}) = \mathit{false}$ for any class $\mathtt{C}$ defined in M'.

5.5 Negative Positions

Quite often[33], we need to restrict the assertions appearing in negative positions. But this is not necessary here. Nevertheless, in general, when **obeys** appears in a negative position in the postcondition of a method specification, the specification is unsatisfiable.

In the example below, x **obeys** $\mathtt{S}_4$ appears in a negative position.

```
module M₄
      ≡
specification S₄{
     policy Pol₄ₐ
         true
             { res=this.m(x) }
         x  obeys  S₄  ⟶  res

     policy Pol₄ᵦ
         true
             { res=this.m(x) }
         x  obeys  S₄  ⟵  res
}
class C₄ satisfies S₄{
    method m(x){ ... }
}
```

[32] If we adopted Alex's clever definition however, we would have $M_1, \sigma_1 \models y$ **obeys** $\mathtt{S}_1 \rightarrow x$ **obeys** $\mathtt{S}_1$ holds.– Still to think whether we want that.

[33] TODO find citations

In general, it is impossible to satisfy $\texttt{Pol}_{4a}$ unless the method `m` always returns true. Namely, this specification requires that the method should recognize all objects which satisfy $\texttt{S}_4$. Since there are many classes that can satisfy this specification, and not all such classes are known to the module M_4, it is impossible to write a method body for `m` such that it satisfies this requirment.

Policy $\texttt{Pol}_{4b}$ is even more difficult to satisfy.

However, in the below we strengthen the specification $\texttt{S}_4$ from he example above, so that $\texttt{Pol}_{4a}$ became satisfiable. Namely, we added the policy $\texttt{Pol}_{4c}$ which requires that objects which satisfy $\texttt{Spec}_4$ return *true* when executing method `check()`. On the other hand, policy $\texttt{Pol}_{4b}$ remains unsatisfiable.

```
module M$_{4a}$
     ≡
specification S$_4${
     policy Pol$_{4a}$
         true
             { res=this.m(x) }
         x obeys S$_4$ ⟶ res

     policy Pol$_{4b}$
         true
             { res=this.m(x) }
         x obeys S$_4$ ⟵ res

     policy Pol$_{4c}$
         true
             { res=this.test() }
        res
}
class C$_4$ satisfies S$_4${
    method m(x){ z=x.check(); return z }
}
```

On the other hand, the following spec is satisfiable (in fact, we have similar policies for Purse-s). Here we have an abstract predicate `AP` on the right hand side of the implication. Since the abstract predicate will be made concrete by the module, it can be defined in such a way as to make it possible to satisfy the specification. Note that the predicate `AP` is a *binary* relation, while **obeys** is unary.

```
module M$_5$
     ≡
specification S$_5${
     abstract predicate AP(o1,o2)

     policy Pol$_5$
         this.AP(o) → o obeys S$_5$

     policy Pol$_6$
         true
             { res this.m(x) }
        this.AP(o)  ⟷ res
}
class C$_5$ satisfies S$_5${
  ...
}
```

5.6 Cyclic Definitions

```
module M$_6$
     ≡
specification S$_6${
     field next
     policy Pol$_6$
         this.next≠ null → this.next obeys S$_7$
}
specification S$_7${
     field next
     policy Pol$_7$
         this.next≠ null → this.next obeys S$_6$
}
private class C$_6$ satisfies S$_6$ {
    field next
    method m( ){ this.next = new C$_7$(null) }
}
private class C$_7$ satisfies S$_7$ {
    field next
    method m( ){ this.next = new C$_6$(null) }
}
class D{
  ... new C$_6$(null)
  ...
}
```

The specifications $\texttt{S}_6$ and $\texttt{S}_7$ are cyclic. An oracle $\mathcal{O}$ such that $\mathcal{O}(M_6, \texttt{C}_6, \texttt{S}_6) = \mathit{true}$ is sound only if in each reachable state σ, if $\mathcal{C}lass(x)_\sigma) = \texttt{C}_6$ and $\lfloor x.next \rfloor_\sigma \neq \mathbf{null}$, then $\mathcal{O}(M_6, \texttt{C}, \texttt{S}_7) = \mathit{true}$ where $\texttt{C} = \mathcal{C}lass(\texttt{x.next})_\sigma$. In the particular case, given the code of the classes $\texttt{C}_6$, this amounts to requiring that $\mathcal{O}(M_6, \texttt{C}_7, \texttt{S}_7)$.

```
module M$_8$
     ≡
specification S$_8${
     policy Pol$_8$
         ∀x. x obeys S$_9$  → x obeys S$_9$
}
specification S$_9${
     policy Pol$_9$
      ... some unsatisfiable requirement ....
}
class C$_{surprise}$ satisfies S$_8$ {
}
```

Perhaps unsurprisingly, there exists a sound oracles $\mathcal{O}$, such that $\mathcal{O}(\texttt{M}_8, \texttt{C}_{surprise}, \texttt{S}_8) = \mathit{true}$,

References

[1] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Baltimore, Maryland, 2006.