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

Sophia Drossopoulou[1], James Noble[2], Mark Miller[3], Toby Murray[4],
[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 $\mathcal{F}ocal$, *Chainmail*, our Hoare logic, prove soundness of our Hoare logic, and then prove that our escrow exchange implementation establishes mutual trust while managing risk.

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

2.1 Modules and Linking

$\mathcal{F}ocal$ modules map class identifiers to class descriptions, function identifiers to function descriptions, and predicate identifiers to predicate descriptions. We also require implicitly for any module M, class identifier c, function identifier f, and predicate identifier P, that that $M(c) \in ClassDescr$ or undefined, that $M(f) \in FunDescr$ or undefined, and $M(P) \in PredDescr$ or undefined.

Definition 1 (Modules)**.**

$$\begin{array}{rl} \mathit{Module} & = \mathit{ClassId} \cup \mathit{FunId} \cup \mathit{PredId} \cup \mathit{SpecId} \\ & \longrightarrow \\ & (\ \mathit{ClassDescr} \cup \mathit{FuncDescr} \cup \mathit{PredDescr} \\ & \ \cup \mathit{Specification}\ ) \end{array}$$

We define linking of modules, $M * M'$, to be the union of their respective mappings, provided that the domains of the two modules are disjoint:

Definition 2 (Linking and Lookup)**.** *Linking of modules M and M' is*

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

$$M*M' = \begin{cases} M *_{aux} M', & \text{if } dom(M) \cap dom(M') = \emptyset \\ \bot & \text{otherwise.} \end{cases}$$

$$(M *_{aux} M')(c) = \begin{cases} M(id), & \text{if } M(id) \text{ is defined} \\ M'(id) & \text{otherwise.} \end{cases}$$

Classes We define the syntax ....

Definition 3 (Classes, Methods, Args)**.** *We define the syntax of modules below.*

$$\begin{array}{lcl} \mathit{ClassDescr} & ::= & \textbf{class}\ \mathit{ClassId} \\ & & \quad \{\ (\textbf{fld}\ \mathit{FieldId})^*\ (\ \mathit{methBody}\ )^*\ \} \\ \mathit{methBody} & ::= & \textbf{method}\ m\ (\ \mathit{ParId}^*\ ) \\ & & \quad \{\ \mathit{Stmts}\ \textbf{;}\ \textbf{return}\ \mathit{Arg}\ \} \\ \mathit{Stmts} & ::= & \mathit{Stmt} \mid \mathit{Stmt}\ \textbf{;}\ \mathit{Stmts} \\ \mathit{Stmt} & ::= & \textbf{var}\ \mathit{VarId}\ \textbf{:=}\ \mathit{Rhs} \\ & \mid & \mathit{VarId}\ \textbf{:=}\ \mathit{Rhs} \\ & \mid & \textbf{this}.\mathit{FieldId}\ \textbf{:=}\ \mathit{Rhs} \\ & \mid & \textbf{if}\ \mathit{Arg}\ \textbf{then}\ \mathit{Stmts}\ \textbf{else}\ \mathit{Stmts} \\ & \mid & \textbf{skip} \\ \mathit{Rhs} & ::= & \mathit{Arg}.\mathit{MethId}(\ \mathit{Arg}^*\ ) \mid \mathit{Arg} \\ & \mid & \textbf{new}\ \mathit{ClassId}(\ \mathit{Arg}^*\ ) \\ \mathit{Arg} & ::= & \mathit{Path} \mid \textbf{true} \mid \textbf{false} \mid \textbf{null} \\ \mathit{Path} & ::= & \mathit{ParId} \mid \mathit{VarId} \mid \textbf{this} \\ & \mid & \mathit{Path}.\ \mathit{FieldId} \end{array}$$

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.

Method Lookup We define the method lookup function, $\mathcal{M}$ which returns the corresponding method definition given a class and a method identifier.

Definition 4 (Lookup)**.** *The lookup function*

$\mathcal{M}(M, c, m) = \textbf{method}\ m\ (\ p_1, ...p_n)\ \{\ stms\textbf{;}\ \textbf{return}\ a\}$
iff $M(c) = \textbf{class}\ c\{\ ...$
$\textbf{method}\ m\ (\ p_1, ...p_n)\ \{\ stms\textbf{;}\ \textbf{return}\ a\}$
$...\}$.
undefined, otherwise.

2.2 Execution of $\mathcal{F}ocal$

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.

(METHCALL_OS)

$$\frac{\lfloor a \rfloor_{\phi\cdot\chi} = \iota \quad \forall i \in \{1..n\}.\ \ \lfloor a_i \rfloor_{\phi\cdot\chi} = val_i \quad \mathcal{M}(\mathtt{M}, \chi(\iota)\downarrow_1, \mathtt{m}) = \mathbf{method}\ m(\ par_1, \ldots par_n)\ \{\ stms\mathbf{;\ return}\ a'\} \quad \phi'' = \mathbf{this} \mapsto \iota, par_1 \mapsto val_1, \ldots par_n \mapsto val_n \quad M,\ \phi''\cdot\chi, stmts\ \rightsquigarrow\ \phi'\cdot\chi'}{M,\ \phi\cdot\chi,\ a.m(\ a_1, \ldots a_n)\ \rightsquigarrow\ \chi', \lfloor a' \rfloor_{\phi'\cdot\chi'}}$$

(ARG_OS)

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

(NEW_OS)

$$\frac{\iota \text{ is new in } \chi \quad \mathtt{f}_1, \ldots \mathtt{f}_n \text{ are the fields defined in } C}{M,\ \phi\cdot\chi,\ \mathbf{new}\ C(\ a_1, \ldots a_n) \ \rightsquigarrow\ \chi[\iota \mapsto (C, \mathtt{f}_1 \mapsto \lfloor a_1 \rfloor_{\phi,\sigma} .. \mathtt{f}_n \mapsto \lfloor a_n \rfloor_{\phi,\sigma})]}$$

(VARASG-1_OS)

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

(VARASG-2_OS)

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

(FIELDASG_OS)

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

(SEQUENCE_OS)

$$\frac{M,\ \sigma, stmt\ \rightsquigarrow\ \sigma'' \quad M,\ \sigma'',\ stmts\ \rightsquigarrow\ \sigma'}{M,\ \sigma,\ stmt\ \mathbf{;}\ stmts\ \rightsquigarrow\ \sigma'}$$

(COND-TRUE_OS)

$$\frac{\lfloor a \rfloor_\sigma = \mathbf{true} \quad M,\ \sigma,\ stmts_1\ \rightsquigarrow\ \sigma'}{M,\ \sigma,\ \mathbf{if}\ a\ \mathbf{then}\ stmts_1\ \mathbf{else}\ stmts_2\ \rightsquigarrow\ \sigma'}$$

(COND-FALSE_OS)

$$\frac{\lfloor a \rfloor_\sigma = \mathbf{false} \quad M,\ \sigma,\ stmts_2\ \rightsquigarrow\ \sigma'}{M,\ \sigma,\ \mathbf{if}\ a\ \mathbf{then}\ stmts_1\ \mathbf{else}\ stmts_2\ \rightsquigarrow\ \sigma'}$$

(SKIP_OS)

$$\frac{}{M,\ \sigma,\ \mathbf{skip}\ \rightsquigarrow\ \sigma}$$

Figure 1. Operational Semantics

$\sigma \in$ 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 5 (Interpretation)**.** *For a state $\sigma = (\phi, \chi)$ we define*

$$\begin{array}{lcl} \lfloor \mathbf{null} \rfloor_\sigma & = & \mathbf{null} \\ \lfloor \mathbf{true} \rfloor_\sigma & = & \mathbf{true} \\ \lfloor \mathbf{false} \rfloor_\sigma & = & \mathbf{false} \\ \lfloor x \rfloor_\sigma & = & \phi(x) \quad (\text{for x} \in StackId) \\ \lfloor p.f \rfloor_\sigma & = & \chi(\lfloor p \rfloor_\sigma)(f) \qquad \textit{is} \lfloor p \rfloor_\sigma \textit{ is defined,} \\ & & \textit{undefined,} \qquad \textit{otherwise} \end{array}$$

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

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

Arising and Reachable 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.

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

$$\mathcal{R}each\ :\ Module \times state \times Stmts \longrightarrow \mathcal{P}(state \times Stmts)$$

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 $\mathcal{R}each(M, \sigma, \mathtt{stmts})$ collects all configurations reachable during execution of $\sigma, \mathtt{stmts}$.

Note that the function $\mathcal{R}each(M, \sigma, \texttt{stmts})$ is defined, even when the execution should diverge. This is important, because it allows us to give meaning to capability policies without requiring termination.

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

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

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

as follows:

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

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.

3. The Specification Language Chainmail

Our specifications and policies are fundamentally 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 $\sigma(x)$=4, and $\sigma'(x)$=3, we have $M, \sigma, \sigma' \models x_{pre} - x_{post} = 1$.

Expressions and Assertions We first define expressions, Expr, and assertions A, 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 the function `length` of a list). We use the identifier sR to indicate predicates whose validity depends on the state (eg the predicate `Acyclic` for a list).

The difference between expressions and arguments is that expressions may express 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.

Definition 8 (Expressions)**.**

$$\begin{array}{lcl} \textit{Expr} & ::= & \textit{Arg} \mid \textit{Val} \mid \textit{Expr} + \textit{Expr} \mid \ldots \\ & \mid & \textit{f(Expr*)} \\ & \mid & \textbf{if}\ \textit{Expr}\ \textbf{then}\ \textit{Expr}\ \textbf{else}\ \textit{Expr} \\ \textit{funDescr} & ::= & \textbf{function}\ \textit{f}(\ \textit{ParId*}\ )\ \{\ \ \textit{Expr}\ \} \end{array}$$

We now define the values of such expressions, and the validity of one-state assertions as follows:

Definition 9 (Interpretations)**.** *We define the interpretation of expressions,* $\lfloor \cdot \rfloor : Expr \times Module \times state \to Value$ *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* $a \in Arg$.
- $\lfloor e_1 + e_2 \rfloor_{M,\sigma}$ = $\lfloor e_1 \rfloor_{M,\sigma} + \lfloor e_2 \rfloor_{M,\sigma}$.
- $\lfloor f(e_1, \ldots e_n) \rfloor_{M,\sigma}$ = $\lfloor Expr[e_1/p_1, \ldots e_n/p_n] \rfloor_{M,\sigma}$ *where* $M(f) = \textbf{function}\ f\ (\ p_1 \ldots p_n)\ \{\ \ Expr\ \}$*, undefined, otherwise.*
- $\lfloor \textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \rfloor_{M,\sigma}$ $= \lfloor\ e_1 \rfloor_{M,\sigma}$*, if* $\lfloor\ e_0 \rfloor_{M,\sigma}$=**true**, $= \lfloor\ e_2 \rfloor_{M,\sigma}$*, if* $\lfloor\ e_0 \rfloor_{M,\sigma}$=**false**. *and undefined, otherwise.*

One-state assertions We now define a language of assertions which depend on one state. We introduce three specific predicates: $\mathcal{M}ay\mathcal{A}ffect$ and $\mathcal{M}ay\mathcal{A}ccess$ which we use to model risk, the assertion $Expr{:}ClassId$ which expresses class membership, and the assertion $Expr$ **obeys** $SpecId$. 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 $\mathcal{M}ay\mathcal{A}ffect$ predicate ascertains whether its first parameter may execute code which affects the second one, while $\mathcal{M}ay\mathcal{A}ccess$ predicates ascertains whether its first parameter has *any* path to the second one.

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

$$\begin{array}{lcl} A & ::= & \textit{Expr} \mid \textit{R(Expr*)} \\ & \mid & \textit{Expr} \geq \textit{Expr} \mid A \wedge A \mid \ldots \\ & \mid & \exists x.A \mid \forall x.A \mid \ldots \\ & \mid & \textit{Expr}\textbf{:}\textit{ClassId} \\ & \mid & \mathcal{M}ay\mathcal{A}ffect\ (\ \textit{Expr,Expr}) \\ & \mid & \mathcal{M}ay\mathcal{A}ccess(\ \textit{Expr,Expr}) \\ & \mid & \textit{Expr}\ \textbf{obeys}\ \textit{SpcId} \\ \\ \textit{PredDescr} & ::= & \textbf{predicate}\ \textit{R}(\ \textit{ParId*}\ )\ \{\ \ A\ \} \end{array}$$

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

Definition 11 (Two-state Assertions)**.**

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

Given the syntax from above, we can express assertions like

$\forall$p.p $:_{pre}$ `Purse`.

`p.bank` $=_{pre}$ `RBS` $\rightarrow$ $\texttt{p.balance}_{pre} = \texttt{p.balance}_{post}$,

to require that the `balance` of any `Purse` belonging to `RBS` is immutable across the to states. Notice that for legibility, for infix predicates (such as = or :) we annotate the

$$
\begin{array}{lcl}
\mathcal{R}each(M,\sigma,\texttt{v:=}\mathbf{new}\ \texttt{c(}\ \texttt{a}_1,...\texttt{a}_n\texttt{)}\ ) & = & \{\ (\texttt{v:=}\mathbf{new}\ \texttt{c(}\ \texttt{a}_1,...\texttt{a}_n\texttt{)}\ ,\sigma),\ (\mathbf{skip},\sigma')\} \\
 & & \text{where } M,\sigma,\texttt{v:=}\mathbf{new}\ \texttt{c(}\ \texttt{a}_1,...\texttt{a}_n\texttt{)}\ \rightsquigarrow \sigma' \\
\mathcal{R}each(M,\sigma,\texttt{stmt;}\ \texttt{stmts}) & = & \mathcal{R}each(M,\sigma,\texttt{stmt}) \cup \mathcal{R}each(M,\sigma',\texttt{stmts}) \\
 & & \text{where } M,\sigma,\texttt{stmt} \rightsquigarrow \sigma' \\
\mathcal{R}each(M,\sigma,\texttt{v:=a}) & = & \{(\texttt{v:=a},\sigma),\ (\mathbf{skip},\sigma')\} \\
 & & \text{where } M,\sigma,\texttt{v:=a} \rightsquigarrow \sigma' \\
\mathcal{R}each(M,\sigma,\texttt{v:=a.m(}\ \texttt{a}_1,...\texttt{a}_n\texttt{)}\ ) & = & \{\ (\texttt{v:=a.m(}\ \texttt{a}_1,...\texttt{a}_n\texttt{)}\ ,\sigma),\ (\mathbf{skip},\sigma''')\ \} \cup \mathcal{R}each(M,\sigma',\texttt{stmts}) \\
 & & \text{where} \\
 & & \text{and } BLA = ...(\ \texttt{stmts;}\mathbf{return}\texttt{a}\ ) \text{ and} \\
 & & M,\sigma',\texttt{stmts} \rightsquigarrow \sigma'' \text{ and } \sigma''' = (\sigma\downarrow_1[\texttt{v} \mapsto \lfloor a \rfloor_{\sigma''}], \sigma''\downarrow_2) \\
\mathcal{R}each(M,\sigma,\mathbf{skip}) & = & \{\ (\mathbf{skip},\sigma)\ \} \\
\mathcal{R}each(M,\sigma,\mathbf{if}\ \texttt{a}\ \mathbf{then}\ \texttt{stmts}_1\ \mathbf{else}\ \texttt{stmts}_2\ ) & = & \{\ (\mathbf{if}\ \texttt{a}\ \mathbf{then}\ \texttt{stmts}_1\ \mathbf{else}\ \texttt{stmts}_2,\sigma),\ \} \cup \mathcal{R}each(M,\sigma,\texttt{stmts''}) \\
 & & \text{where } \texttt{stmts''} = \texttt{stmts}_1 \text{ if } \lfloor a \rfloor_\sigma = \mathbf{true},\ \text{otherwise } \texttt{stmts''} = \texttt{stmts}_2
\end{array}
$$

Figure 2. Reachable Configurations

predicate application rather than the assertion, *e.g.* we write $\texttt{p.bank=}_{pre}\texttt{RBS}$ to stand for $(\texttt{p.bank=RBS})_{pre}$.

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) $A\,\{\ \texttt{code}\ \}\ B$, 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) $A\,\{\texttt{any\_code}\}\ B$ 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 12 (Policies)**.**

$$
\begin{array}{lcl}
Policy & ::= & A \mid A\,\{\texttt{code}\}\ B \mid\ A\,\{\texttt{any\_code}\}\ B \\
PolSpec & ::= & spec\ SpcId\{\ Policy^*\ \}
\end{array}
$$

.

Validity of one-state, two-state assertions, and policies We first defined validity of one-state assertions:

Let $\sigma = (\phi,\chi)$ be a state. Then write $\sigma[v\mapsto\iota]$ as shorthand for $(\phi[v\mapsto\iota],\chi)$.

Definition 13 (Validity of one-state assertions – $\mathcal{M}ay\mathit{Affect}$ and $\mathcal{M}ay\mathit{Access}$)**.** *We define the validity an assertion A:*

$$\models \subseteq Module \times state \times Assertion$$

using the notation $M,\sigma \models A$:

- $M,\sigma \models e$ *iff* $\lfloor e \rfloor_{M,\sigma} = \mathbf{true}$.
- $M,\sigma \models P(e_1,...e_n)$ *iff* $M,\sigma \models A[e_1/p_1,...e_n/p_n]$ *where* $M(P) = \mathbf{predicate}\ P\ (\ p_1...p_n)\ \{\ A\ \}$, *undefined, otherwise.*
- $M,\sigma \models e_1 \geq e_2$ *iff* $\lfloor e_1 \rfloor_{M,\sigma} \geq \lfloor e_2 \rfloor_{M,\sigma}$.
- $M,\sigma \models A_1 \wedge A_2$ *iff* $M,\sigma \models A_1$ *and* $M,\sigma \models A_2$.
- $M,\sigma \models \exists x.A$ *iff for some address ι and some fresh variable $z \in VarId$, we have* $M,\sigma[z\mapsto\iota] \models A[z/x]$
- $M,\sigma \models \forall x.A$ *iff for all addresses $\iota \in dom(\sigma)$, and fresh variable z, we have* $M,\sigma[z\mapsto\iota] \models A[z/x]$.
- $M,\sigma \models e{:}C$ *iff* $\sigma(\lfloor e \rfloor_{M,\sigma})\downarrow_1 = C$.
- $M,\sigma \models \mathcal{M}ay\mathit{Affect}(e,e')$ *iff there exists method m, arguments $\bar{a}$, state σ', identifier z, such that* $M,\sigma[z\mapsto\lfloor e \rfloor_{M,\sigma}], z.m(\bar{a}) \rightsquigarrow \chi'$, *and* $\lfloor e' \rfloor_{M,\sigma} \neq \lfloor e' \rfloor_{M,\sigma\downarrow_1,\chi'}$.
- $M,\sigma \models \mathcal{M}ay\mathit{Access}(e,e')$ *iff there exist fields* f_1,... f_n, *such that* $\lfloor z.f_1...f_n \rfloor_{M,\sigma[z\mapsto\lfloor e \rfloor_{M,\sigma}]} = \lfloor e' \rfloor_{M,\sigma}$.
- $M,\sigma \models e\ \mathbf{obeys}\ S$ *iff* $\sigma(\lfloor e \rfloor_{M,\sigma})\downarrow_1 = C$, *and* $M(C) = \mathbf{class}\ C\ \mathbf{satisfies}\ ..,S,...\{\ ...\}$.

Note that the definition of $M,\sigma \models e\ \mathbf{obeys}\ S$ is syntactic; it depends on whether C, the *class* of the object denoted by e, claims that it satisfies specification S.

We now define validity of two state assertions, ...

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

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

using the notation $M,\sigma,\sigma' \models B$ as follows

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

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 $\texttt{M},\sigma_1,\sigma_2 \models \texttt{x.balance}_{post} = \texttt{x.balance}_{pre} + 10$.

Finally, we define what it means for a module M to satisfy a one-state assertion A, and require that the A holds

in all configuration arising from all possible extensions of M:

Definition 15. • $M \models A$ *iff*
$\forall M'.\forall(\texttt{stmts},\sigma) \in \mathcal{A}rising(M*M').\ M*M',\sigma \models A$

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 either the format

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

or the format

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

The former promises that execution of `stms` in any state which satisfies A will lead to a state which satisfies A'. The latter 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. Both the former and latter tuples also promise that the relation between the initial state, and any of the the intermediate states reached by execution of `stms` will be described by B.

The execution of `stmts` may call methods defined in M, and the predicates appearing in A, A', and B, may use predicates as defined in M. When the module M is implicit from the context we use the shorthand $\vdash A \,\{\, \texttt{stms} \,\}\, A' \bowtie B$.

*** I commented out the stuff about the logical variables; I hope we can do without ***

4.1 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.[1]

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

$$\begin{array}{l}\textbf{true}\\ \quad\{\texttt{this.f=this.f+3}\}\\ \texttt{this.f} = \texttt{this.f}_{pre} + 3\ .\\ \bowtie\\ \textbf{true}\end{array}$$

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

The rule (METH-CALL-1) describes method call. [2]

On the other hand, rule (METH-CALL-2) 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 assumptions 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'.[3]

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.

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 16. The requirement $A, B_1 \rightarrow_M \textbf{true}, A_2$ 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 16.3. The requirement $B_1, B_2 \rightarrow_M B$ 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 16.5. For example, with 16.3 we have $\texttt{x} = 5, \texttt{x}_{post} = \texttt{x} + 2 \rightarrow_M \textbf{true}, \texttt{x} = 7$, while with 16.5 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.

Definition 16 (Entailment)**.**

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

[1] 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.

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

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

(VARASG)

$$\frac{}{\vdash \textbf{true}\,\{\,\textbf{var}\,\mathtt{v}{:=}\mathtt{a}\,\}\,\mathtt{v} = \mathtt{a}_{pre} \bowtie \textbf{true}}$$
$$\vdash \textbf{true}\,\{\,\mathtt{v}{:=}\mathtt{a}\,\}\,\mathtt{v} = \mathtt{a}_{pre} \bowtie \textbf{true}$$

(FIELDASG)

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

(COND-1)

$$\frac{A \rightarrow_M \mathtt{cond} \quad \vdash A\,\{\,\mathtt{stmts}_1\,\}\,B \bowtie B'}{\vdash A\,\{\,\textbf{if}\,\mathtt{cond}\,\textbf{then}\,\mathtt{stmts}_1\,\textbf{else}\,\mathtt{stmts}_2\,\}\,B \bowtie B'}$$

(COND-2)

$$\frac{A \rightarrow_M \neg\mathtt{cond} \quad \vdash A\,\{\,\mathtt{stmts}_2\,\}\,B \bowtie B'}{\vdash A\,\{\,\textbf{if}\,\mathtt{cond}\,\textbf{then}\,\mathtt{stmts}_1\,\textbf{else}\,\mathtt{stmts}_2\,\}\,B \bowtie B'}$$

(SKIP)

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

(METH-CALL-1)

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

(METH-CALL-2)

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

(FRAME-METHCALL)

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

(SEQUENCE)

$$\frac{\vdash A\,\{\,\mathtt{stmts}_1\,\}\,B_1 \bowtie B' \qquad \vdash A_2\,\{\,\mathtt{stmts}_2\,\}\,B_2 \bowtie B' \qquad A, B_1 \rightarrow_M \textbf{true}, A_2 \qquad B_1, B_2 \rightarrow_M B}{\vdash A\,\{\,\mathtt{stmts}_1;\,\mathtt{stmts}_2\,\}\,B \bowtie B'}$$

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

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

Note that $A \rightarrow_M A'$ is equivalent with $M \models A \rightarrow A'$. TODO: what about the rest? Can they be expressed more succinctly?

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

Rule (FRAME-GENERAL) allows us to frame onto a tuple any assertion that has not been affected by the code. . For this, we need two notions of some code being disjoint from an assertion:

Definition 17 (Disjointness)**.**

- $M, \sigma \models \mathtt{stms} \,\#\#\, A$ *iff*
  $M, \sigma \models A \ \wedge\ \forall \sigma' \in \mathcal{R}each(M, \mathtt{stmts}, \sigma).\ M, \sigma' \models A.$
- $M, \sigma \models \mathtt{stms} \,\#\, A$ *iff*
  $M, \sigma \models A \ \wedge\ M, \sigma, \mathtt{stms} \rightsquigarrow \sigma' \ \rightarrow\ M, \sigma' \models A.$

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

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

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

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

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

$$\frac{\vdash A\,\{\,\texttt{stmts}\,\}\,B \bowtie B' \qquad A, B \to_M \mathbf{true}, A'}{\vdash A\,\{\,\texttt{stmts}\,\}\,A' \bowtie B'}\ (\textsc{Cons-3})$$

$$\frac{\vdash A\,\{\,\texttt{stmts}\,\}\,A' \bowtie B' \qquad A, A' \to_M B}{\vdash A\,\{\,\texttt{stmts}\,\}\,B \bowtie B'}\ (\textsc{Cons-4})$$

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

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

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

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 1. *For all modules M, and states σ,*

- *If* x *and* y *are textually different variables, then* $M, \sigma \models \texttt{x=a} \,\#\#\, \texttt{y} := \texttt{a'}$.
- *If* x *is not a prefix of the path* p*, then* $M, \sigma \models \texttt{p.f=a} \,\#\#\, \texttt{x} := \texttt{a'}$.
- *If* $M, \sigma \models \texttt{stms} \,\#\#\, A$ *then* $M, \sigma \models \texttt{stms} \,\#\, A$.

The rule (CONJ) allows us to combine different Hoare tuples for the same code, and follows standard Hoare logics.

Interestingly, our system has *four* rules of consequence. The fist rule, (CONS-1), is largely standard, as it allows us to strengthen the precondition A, and weaken the postcondition B, and 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.

For (CONS-1) we use the entailment $A \to_M A'$, which guarantees that any state which satisfied A also satisfies A', and that of the form $B \to_M B'$ which guarantees that any pair of states which together satisfy B also satisfy B'. This is described in Definition 16.

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 16. For example, we can use this rule to take

p1 **obeys** $Purse$
{ p2:=p1.sprout}
p2 **obeys** $Purse$
⋈
true

and deduce that

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

The next two rules, (CONS-3) and (CONS-4), allow us to swap between tuples where the postcondition is a one-state assertion, *i.e.*$\vdash A\,\{\,\texttt{stms}\,\}\,A' \bowtie B'$ and that where the postcondtion is a one state assertion, *i.e.*$\vdash A\,\{\,\texttt{stms}\,\}\,B \bowtie B'$.

The following lemma is an example entailment.

Lemma 2. *For all modules M:*
$\mathcal{MayAccess}(\texttt{x}, \texttt{y}) \wedge \mathcal{MayAccess}(\texttt{y}, \texttt{z}) \to_M \mathcal{MayAccess}(\texttt{x}, \texttt{z})$.

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.

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`.

4.2 Soundness

We first demonstrate that judgments made in the context of a module are preserved when we link a larger module. In lemma 3, we state that entailment is preserved by linking:

Lemma 3.

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

In lemma 1 we state that derivability and validity of Hoare tuples is preserved for larger modules

Theorem 1 (Linking preserves derivations and validity)**.** *For all modules* M, M'.

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

We now define what it means for a method body, and a class definition to adhere to its specification

We say that a method `m` defined a class `C` adheres to is specification,

$M \vdash \texttt{C}, \texttt{m}$

if we able to show that the body of `m` when executed in a state that satisfies A, the difference between the initial and final state is described by B, and will preserve B', where A and B' and B are the method's pre, postcondition, and invariant. Moreover, we say that a class adheres to its specification

$M \vdash \texttt{C}$

of all its methods adhere to their specification. Finally, a module adheres to its specification,

$M \vdash M$

if all the classes in Madhere to their specifications.

Definition 18 (Proving code's adherence to specification)**.**

- $M, C \vdash A \{ \texttt{this.m(par)} \} B$ *iff*
  we can prove that
  $M \vdash A \wedge \textbf{this} : C \{ \texttt{stmts} \} B[a/res] \bowtie \textbf{true}$
  where
  $\mathcal{M}(M, C, m) = \textbf{method}\ m(\texttt{par})\ \{\ \texttt{stmts}; \textbf{return}\ a \}$.
- $M, C \vdash A$ *iff*
  forall M'*, and for all* $(\sigma, \texttt{stmts}) \in \mathcal{A}rising(M' * M)$,[4]
  $M * M', \sigma \models \textbf{this} : C \rightarrow A$
- $M, C \vdash S$ *iff*
  for all policies $Pol \in M(S)$, [5] *we have* $M, C \vdash Pol$
- $M \vdash C$ *iff*
  for all S*, with* $M(C) = \textbf{class}\ C\ \textbf{satisfies} \ldots, S, \ldots \{ \ldots \}$ *,*
  we have that $M, C \vdash S$.
- $\vdash M$ *iff*
  $M \vdash C$ *for all classes* C *from* M

*** Very important: 2nd bullet point above. This one worries me. ****

Below we are defining and proving the soundness of our Hoare logic. Note that we do not require that $M \vdash M$, because we do not model object creation. If we had object creation in our system, we would have needed that requirement, and the proof of soundness would have required slightly more complex proof techniques such as a generation lemma, or double induction.

Lemma 4. *If* $M, \sigma, \texttt{stmts} \rightsquigarrow \sigma'$ *and if* $z \in dom(\sigma)$*, and* $M, \sigma' \models \mathcal{M}ayAccess(..)$ *then* $M, \sigma \models \mathcal{M}ayAccess(..)$ *or* ....

This lemma expresses the basic axiom of object-capability systems that "only connectivity begets connectivity" [1],

Proof. By structural induction over the derivation of $M, \sigma, \texttt{stmts} \rightsquigarrow \sigma'$. □

Theorem 2 (Soundness of the Hoare Logic)**.** *For any modules* M*and* M'*, code* `stms`*, assertions* A*,* A' *and* B *and* B'*. If*

1. $\vdash M$*, and*
2. $M \vdash A \{ \texttt{stms} \} B' \bowtie B$*, and*
3. $M, \sigma \models A$*, and*
4. $M * M', \sigma, \texttt{stms} \rightsquigarrow \sigma'$

then

1. $M, \sigma, \sigma' \models B'$*, and*
2. $\forall \sigma'' \in \mathcal{R}each(M, \sigma, \texttt{stmts}).\ M*M', \sigma, \sigma'' \models B$

Note that the first and second requirement above only talk of module M which has been verified, and which is used to prove the tuple $M \vdash A \{ \texttt{stms} \} B' \bowtie B$. However execution of the `stmts` is in the context of the linked program M', and validity of the assertion A is again wrt to both M and M' – ACTUALLY SD NOT SURE ABOUT THE LATTER

Proof. 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[6]

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

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

[5] Perhaps this needs to be expressed better?

[6] need to express better

If $M * M', \sigma, \texttt{stmts} \rightsquigarrow \sigma''$ in fewer steps than $M * M', \sigma', \texttt{stmts}' \rightsquigarrow \sigma'''$[7], 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 from Figure 4[8], 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. $A \equiv x$ **obeys** $\texttt{S} \wedge A'[\texttt{x/this}, \texttt{y/par}]$, where
7. $M(S) =$ **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. $\mathcal{M}(\texttt{M}, \texttt{C}, \texttt{m}) =$ ..**{** $\texttt{stmts}'$; **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. $M \vdash \texttt{this} : \texttt{C} \wedge A \{ \texttt{stms}' \} B''[\texttt{a/res}] \bowtie \textbf{true}$

From 10, and .. we obtain that
uu. $(\phi' \cdot \chi, \texttt{stms}'', x$ **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 lemma 4.

Case (FRAME-METHCALL) needs work

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 16) and the fact that $(\sigma, \texttt{stms}) \in \mathcal{R}each(M, \sigma, \texttt{stms})$.

Case (CONS-2) follows because $\sigma, \sigma' \models Q' \rightarrow Q$ if and only iff $\sigma, \sigma' \models Q$ assuming $\sigma, \sigma' \models Q'$.

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.

□

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

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

References

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