External Calls

1 Questions we discussed during in the Feb/March London meeting

1. How are Necessity Logic and Incorrectness Logic related?

   We discussed this, and have a crisp statement. Sophia has written it up in 2.

2. Can we find a different running example?

   ???

3. Do we need/want to change the specification language? In particular, use ideas from adversarial logic?

   At least we will change the notation, cf. sect 3. Also some new concepts possibly, eg $Path(...)$. **STOP PRESS:** : Some new ideas, with which we would need only binary operators for next-only-if and only-if (do not think only-through could be binary); cf. sect 3.

4. How can we reason about methods which allow external calls? How to extend our Logic, so that we can make the argument form sect 2.3.1, from the OOPSLA paper this document, and in particular

   - Can we link modules, and how much of spec can we inherit from small to large?
   - ???

5. Do we want "deep" or "shallow" meaning of "next observable" state in the semantics?

   We (provisionally) settled on "shallow". **STOP PRESS:** : We had thought that "shallow" cannot express the DAO vulnerability, but now Sophia think it does, c.f. Sect4.6.

6. Title for next paper: "Local reasoning about non-local properties" SD thinks it is a great title, but fears it is not true.

7. When we consider whether an assertion is satisfied in a state, do we a) **From_Current** only consider the objects which are transitively accessible from the currently top frame, or b) **From_All** all the objects which are transitively accessible from the all

the frames? Sophia prefers a), but Julian developed the `Blackadder` example, which, he convinced us all on Friday, would create problems for **From_Current** . **STOP PRESS:** : But now Sophia thinks it does not!.

8. Sophia's report about the meeting with Jules,

   (a) He asked about tool support for NL, and whether NL is modular, and how NL treats the universal quantifiers

   (b) He pointed us to Unrealizability Logic, from POPL 2023

   (c) Discussed relation with Incorrectness Logic

   (d) Asked about relation with temporal logic, and suggested (very humbly) that we use more succinct an "mathematical" notation

   (e) Sophia thinks that we can adopt "adversarial" notation and so facilitate the explanation of the "external steps—" semantics.

2 Comparison with Hoare Logic and Incorrectness Logic

Hoare/Incorrectness Logic are concerned with the study of the effect of some given code. They answer the question whether some specific code (`cmd`) might lead have a certain effect (going from A_1 to A_2). Thus they under/under approx the post condition (A_2) given a precondition (here A_1) and some code (here `cmd`).

In contrast, NL is concerned with security properties; in security, the code being executed might come from any, untrusted, third party, and therefore is unknown. Therefore, NL is not concerned with a *particular* code; instead, it is concerned with the conditions under which some effect (going from A_1 to A_2) , which take place. NL can be used to prove that certain effect will never take place.

Hoare Logic $\{\, A_1 \,\}\ \texttt{cmd}\ \{\, A_2 \,\}$	$\forall \sigma, \sigma'.[\ \ \sigma \models A_1\ \wedge\ \sigma, \texttt{cmd} \leadsto^* \sigma'\ \ \longrightarrow\ \ \sigma' \models A_2\ \ ]$ A_2 over–approximates the outcome of executing cmd in A_1
Incorrectness Logic $[\, A_1 \,]\ \texttt{cmd}\ [\, A_2 \,]$	$\forall \sigma'.[\ \ \sigma' \models A_2\ \ \longrightarrow\ \exists \sigma.(\ \sigma \models A_1\ \wedge\ \sigma, \texttt{cmd} \leadsto^* \sigma'\ )\ \ ]$ A_2 under-approximates the outcome of executing cmd in A_1
Necessity Logic $\{\, A_1 \diamondsuit A_2 \,\} \subseteq \{\, A_3 \,\}$	$\forall \sigma, \sigma', \texttt{cmd}.\ [\ \sigma \models A_1\ \wedge\ \sigma, \texttt{cmd} \leadsto^* \sigma'\ \wedge\ \sigma' \models A_2\ \ \longrightarrow\ \ \sigma \models A_3\ \ ]$ A_3 over–approximates preconditions that lead from A_1 to A_2

In Hoare Logic as well as in Incorrectness Logic, A_1 is a *sufficient* condition for the *particular* command cmd to reach A_2. But in Necessity Logic, A_3 is a *necessary* precondition for *any* command starting at A_1 to reach A_2.

In particular, if we have proven that a module satisfies $\{\, A_1 \diamondsuit A_2 \,\} \subseteq \{\, false \,\}$, then we know that a transition from A_1 to A_2 will never happen. Note that we could also have Incorrectness-Necessity Logic, where we underapproxiate the necessary condition, defined below. Now if we can prove that $\{\, A_1 \diamondsuit A_2 \,\} \supseteq \{\, A_3 \,\}$ and A_3 is not $false$, then we would have proven that a a transition from A_1 to A_2 can happen.[1]

I-Necessity Logic (future work ?) $\{\, A_1 \diamondsuit A_2 \,\} \supseteq \{\, A_3 \,\}$	$\forall \sigma, [\ \ \sigma \models A_3\ \wedge\ \sigma \models A_1\ \ \Longrightarrow \exists \texttt{cmd}, \sigma'(\ \sigma, \texttt{cmd} \leadsto^* \sigma'\ \wedge\ \sigma' \models A_2\ \ ]$ A_3 under–approximates preconditions that lead from A_1 to A_2

3 Notation

3.1 Specifications

Proposed new notation for specifications

[1]SOPHIA: not clear this is so; would we need the logic to be complete? TO-THINK.

For $from\ A1\ to\ A2\ onlyIf A3$, is: $\{\, A1 \diamondsuit A2 \,\} \subseteq \{\, A3 \,\}$.
For $from\ A1\ next\ A2\ onlyIf A3$, it is: $\{\, A1 \bigcirc A2 \,\} \subseteq \{\, A3 \,\}$.
For $from\ A1\ to\ A2\ onlyThrough A3$, is: $\{\, A1 \diamondsuit A2 \,\} \curvearrowright \{\, A3 \,\}$.

Proposal Should we perhaps only write necessity specs which will never happen, eg rather than $\{\, A_1 \diamondsuit A_2 \,\} \subseteq \{\, A_3 \,\}$ have something like $Never((A_1 \wedge \neg A_3) \diamond A_2)$. In terms of temporal logic, this would have the meaning: $\Box(\neg((A_1 \wedge \neg A_3) \wedge \diamond A_2))$. Or, we could go one step further, and specify in the terms of $\Box((A_1 \wedge \neg A_3) \diamond \neg A_2)$. This would turn our specs into binary operators.

We could have something like
(∗) $\{\, A \,\} \diamondsuit \{\, A' \,\}$
which would be like a shorthand for the temporal $\Box(A \to \diamond A')$, which means
(∗∗) $\forall \sigma, \sigma', \texttt{cmd}.\ [\ \sigma \models A\ \wedge\ \sigma, \texttt{cmd} \leadsto^* \sigma' \quad \longrightarrow \quad \sigma' \models A'\ ]$.

Then, our Bank spec could have the form
$\{\, \texttt{a:Account} \wedge Insd(\texttt{a.passwd}) \wedge \texttt{a.balance} = b \,\} \diamondsuit \{\, \texttt{a.balance} \geq b \,\}$

Discussion Having only binary operators would be good, but would open two problems a) how to describe "only-through", b) the "narrative" would be different. We would not have "necessary preconditions" any more. Our spec would be more like 2-state invariants.

Specification Implication

Definition 3.1. A specification $\texttt{S}'$ is *stronger* than a specification $\texttt{S}$:
$$\texttt{S}' \prec \texttt{S} \quad \triangleq \quad \forall \texttt{M}.[\ \ \texttt{M} \models \texttt{S}' \quad \longrightarrow \quad \texttt{M} \models \texttt{S}\ \ ]$$

[2]

3.2 Assertions

Proposed new notation for access: $Acc(y, x)$, and for external: $Extrn(p)$. Therefore, we will have nothing like "words" or "keywords" in the assertions, or the specifications/

Based on the above, we define "outside", as something that is accessible from something external:

Definition 3.2. We define the predicates $Outsd(\_)$, and $Insd(\_)$ as follows:
$$\texttt{M}, \sigma \models Outsd(x) \quad \triangleq \quad \texttt{M}, \sigma \models \exists y.[\ Acc(y, x) \wedge Extrn(y)\ ]$$

$$\texttt{M}, \sigma \models Insd(x) \quad \triangleq \quad \neg\ (\ \texttt{M}, \sigma \models Outsd(x)\ )$$

[2] We should explore some $\prec$ relations, in the future. And relate with similar in temporal logic. But perhaps all the $\prec$ are already covered in the inference rules that Julian had created? Check whether Julian's rules are complete?

Implication Implication in assertions must be $\longrightarrow$. But what would it be in the meta-language, eg in Def. 3.1.

4 Code Examples

Here we will write examples of the codes. We denote though `untrust.unkn(...)` the call to an untrusted (external) object.

4.1 Something

These method s have no ”story”; they serve as warnings about things we should not take for grated.

```
class Account

    void warning( ) {
        p := new Password
        p' := new Password
        a := new Accoount.
        a.setPassword(null,p);
        untrust.unkn(a)
        this.password := p  /*  and in variant\_1 */     this.password := p'
        return p
    }

    void dare_you(){
        this.status := frozen    //  to transfer we need the account to be unfrozen
        untrust.unkn(this.password)
        this.password:=new Password
        this.frozen := false
    }   // SD: hmhhh,  essentially we de-activated the account! We need 2.3.1!!!

...
```

Deactivate and leak old

This method writes a new `Passowrd` into the account, and thus essentially disables it. And it returns the old password.

```
class Account

    Password deacrtivate_and_leak_old( ) {
        p := this.password
        this.password := new Password
        return p
    }

...
```

Julian will write how this function can be used in the client to xxxx.

4.2 Transfer with external

This method does half the job of `transfer`, then calls ”outside”, and then does the rest. Was meant to demo proof of preservation of currency. Proof ot fre- and post in the presence of unknown.

```
class Account

    void transfer_2 (Password p, Account to, int amt) {
        if (p==this.password){
                this.balance -= 2
                to.balance += 2
                untrust.unkn();
                this.balance -= amt -2
                to.balance += amt-2

    }

...
```

4.3 Bank may leak unused Passwords

Bank keeps all unused passwords in a list. Sophia will expand it once ”deactive and leak old” is done.

```

class Bank

    Password makePassword( ) {
        p := new Password
        this.listUnusedPwds.enter(p)
        return p;
    }

    Password getMeAPassword(){
        p := this.listUnusedPwds.popAndTop();
        return p;
    }

...
```

4.4 Safe External Call

Here we overwrite the password before making the external calls

```
class MrBean
  method transfer(from, to, pwd)
    from.transfer(to, pwd)
    p = new Password()
    from.setP(pwd, p)
    log.logger(from)
    from.setP(p, pwd)
```

4.5 BlackAdder's Box

Below is a variant of the example of 2.3.1 where the Bank Account has a box field that may contain the password. If it contains the password, then it will allow anyone to change the balance.

This example has "ambient authority", but the authority is not explicit. INTERESTING!

```
1  module BankAccount
2    class BankAccount
3      field password
4      field balance
5      field box
6      method transfer(pwd, to)
7        if pwd == password || box.open() == password
8          this.balance -= 10
9          to.balance += 10
10     method setBox(b)
11       this.box = b
12     method setPassword(pwd, newP)
13       if this.password == pwd
14         this.password = newP
15
16 module Ext
17   class Box
18     field thing
19     method open()
20       return thing
21     method setThing(t)
22       this.thing = t
23
24 module Client
25   class BlackAdder
26     method doSomething()
27       a = new Account()
28       p = new Password()
29       b = new Box()
30       b.setThing(p)
31       a.setPassword(null, p)
32       a.setBox(b)
33       u = new Unknown()
34       u.stuff(a)
```

On Friday, we had thought that RULE-2 from Sect. 7.2 would be unsound if applied on line 34 (the call `u.stuff(a)`). Namely, we can see that $\texttt{BankAccount} \models \texttt{S}_{\texttt{pwd\_prot\_bal}}$. Also, in line 34, the account is no longer protected, as its box contains the password. On Friday we thought that RULE-2 would erroneously guarantee that the balance would not decrease.

Here we overwrite the password before making the external calls. We can prove it with this week's work. **STOP PRESS:** But SD now argues that Rule-2 would make no such guarantee, Namely, in line 34, we are passing `a` as an argument, and `a` has access to its box (which is external), and which has access to the password. Therefore, in we consider only the objects that are (transitively) accessible from the frame that gets created in line 34, then $Insd(a.pwd)$ would not hold. And therefore $\texttt{S}_{\texttt{pwd\_prot\_bal}}$ would not be applicable.

4.6 DAO

STOP PRESS: Sophia maintains that the DAO breaks the invariant even if we consider "shallow semantics" Here the DAO problem paraphrased:

```
1  private class Bank
2      field money: int
3
4      method pay(u: Unknown, amt: int)
5      {
6           money:- ant
7           ...
8       }
9
10 class DAOAccount
11     field balance: int
12     field myBank: Bank
13
14     method payOut(u: Unknown){
15         if balance>0 then
16             myBank.pay(u,this.balance);
17             u.notify("made the payment")
18     }
19 }
```

On line 17 we are making an external call. At this point, $\texttt{S}_{\texttt{curr\_consistnt}}$ from Sect. 5.5 mandates that we have to establish that $\texttt{myBank.currcy} = \texttt{myBank.money}$. However, $\texttt{myBank.currcy} = \texttt{myBank.money}$ does not hold. Note that for this argument we make no special use of NL, and it makes no difference wether we have shallow or deep semantics.

5 Specification Examples

Here we will write examples of the specs

5.1 Flavours

There is an open question as to whether the semantics of from-to-... is "deep" or "shallow". By "deep" we mean the semantics where to "to"-part can see inside the future call stack. For example, if external method "m1" called internal method "m2", and if "m2" called external method " m3", then, fo the semantic oif from-to, do we consider pairs of states where σ is in "m1" and σ' is in "m3"? The "deep" defintion would say "yes", and the shallow definition would say "no". The choice does make a difference.

5.2 Passwords protect the balance

$$\mathtt{S}_{\mathtt{pwd\_prot\_bal}} \triangleq \forall a.\forall b.\ \{\, a : \mathtt{Account} \wedge a.\mathtt{balance} = b \diamondsuit a.\mathtt{balance} < b \,\} \subseteq \{\, \mathit{Outsd}(a.\mathtt{pwd}) \,\}$$

5.3 No Password get leaked

$$\mathtt{S}_{\mathtt{no\_leak\_val}} \triangleq \forall a.\forall p.\ \{\, a : \mathtt{Account} \wedge a.\mathtt{pwd} = p \diamondsuit \mathit{Outsd}(p) \,\} \subseteq \{\, \mathit{Outsd}(p) \,\}$$

$$\mathtt{S}_{\mathtt{no\_leak\_fld}} \triangleq \forall a.\forall p.\ \{\, a : \mathtt{Account} \diamondsuit \mathit{Outsd}(a.\mathtt{pwd}) \,\} \subseteq \{\, \mathit{Outsd}(a.\mathtt{pwd}) \,\}$$

$$\mathtt{S}_{\mathtt{no\_leak\_strong}} \triangleq \forall p.\ \{\, p : \mathtt{Password} \diamondsuit \mathit{Outsd}(p) \,\} \subseteq \{\, \mathit{Outsd}(p) \,\}$$

$$\mathtt{S}_{\mathtt{no\_leak\_wrong}} \triangleq \forall p.\ \{\, \mathit{true} \diamondsuit p : \mathtt{Password} \wedge \mathit{Outsd}(p) \,\} \subseteq \{\, \mathit{Outsd}(p) \,\}$$

Here the following holds:

$$\mathtt{S}_{\mathtt{no\_leak\_wrong}} \prec \mathtt{S}_{\mathtt{no\_leak\_strong}} \prec \mathtt{S}_{\mathtt{no\_leak\_val}}$$
$$\mathtt{S}_{\mathtt{no\_leak\_strong}} \prec \mathtt{S}_{\mathtt{no\_leak\_fld}}$$

Note also that $\mathtt{S}_{\mathtt{no\_leak\_wrong}}$ is too strong, because it precludes the creation of any external `Password`. Notice the difference between $\mathtt{S}_{\mathtt{no\_leak\_fld}}$ and $\mathtt{S}_{\mathtt{no\_leak\_val}}$:

- $\mathtt{S}_{\mathtt{no\_leak\_fld}}$ says that if a password to an account is not externally known, then the field will never be externally known. This does not preclude the example in §4.3, as it only says that at any moment, the present value will not be externally known. This is useful in the case of protecting the account balance, as it is necessary to know the current password to withdraw money from the account, not previous passwords.

- $\mathtt{S}_{\mathtt{no\_leak\_val}}$ says that if the value pointed at by the field is not externally known, then it will never be externally known. This is useful to say that certain fields are safe, and do not leak the values stored there. (julian: I'm not entirely sure that this spec is good enough though)

5.4 Bank currency constant

We define as `currcy` the currency of a bank, ie the sum of the balances of all accounts held by that bank.

$$\mathtt{S}_{\mathtt{curr\_const}} \triangleq \forall b, n.\ \{\, b : \mathtt{Bank} \wedge b.\mathtt{currcy} = n \diamondsuit b.\mathtt{currcy} \neq n \,\} \subseteq \{\, \mathit{false} \,\}$$

$$\mathtt{S}_{\mathtt{curr\_infl}} \triangleq \forall b.\ \{\, b : \mathtt{Bank} \wedge b.\mathtt{currcy} = n \diamondsuit b.\mathtt{currcy} > n \,\} \curvearrowright \{\, \mathit{Calls}(\_, b.\mathtt{print}(\_)) \,\}$$

[3]

5.5 DAO Bank currency consistent

As in Sect 5.4, we define as `currcy` the currency of a bank, ie the sum of the balances of all accounts held by that bank. And we require that the currency is the same as the `money`:

$$\mathtt{S}_{\mathtt{curr\_consistnt}} \triangleq \forall b, n. \quad \{\, b : \mathtt{Bank} \Diamond b.\mathtt{currcy} \neq b.\mathtt{money} \,\} \subseteq \{\, false \,\}$$

The above is an invariant, ie it requires that $b.\mathtt{currcy} = b.\mathtt{money}$ always holds (since $\diamond$ means 0 or more execution steps.

6 Reflections

Here we will say which function satisfy which specs, under which other specs.

7 Extensions to the Inference System of Necessity Logic

We discussed a lot of variations of inference rules. But in the end, Sophia thinks that we need no more than a way to argue that a sequence of "known" code and calls to unknown code satisfies a Hoare triple. And I now think we do not need to "go beyond" Hoare triple notation. :-)

Note that here, the code about which we are reasoning (ie creating Hoare triples) does not need to be coming from the "safe" module. Instead, we may be reasoning about code which is a client of the "safe" module. That is, we have three "views": the code being checked, i.e., $stmts_1;\ stmts_2$ below, the "unknown" code (here the calls in $stmts_2$), and the "safe" module, M, which satisfies the specification S. it is possible that in $stmts_1$ we have calls to function from M, and we could reason about them using their pre- and post-conditions. [4]

7.1 Rule-1

We believe that the current rule would be sound:

$$\frac{\begin{array}{ll} \mathtt{M},\mathtt{S} \vdash \{A_1\}\ stms_1\ \{A_2\} & \overline{x} \text{ is external} \\ \mathscr{R}ev(A_2,\, x, \overline{z}) = A_3 & \mathscr{R}et(A_3,\, \mathtt{S})_{\mathtt{M}} = A_4 \end{array}}{\mathtt{M},\mathtt{S} \vdash \{A_1\}\ stmts_1;\ x.m(\overline{z})\ \{A_4\}}$$

[3] The following was written here, but Sophia it is not true: Note that with the deep semantics, the method `transfer_2` does not satisfy the spec $\mathtt{S}_{\mathtt{curr\_const}}$, but with the shallow semantics, it does satisfy the spec $\mathtt{S}_{\mathtt{curr\_const}}$. The proof will require the use of the spec itself (as in modular verification). And similar arguments about satisfaction of, and proof of adherence to, $\mathtt{S}_{\mathtt{curr\_infl}}$.

[4] The above should address the question that was raised by Peter Mueller when he read section 2.3.1. I fear it is not that crisp... HELP.

In the above, the term $\mathscr{R}ev(A,\ \overline{z})$ returns an assertion A' which is essentially A, with the difference that all elements from $\overline{z}$ are considered as $Outsd(\_)$. Moreover, $\mathscr{R}et(A,\ \mathtt{S})_{\mathtt{M}}$, contains the parts of A that are preserved through $\mathtt{S}$.[5]

Definition 7.1. We define the function $\mathscr{R}ev(\_,\ \_)$ below:

$$
\begin{array}{lcll}
\mathscr{R}ev(A,\ \epsilon) & \triangleq & A & \\
\mathscr{R}ev(A,\ (z, \overline{z})) & \triangleq & \mathscr{R}ev(\mathscr{R}ev(A,\ z),\ \overline{z}) & \\
\mathscr{R}ev(true,\ z) & \triangleq & true \wedge Outsd(z) & \\
\mathscr{R}ev(false,\ z) & \triangleq & false \wedge Outsd(z) & \\
\mathscr{R}ev(Insd(u),\ z) & \triangleq & (u \neq z \wedge Insd(u) \vee u = z) \wedge Outsd(z) & \\
\mathscr{R}ev(Outsd(u),\ z) & \triangleq & Outsd(u) \wedge Outsd(z) & \\
\mathscr{R}ev(p = p',\ z) & \triangleq & p = p' \wedge Outsd(z) & \text{where } p, p' \text{ are paths} \\
\mathscr{R}ev(A \wedge A',\ z) & \triangleq & \mathscr{R}ev(A,\ z) \wedge \mathscr{R}ev(A',\ z) & \\
\mathscr{R}ev(A \vee A',\ z) & \triangleq & \mathscr{R}ev(A,\ z) \vee \mathscr{R}ev(A',\ z) & \\
\mathscr{R}ev(\neg A,\ z) & \triangleq & \mathscr{R}ev(A',\ z) & \text{where } A' \equiv \neg A \\
\mathscr{R}ev(\forall u.A,\ z) & \triangleq & \forall u.\mathscr{R}ev(A,\ z) & \\
\mathscr{R}ev(\exists u.A,\ z) & \triangleq & \exists u.\mathscr{R}ev(A,\ z)^{6} & \\
\mathscr{R}ev(Q(u),\ z) & \triangleq & \mathscr{R}ev(Qbody[x/u],\ z) & \text{where } Q \text{ is inductively defined}
\end{array}
$$

Lemma 1. *If $\mathscr{R}ev(A,\ z){=}A'$, then $A \rightarrow A'$, and $A' \rightarrow Outsd(z)$.* [7]

We will now define the preservation function:

Definition 7.2. We define $\mathscr{R}et(A,\ \mathtt{S})_{\mathtt{M}}$ by cases over $\mathtt{S}$ and its relation to A below:

$$
\mathscr{R}et(A,\ \{\, A_1 \diamondsuit A_2 \,\} {\subseteq} \{\, A_3 \,\})_{\mathtt{M}} \ \triangleq \ \begin{cases} \neg A_3 & \text{if } \mathtt{M} \vdash A \ \rightarrow \ A_1 \wedge \neg A_3\,, \\ true & \text{otherwise}\,. \end{cases}
$$

$$
\begin{array}{lcl}
\mathscr{R}et(A,\ \{\, A_1 \diamondsuit A_2 \,\} {\subseteq} \{\, A_3 \,\};\, \mathtt{S})_{\mathtt{M}} & \triangleq & \mathscr{R}et(A,\ \{\, A_1 \diamondsuit A_2 \,\} {\subseteq} \{\, A_3 \,\})_{\mathtt{M}} \wedge \\
& & \mathscr{R}et(A,\ \mathtt{S})_{\mathtt{M}} \\
\mathscr{R}et(A,\ \{\, A_1 \diamondsuit A_2 \,\} {\curvearrowright} \{\, A_3 \,\};\, \mathtt{S})_{\mathtt{M}} & \triangleq & \mathscr{R}et(A,\ \mathtt{S})_{\mathtt{M}} \\
\mathscr{R}et(A,\ \{\, A_1 \bigcirc A_2 \,\} {\subseteq} \{\, A_3 \,\};\, \mathtt{S})_{\mathtt{M}} & \triangleq & \mathscr{R}et(A,\ \mathtt{S})_{\mathtt{M}}
\end{array}
$$

7.2 Rule-2

On Friday we thought that the rule below was not sound (because of `Blackadder`); **STOP PRESS:** but now SD thinks it is sound, cf discussion under Secr 4.5.

$$
\frac{\begin{array}{cc} \mathtt{M},\ \mathtt{S}\ \vdash\ \{A_1\}\ stms_1\ \{A_2\} & \overline{x} \text{ is external} \\ \mathscr{R}estrTo(A_2, x, \overline{z}) = A_3 & \mathscr{R}et(A_3,\ \mathtt{S})_{\mathtt{M}} = A_4 \end{array}}{\mathtt{M},\ \mathtt{S}\ \vdash\ \{A_1\}\ stmts_1;\ x.m(\overline{z})\ \{A_4\}}
$$

[5]needs better explanation.

[7]here we also need a M

This rule is very similar to that from 7.1, with the only difference that we use $\mathscr{R}estrTo(A_2,\, x, \overline{z})$, rather than $\mathscr{R}ev(A_2,\, x, \overline{z})$. $\mathscr{R}estrTo(A_2,\, \overline{x, z})$ is meant to restrict visibility to only the objects that are transitively accessible from $\overline{x, z}$, and then also turns the external $\overline{z}$s into ousiders. This assumes that we went with **From_Current**. **STOP PRESS:** I am not that sure how to define $\mathscr{R}estrTo(,\,)$.

7.3 Can we combine the holistic specs of two modules?

$\texttt{M1} \models \texttt{S}_\texttt{a}$ does *not* imply that $\texttt{M1} \circ \texttt{M2} \models \texttt{S}_\texttt{a}$. Then, Julian tried something like $\texttt{M1} \models \texttt{S}_\texttt{a}$ and $\texttt{M2} \models \texttt{S}_\texttt{a}$, implies that $\texttt{M1} \circ \texttt{M2} \models \texttt{S}_\texttt{a}$. This approach is promising, but we still have some problems because $Outsd(_)$'s meaning depends on the module, and the fact that $Outsd(_)$ can appear in both positive and negative positions.

During our meeting, we thought that such an inference would not ever be sound, because of a counterexampe proposed by SD. But **STOP PRESS:** , now SD no longer things that this counterexample would be valid.

The counterexample had the structure that for some modules and assertions, we would have

$\texttt{M}x \models \{\, A_1 \diamondsuit A_2 \,\} \subseteq \{\, false \,\}$

$\texttt{M}y \models \{\, A_1 \diamondsuit A_2 \,\} \subseteq \{\, false \,\}$

$\texttt{M}x \circ \texttt{M}y \models \{\, A_1 \diamondsuit A_2 \,\} \subseteq \{\, A_3 \,\}, \quad$ and $A_3 \neq false$.

And the example I had given was something like "an object of class `Z` will perform a certain action (here $A_1 \diamond A_2$) only if it receives a special message from an object from module `Mx`, as well as a special message from module `My`". I thought that the specifications above were correctly representing that state of affairs. But I was wrong! It should be something like

$\texttt{M}x \models onlyIf A_1 A_2$Z has received both the X and Y messages

8 What we can / cannot prove

Here we discuss which examples we can and which we cannot prove now

8.1 Adding external calls to the "safe module's" code

The method from below can be checked with $Rule_2$. **STOP PRESS:** But notice that the spec is too strong! as it assumes that no external entity has access to the password. In that case, nobody can call the method `transfer_3`

```
class Account

    fld logger: Logger // external class, untrusted

     void transfer_3a (Password p, Account toAcc, int amt)
     POST:  Insd(this.passwd) ⟶ [ p==this.password) → ...
                                        ∧
                                        ∀ a:Account.( a≠fromAcc,toAcc → ...     )  ]
     {
```

```
            if (p==this.password)
                    this.balance -= amt
                    toAcc.balance += amt
                    logger.log(this)
     }
 ...
```

STOP PRESS: Should we consider "relative" inside/outside predicates, e.g. $Outsd(x, y)$ holds if we only consider objects transitively accessible from `y`; that is $Outsd(x, y)$ holds if none of the external objects transitively accessible from `y` had access to `x`?

Here with the new spec

```
class Account

     field passwd : Password
     field logger: Logger // external class, untrusted

     void transfer_3b (Password p, Account toAcc, int amt)
     POST: [ ∀ a:Account. Outsd(a.passwd, logger) ] ⟶
                                 [ p == this.password) → ...
                                   ∧
                                   ∀ a:Account.( a≠fromAcc,toAcc → ...    )  ]
     {  ... as in  transfer_3a  ...  }
 ...
```

8.2 Adding external calls in "our" code – Me Bean

We now consider functions which belong to a class that is not part of $\mathtt{M}_{BA}$, which use $\mathtt{M}_{BA}$ as well as some untrusted third party code. Here we have the module `MrBean` (ours), which relies on $\mathtt{M}_{BA}$ in order to pass its account to a third party while preserving some guarantees.

Consider method `transfer_4`, which is similar to `transfer_3`. **STOP PRESS:** I changed the spec. [8]

```
module MrBean

   class MrAtkinson

      field logger: Logger // external class, untrusted

     void transfer_4 (Account fromAcc, Password p, Account toAcc, int amt)
     POST:  [  ∀ a:Account. Outsd(a.passwd, logger) ] ⟶
                                 [ p == fromAcc.password) → ...
                                   ∧
                                   ∀ a:Account.( a≠tfromAcc,toAcc → ...    )  ]

```

[8] Here some earlier thoughts on the matter in the new spec: I suppose that we were assuming that we have something like $\neg Outsd$(`a.passwd`)? Or that `a.passwd` is not outside the module `MrBean` $\circ$ `BankAccount`? Perhaps $Outsd(\_)$ should mention what it is outside of? Perhaps the module `MrBean` would have as invariant that all "its" Accounts keeps their Passwords "inside"?

```
    {
         fromAcc.transfer(p,toAcc,amt);
         logger.log(fromAcc)
     }
...
```

In transfer_4 from above, we are passing an object from the "safe" module (here fromAcc) to the untrusted object. What if we were to pass not only a "safe" object, but also an object from "our" module? Consider the function transfer_5 from below.

```
module MrBean

   class MrAtkinson

    field logger: Logger // external class, untrusted

     // CHALLENGE
     void transfer_5 (Account fromAcc, Password p, Account toAcc, int amt)
     POST:  [  ∀ a:Account. Outsd(a.passwd,logger) ] ⟶
                                  [ p == fromAcc.password) → ...
                                    ∧
                                    ∀ a:Account.  a≠this,toAcc → ...     ]

     {
         fromAcc.transfer(p,toAcc,amt);
         logger.log(fromAcc,this)
     }
...
```

In transfer_5 above, we are passing this to the logger. The function transfer_5 is only satisfied if MrAtkinson does not leak the password, and does nor make unauthorized payments. So, it seems as if proving transfer_5 requires a holistic spec for MrArkinson too.

8.3 What can be proven and how

We can prove transfer_3a using $Rule_1$, but as noted above, its spec is far too strong.

How could we prove the spec of transfer_3a, and transfer_4a and transfer_4b.There are the following solutions

STOP PRESS: The below needs revisiting:

1. Include the Logger's code (or its spec) into the proof basis of Bank, and make a holistic spec for both of them together. This is a non-solution, because then the logger would not be "external", and "untrusted".
2. Include the Logger's code (or its spec) into the proof basis of MrBean. This is also a non-solution, because then the logger would not be "external", and "untrusted".
3. Include MrBean's code into the proof basis of Bank, and make a holistic spec for both of them together.

(a) We just prove holistically the "whole" new module. We then only need

(b) We were hoping that we would be able to "inherit" some of the holistic spec of the `Bank`. As we know, she meaning of $Outsd(\_)$ is delicate as it depends on modules, and appears in both positive and negative positions. But it might be possible.

4. Introduce the concept of path-accessibility, which might be $Path(o, o', A(o''))$, and would mean that any path from o to o' must go through an object o'' which satisfies property $A(o'')$

5. Adopt $Rule_2$

, **Wrong Conjecture** Sophia thought that if all the calls in the body of a function in a client module are either calls to untrusted, or to the safe module (ie no calls to the module at hand), then we should be able to verify it, without holistic spec of the module itself. But `transfer_5` proves her wrong.

Still Open How to prove `transfer_5` in a modular way?