

Reasoning about External Calls

ANONYMOUS AUTHOR(S)

In today's complex software, internal trusted code is tightly intertwined with external untrusted code. To reason about internal code, programmers must reason about the potential effects of calls to external code, even though that code is not trusted and may not even be available.

The effects of external calls can be *limited* if internal code is programmed defensively, limiting potential effects by limiting access to the capabilities necessary to cause those effects.

This paper addresses the specification and verification of internal code that relies on encapsulation and object capabilities to limit the effects of external calls. We propose new assertions for access to capabilities, new specifications for limiting effects, and a Hoare logic to verify that a module satisfies its specification, even while making external calls. We illustrate the approach through a running example with mechanised proofs, and prove soundness of the Hoare logic.

CCS Concepts: • **Software and its engineering** → *Access protection*; **Formal software verification**; • **Theory of computation** → *Hoare logic*; • **Object oriented programming** → *Object capabilities*.

1 INTRODUCTION

External calls are pervasive in today's open world software. External, untrusted, or unknown code calls our trusted internal code, that internal code calls out to other external code and external code can even call back into internal code — all within the same call chain. This paper addresses reasoning about *external calls* — when trusted internal code calls out to untrusted, unknown external code. This reasoning is hard because by “external code” we mean untrusted code where we don't have a specification, where we may not be able to get source code, or which may even have been written to attack and subvert the system.

In the code sketch to the right, an internal module, M_{intl} , has two methods. Method `m2` takes an untrusted parameter `untrst`, at line 6 it calls an unknown external method `unkn` passing itself as an argument. The challenge is: What effects will that method call have? What if `untrst` calls back into M_{intl} ?

```
1 module  $M_{intl}$ 
2   method m1 ..
3     ... trusted code ...
4   method m2(untrst:external)
5     ... trusted code ...
6     untrst.unkn(this)
7     ... trusted code ...
```

External calls need not have arbitrary effects. If the programming language supports encapsulation (e.g. no address forging, private fields, etc.) then internal modules can be written *defensively* so that effects are either

Precluded, i.e. guaranteed to *never happen*. E.g., a correct implementation of the DAO [24] can ensure that the DAO's balance never falls below the sum of the balances of its subsidiary accounts, or

Limited, i.e. they *may happen*, but only in well-defined circumstances. E.g., while the DAO does not preclude that a signatory's balance will decrease, it does ensure that the balance decreases only as a direct consequence of calls from the signatory.

Precluded effects are special case of limited effects, and have been studied extensively in the context of object invariants [8, 41, 66, 91, 110]. In this paper, we tackle the more general, and more subtle case of reasoning about limited effects for external calls.

The Object Capability Model. The object-capability model combines the capability model of operating system security [68, 116] with pure object-oriented programming [1, 108, 113]. Capability-based operating systems reify resources as *capabilities* — unforgeable, distinct, duplicable, attenuable, communicable bitstrings which both denote a resource and grant rights over that resource. Effects can only be caused by invoking capabilities: controlling effects reduces to controlling capabilities.

Mark Miller’s [83] *object-capability* model treats object references as capabilities. Building on early object-capability languages such as E [83, 86] and Joe-E [80], a range of recent programming languages and web systems [19, 53, 103] including Newspeak [16], AmbientTalk [32] Dart [15], Grace [11, 59], JavaScript (aka Secure EcmaScript [85]), and Wyvern [79] have adopted the object capability model. Security and encapsulation is encoded in the relationships between the objects, and the interactions between them. As argued in [42], object capabilities make it possible to write secure programs, but cannot by themselves guarantee that any particular program will be secure.

Reasoning with Capabilities. Recent work has developed logics to prove properties of programs employing object capabilities. Swasey et al. [111] develop a logic to prove that code employing object capabilities for encapsulation preserves invariants for intertwined code, but without external calls. Devriese et al. [34] describe and verify invariants about multi-object structures and the availability and exercise of object capabilities. Similarly, Liu et al. [70] propose a separation logic to support formal modular reasoning about communicating VMs, including in the presence of unknown VMs. Rao et al. [100] specify WASM modules, and prove that adversarial code can affect other modules only through functions they explicitly export. Cassez et al. [21] model external calls as an unbounded number of invocations to a module’s public interface.

The approaches above do not aim to support general reasoning about external effects limited through capabilities. Drossopoulou et al. [43] and Mackay et al. [74] begin to tackle external effects; the former proposes “holistic specifications” to describe a module’s emergent behaviour, and the latter develops a tailor-made logic to prove that modules which do not contain external calls adhere to holistic specifications. Rather than relying on problem-specific, custom-made proofs, we propose a Hoare logic that addresses access to capabilities, limited effects, and external calls.

This paper contributes. (1) *protection assertions* to limit access to object-capabilities, (2) a specification language to define how limited access to capabilities should limit effects, (3) a Hoare logic to reason about external calls and to prove that modules satisfy their specifications, (4) proof of soundness, (5) a worked illustrative example with a mechanised proof in Coq.

Structure of this paper. Sect. 2 outlines the main ingredients of our approach in terms of an example. Sect. 3 outlines a simple object-oriented language used for our work. Sect. 4 and Sect 5 give syntax and semantics of assertions and specifications. Sect. 6 develops a Hoare logic to prove external calls, that a module adheres to its specification, and summarises the Coq proof of our running example (the source code will be submitted as an artefact). Sect. 7 outlines our proof of soundness of the Hoare logic. Sect. 8 concludes with related work. Fuller technical details can be found in the appendices in the accompanying materials.

2 THE PROBLEM AND OUR APPROACH

We introduce the problem through an example, and outline our approach. We work with a small, class-based object-oriented, sequential language similar to Joe-E [80] with modules, module-private fields (accessible only from methods from the same module), and unforgeable, un-enumerable addresses. We distinguish between *internal objects* — instances of our internal module M ’s classes — and *external objects* defined in *any* number of external modules \bar{M} .¹ Private methods may only

¹We use the notation \bar{z} for a sequence of z , i.e. for z_1, z_2, \dots, z_n

be called by objects of the same module, while `public` methods may be called by *any* object with a reference to the method receiver, and with actual arguments of dynamic types that match the declared formal parameter types²

We are concerned with guarantees made in an *open* setting; Our internal module M must be programmed so that execution of M together with *any* unknown, arbitrary, external modules \bar{M} will satisfy these guarantees – without relying on any assumptions about \bar{M} 's code.³ All we can rely on, is the guarantee that external code interacts with the internal code only through public methods; such a guarantee may be given by the programming language or by the underlying platform.⁴

Shop – illustrating limited effects

Consider the following internal module M_{shop} , containing classes `Item`, `Shop`, `Account`, and `Inventory`. Classes `Inventory` and `Item` are straightforward: we elide their details. `Accounts` hold a balance and have a key. Access to an `Account`, allows one to pay money into it, and access to an `Account` and its `Key`, allows one to withdraw money from it. A `Shop` has an `Account`, and a public method `buy` to allow a buyer – an external object – to buy and pay for an `Item`:

```

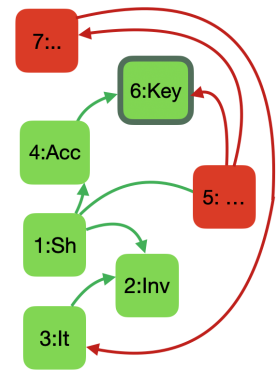
1  module Mshop
2    ...
3    class Shop
4      field acct:Account, invntry:Inventory, clients:external
5      public method buy(buyer:external, anItem:Item)
6        int price = anItem.price
7        int oldBlnc = this.acct.blnc
8        buyer.pay(this.acct, price)
9        if (this.acct.blnc == oldBlnc+price)
10         this.send(buyer, anItem)
11       else
12         buyer.tell("you have not paid me")
13       private method send(buyer:external, anItem:Item)
14       ...

```

The sketch to the right shows a possible heap snippet. External objects are red; internal objects are green. Each object has a number, followed by an abbreviated class name: o_1 , o_2 and o_5 are a `Shop`, an `Inventory`, and an external object. Curved arrows indicate field values: o_1 has three fields, pointing to o_4 , o_5 and o_2 . Fields denote direct access. The transitive closure of direct access gives indirect (transitive) access: o_1 has direct access to o_4 , and indirect access to o_6 . Object o_6 – highlighted with a dark outline – is the key capability that allows withdrawal from o_4 .

The critical point in our code is the external call on line 8, where the `Shop` asks the buyer to pay the price of that item, by calling `pay` on `buyer` and passing the `Shop`'s account as an argument. As `buyer` is an external object, the module M_{shop} has no method specification for `pay`, and no certainty about what its implementation might do.

What are the possible effects of that external call? At line 9, the `Shop` hopes the buyer will have deposited the price into its account, but needs to be certain the buyer cannot have emptied that account instead. Can the `Shop` be certain? Indeed, if



²As in Joe-E, we leverage module-based privacy to restrict propagation of capabilities, and reduce the need for reference monitors etc, c.f. Sect 3 in [80].

³This is a critical distinction from e.g. cooperative approaches such as *rely/guarantee* [52, 114].

⁴Thus our approach is also applicable to intra-language safety.

- (A) Prior to the call of `buy`, the `buyer` has no eventual access to the account's key, — and
- (B) M_{shop} ensures that
 - (a) access to keys is not leaked to external objects, — and
 - (b) funds cannot be withdrawn unless the external entity responsible for the withdrawal (eventually) has access to the account's key,

— then

- (C) The external call on line 8 can never result in a decrease in the shop's account balance.

The remit of this paper is to provide specification and verification tools that support arguments like the one above. This gives rise to the following two challenges: 1st: A specification language which describes access to capabilities and limited effects, 2nd: A Hoare Logic for adherence to such specifications.

2.1 1st Challenge: Specification Language

We want to give a formal meaning to the guarantee that for some effect, E , and an object o_c which is the capability for E :

E (e.g. the account's balance decreases) can be caused only by external objects calling

- (*) methods on internal objects,
and only if the causing object has access to o_c (e.g. the key).

The first task is to describe that effect E took place: if we find some assertion A (e.g. balance is \geq some value b) which is invalidated by E , then, (*) can be described by something like:

- (**) If A holds, and no external access to o_c then A holds in the future.

We next make more precise that “no external access to o_c ”, and that “ A holds in the future”.

In a first attempt, we could say that “no external access to o_c ” means that no external object exists, nor will any external objects be created. This is too strong, however: it defines away the problem we are aiming to solve.

In a second attempt, we could say that “no external access to o_c ” means that no external object has access to o_c , nor will ever get access to o_c . This is also too strong, as it would preclude E from ever happening, while our remit is that E may happen but only under certain conditions.

This discussion indicates that the lack of external access to o_c is not a global property, and that the future in which A will hold is not permanent. Instead, they are both defined *from the perspective of the current point of execution*.

Thus:

- If A holds, and no external object *reachable from the current point of execution* has access to o_c ,
- (***) and no internal objects pass o_c to external objects,
then A holds in *the future scoped by the current point of execution*.

We will shortly formalize “reachable from the current point of execution” as *protection* in §2.1.1, and then “future scoped by the current point of execution” as *scoped invariants* in §2.1.2. Both of these definitions are in terms of the “current point of execution”:

The Current Point of Execution is characterized by the heap, and the activation frame of the currently executing method. Activation frames (frames for short) consist of a variable map and a continuation – the statements remaining to be executed in that method. Method calls push frames onto the stack of frames; method returns pop frames off. The frame on top of the stack (the most recently pushed frame) belongs to the currently executing method.

Fig. 1 illustrates the current point of execution. The left pane, σ_1 , shows a state with the same heap as earlier, and where the top frame is ϕ_1 – it could be the state before a call to `buy`. The middle pane, σ_2 , is a state where we have pushed ϕ_2 on top of the stack of σ_1 – it could be a state during

execution of `buy`. The right pane, σ_3 , is a state where we have pushed ϕ_3 on top of the stack of σ_2 – it could be a state during execution of `pay`.

States whose top frame has a receiver (`this`) which is an internal object are called *internal states*, the other states are called *external states*. In Fig 1, states σ_1 and σ_2 are internal, and σ_3 is external.

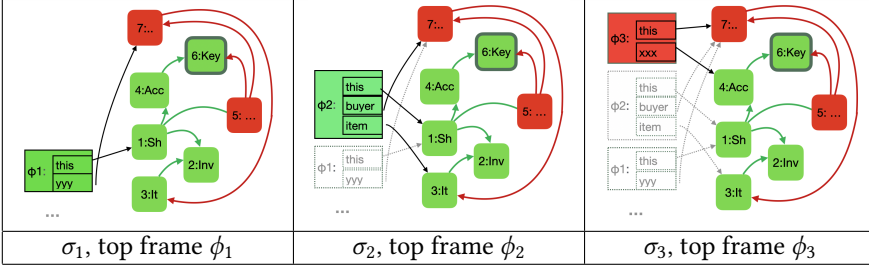


Fig. 1. *The current point of execution before buy, during buy, and during pay.* Frames ϕ_1 , ϕ_2 are green as their receiver (`this`) is internal; ϕ_3 is red as its receiver is external. Continuations are omitted.

2.1.1 Protection.

Protection Object o is *protected from* object o' , formally $\langle o \rangle \leftarrow^* o'$, if no external object indirectly accessible from o' has direct access to o . Object o is *protected*, formally $\langle o \rangle$, if no external object indirectly accessible from the current frame has direct access to o , and if the receiver is external then o is not an argument.⁵ More in Def. 4.4.

Fig. 2 illustrates *protected* and *protected from*. Object o_6 is not protected in states σ_1 and σ_2 , but is protected in state σ_3 . This is so, because the external object o_5 is indirectly accessible from the top frame in σ_1 and in σ_2 , but not from the top frame in σ_3 – in general, calling a method (pushing a frame) can only ever *decrease* the set of indirectly accessible objects. Object o_4 is protected in states σ_1 and σ_2 , and not protected in state σ_3 because though neither object o_5 nor o_7 have direct access to o_4 , in state σ_3 the receiver is external and o_4 is one of the arguments.

heap	σ_1	σ_2	σ_3
$\dots \models \langle o_6 \rangle \leftarrow^* o_4$	$\sigma_1 \not\models \langle o_6 \rangle$	$\sigma_2 \not\models \langle o_6 \rangle$	$\sigma_3 \models \langle o_6 \rangle$
$\dots \not\models \langle o_6 \rangle \leftarrow^* o_5$	$\sigma_1 \models \langle o_4 \rangle$	$\sigma_2 \models \langle o_4 \rangle$	$\sigma_3 \not\models \langle o_4 \rangle$

Fig. 2. *Protected from and Protected.* – continuing from Fig. 1.

If a protected object o is never passed to external objects (*i.e.* never leaked) then o will remain protected during the whole execution of the current method, including during any nested calls. This is the case even if o was not protected before the call to the current method. We define *scoped invariants* to describe such guarantees that properties are preserved within the current call and all nested calls.

⁵An object has direct access to another object if it has a field pointing to the latter; it has indirect access to another object if there exists a sequence of field accesses (direct references) leading to the other object; an object is indirectly accessible from the frame if one of the frame's variables has indirect access to it.

2.1.2 *Scoped Invariants.* We build on the concept of history invariants [27, 67, 69] and define:

Scoped invariants $\forall \bar{x} : \bar{C}. \{A\}$ expresses that if an external state σ has objects \bar{x} of class \bar{C} , and satisfies A , then all external states which are part of the *scoped future* of σ will also satisfy A . The scoped future contains all states which can be reached through any program execution steps, including further method calls and returns, but stopping just before returning from the call active in σ ⁶ – c.f. Def 3.2.

Fig. 3 shows the states of an unspecified execution starting at internal state σ_3 and terminating at internal state σ_{24} . It distinguishes between steps within the same method (\rightarrow), method calls (\uparrow), and method returns (\downarrow). The scoped future of σ_6 consists of σ_6 - σ_{21} , but does not contain σ_{22} onwards, since scoped future stops before returning. Similarly, the scoped future of σ_9 consists of σ_9 , σ_{10} , σ_{11} , σ_{12} , σ_{13} , and σ_{14} , and does not include, e.g., σ_{15} , or σ_{18} .

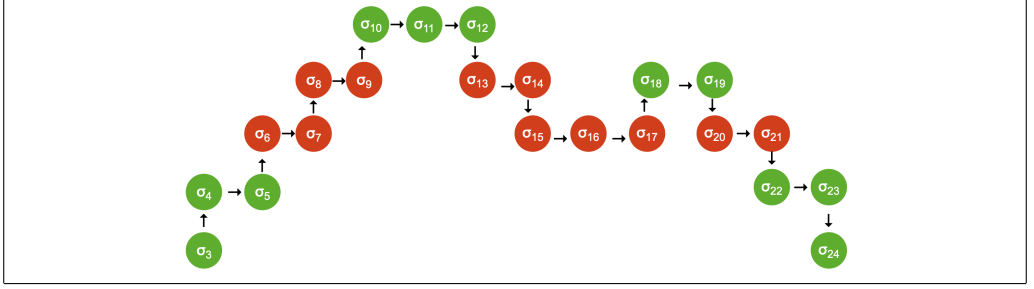


Fig. 3. *Execution.* Green disks represent internal states; red disks external states.

The scoped invariant $\forall \bar{x} : \bar{C}. \{A_0\}$ guarantees that if A_0 holds in σ_8 , then it will also hold in σ_9 , σ_{13} , and σ_{14} ; it doesn't have to hold in σ_{10} , σ_{11} , and σ_{12} as these are internal states. Nor does it have to hold at σ_{15} as it is not part of σ_9 's scoped future.

Example 2.1. The following scoped invariants

$$S_1 \triangleq \forall a : \text{Account}. \{ \langle a \rangle \} \quad S_2 \triangleq \forall a : \text{Account}. \{ \langle a.\text{key} \rangle \}$$

$$S_3 \triangleq \forall a : \text{Account}, b : \text{int}. \{ \langle a.\text{key} \rangle \wedge a.\text{blnce} \geq b \}$$

guarantee that accounts are not leaked (S_1), keys are not leaked (S_2), and that the balance does not decrease unless there is unprotected access to the key (S_3).

This example illustrates three crucial properties of our invariants:

Conditional: They are *preserved*, but unlike object invariants, they do not always hold. E.g., buy cannot assume $\langle a.\text{key} \rangle$ holds on entry, but guarantees that if it holds on entry, then it will still hold on exit.

Scoped: They are preserved during execution of a specific method but not beyond its return. It is, in fact, expected that the invariant will eventually cease to hold after its completion. For instance, while $\langle a.\text{key} \rangle$ may currently hold, it is possible that an earlier (thus quiescent) method invocation frame has direct access to $a.\text{key}$ – without such access, a would not be usable for payments. Once control flow returns to the quiescent method (i.e. enough frames are popped from the stack) $\langle a.\text{key} \rangle$ will no longer hold.

Modular: They describe externally observable effects (e.g. key stays protected) across whole modules, rather than the individual methods (e.g. `set`) making up a module's interface. Our

⁶Here lies the difference to history invariants, which consider *all* future states, including returning from the call active in σ .

example specifications will characterize *any* module defining accounts with a `blnce` and a `key` – even as ghost fields – irrespective of their APIs.

Example 2.2. We now use the features from the previous section to specify methods.

$$S_4 \triangleq \{ \langle \text{this.accnt.key} \rangle \leftarrow * \text{buyer} \wedge \text{this.accnt.blnce} = b \}$$

```
public Shop :: buy(buyer : external, anItem : Item)
{ this.accnt.blnce ≥ b } || { ... }
```

S_4 guarantees that if the key was protected from `buyer` before the call, then the balance will not decrease⁷. It does *not* guarantee `buy` will only be called when $\langle \text{this.accnt.key} \rangle \leftarrow * \text{buyer}$ holds: as a public method, `buy` can be invoked by external code that ignores all specifications.

Example 2.3. We illustrate the meaning of our specifications using three versions (M_{good} , M_{bad} , and M_{fine}) of the M_{shop} module [74]; these all share the same transfer method to withdraw money:

```
1 module Mgood
2   class Shop ... as earlier ...
3   class Account
4     field blnce:int
5     field key:Key
6     public method transfer(dest:Account, key':Key, amt:nat)
7       if (this.key==key') this.blnce-=amt; dest.blnce+=amt
8     public method set(key':Key)
9       if (this.key==null) this.key=key'
```

Now consider modules M_{bad} and M_{fine} , which differ from M_{good} only in their `set` methods. Whereas M_{good} 's key is fixed once it is `set`, M_{bad} allows any client to `set` an account's key at any time, while M_{fine} requires the existing key in order to replace it.

<pre>1 Mbad 2 public method set(key':Key) 3 this.key=key'</pre>	<pre>1 Mfine 2 public method set(key',key'':Key) 3 if (this.key==key') this.key=key''</pre>
-------------------------------------------------------------------	-----------------------------------------------------------------------------------------------

Thus, in all three modules, the key is a capability which *enables* the withdrawal of the money. Moreover, in M_{good} and M_{fine} , the key capability is a necessary precondition for withdrawal of money, while in M_{bad} it is not. Using M_{bad} , it is possible to start in a state where the account's key is unknown, modify the key, and then withdraw the money. Code such as

```
k=new Key; acc.set(k); acc.transfer(rogue_accnt,k,1000)
```

is enough to drain `acc` in M_{bad} without knowing the key. Even though `transfer` in M_{bad} is “safe” when considered in isolation, it is not safe when considered in conjunction with other methods from the same module.

M_{good} and M_{fine} satisfy S_2 and S_3 , while M_{bad} satisfies neither. So if M_{bad} was required to satisfy either S_2 or S_3 then it would be rejected by our inference system as not safe. None of the three versions satisfy S_1 because `pay` could leak an `Account`.

2.2 2nd Challenge: A Hoare logic for adherence to specifications

Hoare Quadruples. Scoped invariants require quadruples, rather than classical triples. Specifically,

$$\forall x : \bar{C}. \{A\}$$

asserts that if an external state σ satisfies $x : \bar{C} \wedge A$, then all its *scoped* external future states will also satisfy A . For example, if σ was an external state executing a call to `Shop :: buy`, then a *scoped* external future state could be reachable during execution of the call `pay`. This implies that we consider not only states at termination but also external states reachable *during* execution of

⁷We ignore the ... for the time being.

statements. To capture this, we extend traditional Hoare triples to quadruples of form

$$\{A\} \text{ stmt } \{A'\} \parallel \{A''\}$$

promising that if a state satisfies A and executes stmt , any terminating state will satisfy A' , and any intermediate external states reachable during execution of stmt satisfy A'' – c.f. Def. 5.2.

We assume an underlying Hoare logic of triples $M \vdash_{ul} \{A\} \text{ stmt } \{A'\}$, which does not have the concept of protection – that is, the assertions A in the \vdash_{ul} -triples do not mention protection. We then embed the \vdash_{ul} -logic into the quadruples logic (\vdash_{ul} -triples whose statements do not contain method calls give rise to quadruples in our logic – see rule below). We extend assertions A so they may mention protection and add rules about protection (e.g. newly created objects are protected – see rule below), and add the usual conditional and substructural rules. More in Fig.7 and 16.

$$\frac{M \vdash_{ul} \{A\} \text{ stmt } \{A'\} \quad \text{stmt calls no methods}}{M \vdash \{A\} \text{ stmt } \{A'\} \parallel \{A''\}} \quad \frac{}{M \vdash \{true\} u = \text{new } C \{ \langle u \rangle \} \parallel \{A\}}$$

Well-formed modules. A module is well-formed, if its specification is well-formed, its public methods preserve the module’s scoped invariants, and all methods satisfy their specifications – c.f. Fig. 9. A well-formed specification does not mention protection in negative positions (this is needed for the soundness of the method call rules). A method satisfies scoped invariants (or method specification) if its body satisfies the relevant pre- and post-conditions. E.g., to prove that $\text{Shop} : : \text{buy}$ satisfies S_3 , taking stmts_b for the body of buy , we have to prove:

$$\begin{aligned} & \{A_0 \wedge \langle a.\text{key} \rangle \wedge a.\text{blnce} \geq b\} \\ & \quad \text{stmts}_b \\ & \{ \langle a.\text{key} \rangle \wedge a.\text{blnce} \geq b \} \parallel \{ \langle a.\text{key} \rangle \wedge a.\text{blnce} \geq b \} \end{aligned}$$

where $A_0 \triangleq \text{this}:\text{Shop}, \text{buyer}:\text{external}, \text{anItem}:\text{Item}, a:\text{Account}, b:\text{int}$.

External Calls. The proof that a method body satisfies pre- and post-conditions uses the Hoare logic discussed in this section. The treatment of external calls is of special interest. For example, consider the verification of S_4 . The challenge is how to reason about the external call on line 8 (from buy in Shop). We need to establish the Hoare quadruple:

$$\begin{aligned} & \{ \text{buyer}:\text{ext1} \wedge \langle \text{this}.\text{acct}.\text{key} \rangle \leftarrow \text{buyer} \wedge \text{this}.\text{acct}.\text{blnce} = b \} \\ (1) \quad & \text{buyer}.\text{pay}(\text{this}.\text{acct}, \text{price}) \\ & \{ \text{this}.\text{acct}.\text{blnce} \geq b \} \parallel \{ \dots \} \end{aligned}$$

which says that if the shop’s account’s key is protected from buyer , then the account’s balance will not decrease after the call.

To prove (1), we aim to use S_3 , but this is not straightforward: S_3 requires $\langle \text{this}.\text{acct}.\text{key} \rangle$, which is not provided by the precondition of (1). More alarmingly, $\langle \text{this}.\text{acct}.\text{key} \rangle$ may *not hold* at the time of the call. For example, in state σ_2 (Fig. 2), which could initiate the call to pay , we have $\sigma_2 \models \langle o_4.\text{key} \rangle \leftarrow o_7$, but $\sigma_2 \not\models \langle o_4.\text{key} \rangle$.

Fig. 2 provides insights into addressing this issue. Upon entering the call, in state σ_3 , we find that $\sigma_3 \models \langle o_4.\text{key} \rangle$. More generally, if $\langle \text{this}.\text{acct}.\text{key} \rangle \leftarrow \text{buyer}$ holds before the call to pay , then $\langle \text{this}.\text{acct}.\text{key} \rangle$ holds upon entering the call. This is because any objects indirectly accessible during pay must have been indirectly accessible from the call’s receiver (buyer) or arguments ($\text{this}.\text{acct}$ and price) when pay was called.

In general, if $\langle x \rangle \leftarrow y_i$ holds for all y_i , before a call $y_0.m(y_1, \dots, y_n)$, then $\langle x \rangle$ holds upon entering the call. Here we have $\langle \text{this}.\text{acct}.\text{key} \rangle \leftarrow \text{buyer}$ by precondition. We also have that price is a scalar and therefore $\langle \text{this}.\text{acct}.\text{key} \rangle \leftarrow \text{price}$. And the type information gives that all fields transitively accessible from an Account are scalar or internal; this gives that $\langle \text{this}.\text{acct}.\text{key} \rangle \leftarrow \text{this}.\text{acct}$. This enables the application of S_3 in (1). The corresponding Hoare logic rule is shown in Fig. 8.

Summary

In an open world, external objects can execute arbitrary code, invoke any public internal methods, access any other external objects, and even collude with each another. The external code may be written in the same or a different programming language than the internal code – all we need is that the platform protects direct external read/write of the internal private fields, while allowing indirect manipulation through calls of public methods.

The conditional and scoped nature of our invariants is critical to their applicability. Protection is a local condition, constraining accessible objects rather than imposing a structure across the whole heap. Scoped invariants are likewise local: they do not preclude some effects from the whole execution of a program, rather the effects are precluded only in some local contexts. While `a.blnc` may decrease in the future, this can only happen in contexts where an external object has direct access to `a.key`. Enforcing these local conditions is the responsibility of the internal module: precisely because these conditions are local, they can be enforced locally within a module, irrespective of all the other modules in the open world.

3 THE UNDERLYING PROGRAMMING LANGUAGE \mathcal{L}_{ul}

3.1 \mathcal{L}_{ul} syntax and runtime configurations

This work is based on \mathcal{L}_{ul} , a minimal, imperative, sequential, class based, typed, object-oriented language. We believe, however, that the work can easily be adapted to any capability safe language with some form of encapsulation, and that it can also support inter-language security, provided that the platform offers means to protect a module's private state; cf capability-safe hardware as in Cheri [31]. Wrt to encapsulation and capability safety, \mathcal{L}_{ul} supports private fields, private and public methods, unforgeable addresses, and no ambient authority (no static methods, no address manipulation). To reduce the complexity of our formal models, as is usually done, e.g. [35, 57, 97], \mathcal{L}_{ul} lacks many common languages features, omitting static fields and methods, interfaces, inheritance, subsumption, exceptions, and control flow. In our examples, we use numbers and booleans – these can be encoded.

Fig. 4 shows the \mathcal{L}_{ul} syntax. Statements, $stmt$, are three-address instructions, method calls, or empty, ϵ . Expressions, e , are ghost code; as such, they may appear in assertions but not in statements, and have no side-effects [22, 45]. Expressions may contain fields, $e.f$, or ghost-fields, $e.gf(\bar{e})$. The meaning of e is module-dependent; e.g. `a.blnc` is a field lookup in M_{good} , but in a module which stores balances in a table it would be a recursive lookup through that table – c.f. example in §A.3.⁸ In line with most tools, we support ghost-fields, but they are not central to our work.

\mathcal{L}_{ul} states, σ , consist of a heap χ and a stack. A stack is a sequence of frames, $\phi_1 \dots \phi_n$. A frame, ϕ , consists of a local variable map and a continuation, i.e. the statements to be executed. The top frame, i.e. the frame most recently pushed onto the stack, in a state $(\phi_1 \dots \phi_n, \chi)$ is ϕ_n .

Notation. We adopt the following unsurprising notation:

- An object is uniquely identified by the address that points to it. We shall be talking of objects o, o' when talking less formally, and of addresses, $\alpha, \alpha', \alpha_1, \dots$ when more formal.
- x, x', y, z, u, v, w are variables; $dom(\phi)$ and $Rng(\phi)$ indicate the variable map in ϕ ; $dom(\sigma)$ and $Rng(\sigma)$ indicate the variable map in the top frame in σ
- $\alpha \in \sigma$ means that α is defined in the heap of σ , and $x \in \sigma$ means that $x \in dom(\sigma)$. Conversely, $\alpha \notin \sigma$ and $x \notin \sigma$ have the obvious meanings. $[\alpha]_\sigma$ is α ; and $[x]_\sigma$ is the value to which x is mapped in the top-most frame of σ 's stack, and $[e.f]_\sigma$ looks up in σ 's heap the value of f for the object $[e]_\sigma$.

⁸For convenience, $e.gf$ is short for $e.gf()$. Thus, $e.gf$ may be simple field lookup in some modules, or ghost-field in others.

442	$Mdl ::= \overline{C} \mapsto CDef$	Module Def.	$fld ::= \text{field } f : T$	Field Def.
443	$CDef ::= \text{class } C \{ \overline{fld}; \overline{mth}; \overline{gfld}; \}$	Class Def.	$T ::= C$	Type
444	$mth ::= p \text{ method } m (\overline{x:T}): T \{ s \}$	Method Def.	$p ::= \text{private} \mid \text{public}$	Privacy
445				
446	$stmt ::= x := y \mid x := v \mid x := y.f \mid x.f := y \mid x := y_0.m(\overline{y}) \mid \text{new } C \mid stmt; stmt \mid \epsilon$			Statement
447	$gfld ::= \text{ghost } gf(\overline{x:T}) \{ e \} : T$			Ghost Field Def.
448	$e ::= x \mid v \mid e.f \mid e.gf(\overline{e})$			Expression
449				
450	$\sigma ::= (\overline{\phi}, \chi)$	Program State	$C, f, m, gf, x, y ::= \text{Identifier}$	
451	$\phi ::= (\overline{x \mapsto v}; s)$	Frame	$o ::= (C; \overline{f \mapsto v})$	Object
452	$\chi ::= (\overline{\alpha \mapsto o})$	Heap	$v ::= \alpha \mid \text{null}$	Value

Fig. 4. \mathcal{L}_{ul} Syntax. We use x, y, z for variables, C, D for class identifiers, f for field identifier, gf for ghost field identifiers, m for method identifiers, α for addresses.

- $\phi[x \mapsto \alpha]$ updates the variable map of ϕ , and $\sigma[x \mapsto \alpha]$ updates the top frame of σ . $A[e/x]$ is textual substitution where we replace all occurrences of x in A by e .
- As usual, \overline{q} stands for sequence q_1, \dots, q_n , where q can be an address, a variable, a frame, an update or a substitution. Thus, $\sigma[\overline{x \mapsto \alpha}]$ and $A[\overline{e/y}]$ have the expected meaning.
- $\phi.\text{cont}$ is the continuation of frame ϕ , and $\sigma.\text{cont}$ is the continuation in the top frame.
- $\text{text}_1 \stackrel{\text{txt}}{=} \text{text}_2$ expresses that text_1 and text_2 are the same text.
- We define the depth of a stack as $|\phi_1 \dots \phi_n| \triangleq n$. For states, $|(\overline{\phi}, \chi)| \triangleq |\overline{\phi}|$. The operator $\sigma[k]$ truncates the stack up to the k -th frame: $(\phi_1 \dots \phi_k \dots \phi_n, \chi)[k] \triangleq (\phi_1 \dots \phi_k, \chi)$
- $Vs(stmt)$ returns the variables which appear in $stmt$. For example, $Vs(u := y.f) = \{u, y\}$.

3.2 \mathcal{L}_{ul} Execution

Fig. 5 describes \mathcal{L}_{ul} execution by a small steps operational semantics with shape $\overline{M}; \sigma \mapsto \sigma'$. \overline{M} stands for one or more modules, where a module, M , maps class names to class definitions. The functions $\text{classOf}(\sigma, x)$, $\text{Meth}(\overline{M}, C, m)$, $\text{fields}(\overline{M}, C)$, $\text{SameModule}(x, y, \sigma, \overline{M})$, and $\text{Prms}(\sigma, \overline{M})$, return the class of x , the method m for class C , the fields for class C , whether x and y belong to the same module, and the formal parameters of the method currently executing in σ – c.f. Defs A.2 – A.7. Initial states, $\text{Initial}(\sigma)$, contain a single frame with single variable `this` pointing to a single object in the heap and a continuation, c.f. A.8.

The semantics is unsurprising: The top frame's continuation ($\sigma.\text{cont}$) contains the statement to be executed next. We dynamically enforce a simple form of module-wide privacy: Fields may be read or written only if they belong to an object (here y) whose class comes from the same module as the class of the object reading or writing the fields (`this`).⁹ Wlog, to simplify some proofs we require, as in Kotlin, that method bodies do not assign to formal parameters.

Private methods may be called only if the class of the callee (the object whose method is being called – here y_0) comes from the same module as the class of the caller (here `this`). Public methods may always be called. When a method is called, a new frame is pushed onto the stack; this frame maps `this` and the formal parameters to the values for the receiver and other arguments, and the continuation to the body of the method. Method bodies are expected to store their return values in the implicitly defined variable `res`¹⁰. When the continuation is empty (ϵ), the frame is popped and the value of `res` from the popped frame is stored in the variable map of the top frame.

⁹More fine-grained privacy, e.g. C++ private fields or ownership types, would provide all the guarantees needed in our work.

¹⁰For ease of presentation, we omit assignment to `res` in methods returning `void`.

$$\begin{array}{c}
\text{491} \quad \frac{\sigma.\text{cont} \stackrel{\text{txt}}{=} x := y.f; \text{stmt} \quad x \notin \text{Prms}(\sigma, \bar{M}) \quad \text{SameModule}(\text{this}, y, \sigma, \bar{M})}{\bar{M}, \sigma \rightarrow \sigma[x \mapsto \lfloor y.f \rfloor_\sigma][\text{cont} \mapsto \text{stmt}]} \quad (\text{READ}) \\
\text{492} \\
\text{493} \\
\text{494} \quad \frac{\sigma.\text{cont} \stackrel{\text{txt}}{=} x.f := y; \text{stmt} \quad \text{SameModule}(\text{this}, x, \sigma, \bar{M})}{\bar{M}, \sigma \rightarrow \sigma[\lfloor x \rfloor_\sigma.f \mapsto \lfloor y \rfloor_\sigma][\text{cont} \mapsto \text{stmt}]} \quad (\text{WRITE}) \\
\text{495} \\
\text{496} \\
\text{497} \quad \frac{\sigma.\text{cont} \stackrel{\text{txt}}{=} x := \text{new } C; s \quad x \notin \text{Prms}(\sigma, \bar{M}) \quad \text{fields}(\bar{M}, C) = \bar{f} \quad \alpha \text{ fresh in } \sigma}{\bar{M}, \sigma \rightarrow \sigma[x \mapsto \alpha][\alpha \mapsto (C; \bar{f} \mapsto \text{null})][\text{cont} \mapsto s]} \quad (\text{NEW}) \\
\text{498} \\
\text{499} \\
\text{500} \quad \frac{\phi_n.\text{cont} \stackrel{\text{txt}}{=} u := y_0.m(\bar{y}); _ \quad u \notin \text{Prms}((\bar{\phi} \cdot \phi_n, \chi), \bar{M})}{\text{Meth}(\bar{M}, \text{classOf}((\phi_n, \chi), y_0, m) = p \text{ C}::m(x:T):T\{\text{stmt}\} \quad p = \text{public} \vee \text{SameModule}(\text{this}, y_0, (\phi_n, \chi), \bar{M}))}{\bar{M}, (\bar{\phi} \cdot \phi_n, \chi) \rightarrow (\bar{\phi} \cdot \phi_n \cdot (\text{this} \mapsto \lfloor y_0 \rfloor_{\phi_n}, x \mapsto \lfloor y \rfloor_{\phi_n}; \text{stmt}), \chi)} \quad (\text{CALL}) \\
\text{501} \\
\text{502} \\
\text{503} \\
\text{504} \quad \frac{\phi_{n+1}.\text{cont} \stackrel{\text{txt}}{=} \epsilon \quad \phi_n.\text{cont} \stackrel{\text{txt}}{=} x := y_0.m(\bar{y}); \text{stmt}}{\bar{M}, (\bar{\phi} \cdot \phi_n \cdot \phi_{n+1}, \chi) \rightarrow (\bar{\phi} \cdot \phi_n[x \mapsto \lfloor \text{res} \rfloor_{\phi_{n+1}}][\text{cont} \mapsto \text{stmt}], \chi)} \quad (\text{RETURN}) \\
\text{505} \\
\text{506}
\end{array}$$

Fig. 5. \mathcal{L}_{ul} operational Semantics

Thus, when $\bar{M}; \sigma \rightarrow \sigma'$ is within the same method we have $|\sigma'| = |\sigma|$; when it is a call we have $|\sigma'| = |\sigma| + 1$; and when it is a return we have $|\sigma'| = |\sigma| - 1$. Fig. 3 from §2 distinguishes \rightarrow execution steps into: steps within the same call (\rightarrow), entering a method (\uparrow), returning from a method (\downarrow). Therefore $\bar{M}; \sigma_8 \rightarrow \sigma_9$ is a step within the same call, $\bar{M}; \sigma_9 \rightarrow \sigma_{10}$ is a method entry with \bar{M} ; $\sigma_{12} \rightarrow \sigma_{13}$ the corresponding return. In general, $\bar{M}; \sigma \rightarrow^* \sigma'$ may involve any number of calls or returns: e.g. $\bar{M}; \sigma_{10} \rightarrow^* \sigma_{15}$, involves no calls and two returns.

3.3 Fundamental Concepts

The novel features of our assertions — protection and scoped invariants — are both defined in terms of the current point of execution. Therefore, for the semantics of our assertions we need to represent calls and returns, scoped execution, and (in)directly accessible objects.

3.3.1 Method Calls and Returns. These are characterized through pushing/popping frames : $\sigma \nabla \phi$ pushes frame ϕ onto the stack of σ , while $\sigma \Delta$ pops the top frame and updates the continuation and variable map.

Definition 3.1. Given a state σ , and a frame ϕ , we define

- $\sigma \nabla \phi \triangleq (\bar{\phi} \cdot \phi, \chi) \quad \text{if } \sigma = (\bar{\phi}, \chi).$
- $\sigma \Delta \triangleq (\bar{\phi} \cdot (\phi_n[\text{cont} \mapsto \text{stmt}][x \mapsto \lfloor \text{res} \rfloor_{\phi_n}], \chi) \quad \text{if}$
 $\sigma = (\bar{\phi} \cdot \phi_n \cdot \phi_{n+1}, \chi), \text{ and } \phi_n(\text{cont}) \stackrel{\text{txt}}{=} x := y_0.m(\bar{y}); \text{stmt}$

Consider Fig. 3 again: $\sigma_8 = \sigma_7 \nabla \phi$ for some ϕ , and $\sigma_{15} = \sigma_{14} \Delta$.

3.3.2 Scoped Execution. In order to give semantics to scoped invariants (introduced in §2.1.2 and to be fully defined in Def. 5.4), we need a new definition of execution, called *scoped execution*.

Definition 3.2 (Scoped Execution). Given modules \bar{M} , and states $\sigma, \sigma_1, \sigma_n$, and σ' , we define:

- $\bar{M}; \sigma \rightsquigarrow \sigma' \triangleq \bar{M}; \sigma \rightarrow \sigma' \wedge |\sigma| \leq |\sigma'|$
- $\bar{M}; \sigma_1 \rightsquigarrow^* \sigma_n \triangleq \sigma_1 = \sigma_n \vee \exists \sigma_2, \dots, \sigma_{n-1}. \forall i \in [1..n) [\bar{M}; \sigma_i \rightarrow \sigma_{i+1} \wedge |\sigma_1| \leq |\sigma_{i+1}|]$
- $\bar{M}; \sigma \rightsquigarrow_{fin}^* \sigma' \triangleq \bar{M}; \sigma \rightsquigarrow^* \sigma' \wedge |\sigma| = |\sigma'| \wedge \sigma'.\text{cont} = \epsilon$

Consider Fig. 3 : Here $|\sigma_8| \leq |\sigma_9|$ and thus $\bar{M}; \sigma_8 \rightsquigarrow \sigma_9$. Also, $\bar{M}; \sigma_{14} \twoheadrightarrow \sigma_{15}$ but $|\sigma_{14}| \not\leq |\sigma_{15}|$ (this step returns from the active call in σ_{14}), and hence $\bar{M}; \sigma_{14} \not\rightsquigarrow \sigma_{15}$. Finally, even though $|\sigma_8| = |\sigma_{18}|$ and $\bar{M}; \sigma_8 \twoheadrightarrow^* \sigma_{18}$, we have $\bar{M}; \sigma_8 \not\rightsquigarrow^* \sigma_{18}$. This is so, because the execution $\bar{M}; \sigma_8 \twoheadrightarrow^* \sigma_{18}$ goes through the step $\bar{M}; \sigma_{14} \twoheadrightarrow \sigma_{15}$ and $|\sigma_8| \not\leq |\sigma_{15}|$ (this step returns from the active call in σ_8).

The relation \rightsquigarrow^* contains more than the transitive closure of \rightsquigarrow . E.g., $\bar{M}; \sigma_9 \rightsquigarrow^* \sigma_{13}$, even though $\bar{M}; \sigma_9 \rightsquigarrow \sigma_{12}$ and $\bar{M}; \sigma_{12} \not\rightsquigarrow^* \sigma_{13}$. Lemma 3.3 says that the value of the parameters does not change during execution of the same method. Appendix B discusses proofs, and further properties.

Lemma 3.3. For all \bar{M}, σ, σ' : $\bar{M}; \sigma \rightsquigarrow^* \sigma' \wedge |\sigma| = |\sigma'| \implies \forall x \in \text{Prms}(\bar{M}, \sigma). [x]_\sigma = [x]_{\sigma'}$

3.3.3 Reachable Objects, Locally Reachable Objects, and Well-formed States. To define protection (no external object indirectly accessible from the top frame has access to the protected object, c.f. § 2.1.1) we first define reachability. An object α is *locally reachable*, i.e. $\alpha \in \text{LocRchbl}(\sigma)$, if it is reachable from the top frame on the stack of σ .

Definition 3.4. We define

- $\text{Rchbl}(\alpha, \sigma) \triangleq \{ \alpha' \mid \exists n \in \mathbb{N}. \exists f_1, \dots, f_n. [\alpha.f_1 \dots f_n]_\sigma = \alpha' \}.$
- $\text{LocRchbl}(\sigma) \triangleq \{ \alpha \mid \exists x \in \text{dom}(\sigma) \wedge \alpha \in \text{Rchbl}([x]_\sigma, \sigma) \}.$

In well-formed states, $\bar{M} \models \sigma$, the value of a parameter in any callee ($\sigma[k]$) is also the value of some variable in the caller ($\sigma[k-1]$), and any address reachable from any frame ($\text{LocRchbl}(\sigma[k])$) is reachable from some formal parameter of that frame.

Definition 3.5 (Well-formed states). For modules \bar{M} , and states σ, σ' :

$$\begin{aligned} \bar{M} \models \sigma \triangleq & \forall k \in \mathbb{N}. [1 < k \leq |\sigma| \implies \\ & [\forall x \in \text{Prms}(\sigma[k], \bar{M}). [\exists y. [x]_{\sigma[k]} = [y]_{\sigma[k-1]}] \quad \wedge \\ & \text{LocRchbl}(\sigma[k]) = \bigcup_{z \in \text{Prms}(\sigma[k], \bar{M})} \text{Rchbl}([z]_{\sigma[k]}, \sigma) \quad] \end{aligned}$$

Lemma 3.6 says that (1) execution preserves well-formedness, and (2) any object which is locally reachable after pushing a frame was locally reachable before pushing that frame.

Lemma 3.6. For all modules \bar{M} , states σ, σ' , and frame ϕ :

- (1) $\bar{M} \models \sigma \wedge \bar{M}, \sigma \twoheadrightarrow \sigma' \implies \bar{M} \models \sigma'$
- (2) $\sigma' = \sigma \nabla \phi \wedge \bar{M} \models \sigma' \implies \text{LocRchbl}(\sigma') \subseteq \text{LocRchbl}(\sigma)$

4 ASSERTIONS

Our assertions are standard (e.g. properties of the values of expressions, connectives, quantification etc.) or about protection (i.e. $\langle e \rangle \leftarrow e$ and $\langle e \rangle$).

Definition 4.1. Assertions, A , are defined as follows:

$$A ::= e \mid e : C \mid \neg A \mid A \wedge A \mid \forall x : C. A \mid e : \text{extl} \mid \langle e \rangle \leftarrow e \mid \langle e \rangle \quad 11$$

$Fv(A)$ returns the free variables in A ; for example, $Fv(a : \text{Account} \wedge \forall b : \text{int}. [a.\text{blnce} = b]) = \{a\}$.

Definition 4.2 (Shorthands). We write $e : \text{intl}$ for $\neg(e : \text{extl})$, and extl . resp. intl for $\text{this} : \text{extl}$ resp. $\text{this} : \text{intl}$. Forms such as $A \rightarrow A'$, $A \vee A'$, and $\exists x : C. A$ can be encoded.

Satisfaction of Assertions by a module and a state is expressed through $M, \sigma \models A$ and defined by cases on the shape of A , in definitions 4.3 and 4.4. M is used to look up the definitions of ghost fields, and to find class definitions to determine whether an object is external.

¹¹Addresses in assertions as e.g. in $a.\text{blnce} > 700$, are useful when giving semantics to universal quantifiers c.f. Def. 4.3(5), when the local map changes e.g. upon call and return, and in general, for scoped invariants, c.f. Def. 5.4.

4.1 Semantics of assertions – first part

To determine satisfaction of an expression, we use the evaluation relation, $M, \sigma, e \hookrightarrow v$, which says that the expression e evaluates to value v in the context of state σ and module M . Ghost fields may be recursively defined, thus evaluation of e might not terminate. Nevertheless, the logic of assertions remains classical because recursion is restricted to expressions.

Definition 4.3 (Satisfaction of Assertions – first part). We define satisfaction of an assertion A by a state σ with module M as:

- (1) $M, \sigma \models e \triangleq M, \sigma, e \hookrightarrow \text{true}$
- (2) $M, \sigma \models e : C \triangleq M, \sigma, e \hookrightarrow \alpha \wedge \text{classOf}(\alpha, \sigma) = C$
- (3) $M, \sigma \models \neg A \triangleq M, \sigma \not\models A$
- (4) $M, \sigma \models A_1 \wedge A_2 \triangleq M, \sigma \models A_1 \wedge M, \sigma \models A_2$
- (5) $M, \sigma \models \forall x : C. A \triangleq \forall \alpha. [M, \sigma \models \alpha : C \implies M, \sigma \models A[\alpha/x]]$
- (6) $M, \sigma \models e : \text{extl} \triangleq \exists C. [M, \sigma \models e : C \wedge C \notin M]$

Note that while execution takes place in the context of one or more modules, \overline{M} , satisfaction of assertions considers *exactly one* module M – the internal module. M is used to look up the definitions of ghost fields, and to determine whether objects are external.

4.2 Semantics of Assertions - second part

In §2.1.1 we introduced protection – we will now formalize this concept.

An object is protected from another object, $\langle \alpha \rangle \ltimes \alpha_o$, if the two objects are not equal, and no external object reachable from α_o has a field pointing to α . This ensures that the last element on any path leading from α_o to α in an internal object.

An object is protected, $\langle \alpha \rangle$, if no external object reachable from any of the current frame's arguments has a field pointing to α ; and furthermore, if the receiver is external, then no parameter to the current method call directly refers to α . This ensures that no external object reachable from the current receiver or arguments can “obtain” α , where obtain α is either direct access through a field, or by virtue of the method's receiver being able to see all the arguments.

Definition 4.4 (Satisfaction of Assertions – Protection). – continuing definitions in 4.3:

- (1) $M, \sigma \models \langle \alpha \rangle \ltimes \alpha_o \triangleq$
 - (a) $\alpha \neq \alpha_o$,
 - (b) $\forall \alpha'. \forall f. [\alpha' \in \text{Rchbl}(\alpha_o, \sigma) \wedge M, \sigma \models \alpha' : \text{extl} \implies [\alpha'.f]_\sigma \neq \alpha]$.
- (2) $M, \sigma \models \langle \alpha \rangle \triangleq$
 - (a) $M, \sigma \models \text{extl} \implies \forall x \in \sigma. M, \sigma \models x \neq \alpha$,
 - (b) $\forall \alpha'. \forall f. [\alpha' \in \text{LocRchbl}(\sigma) \wedge M, \sigma \models \alpha' : \text{extl} \implies [\alpha'.f]_\sigma \neq \alpha]$.

Moreover,

- (3) $M, \sigma \models \langle e \rangle \ltimes e_o \triangleq \exists \alpha, \alpha_o. [M, \sigma, e \hookrightarrow \alpha \wedge M, \sigma, e_o \hookrightarrow \alpha_o \wedge M, \sigma \models \langle \alpha \rangle \ltimes \alpha_o]$,
- (4) $M, \sigma \models \langle e \rangle \triangleq \exists \alpha. [M, \sigma, e \hookrightarrow \alpha \wedge M, \sigma \models \langle \alpha \rangle]$.

We illustrate “protected” and “protected from” in Fig. 2 in §2. and Fig. 13 in App. C. In general, $\langle \alpha \rangle \ltimes \alpha_o$ ensures that α_o will get access to α only if another object grants that access. Similarly, $\langle \alpha \rangle$ ensures that during execution of the current method, no external object will get direct access to α unless some internal object grants that access¹². Thus, protection together with protection preservation (*i.e.* no internal object gives access) guarantee lack of eventual external access.

¹²This is in line with the motto “only connectivity begets connectivity” from [83].

Discussion. Lack of eventual direct access is a central concept in the verification of code with calls to and callbacks from untrusted code. It has already been over-approximated in several different ways, e.g. 2nd-class [96, 117] or borrowed (“2nd-hand”) references [14, 23], textual modules [74], information flow [111], runtime checks [4], abstract data type exports [70], separation-based invariants Iris [48, 101], – more in § 8. In general, protection is applicable in more situations (i.e. is less restrictive) than most of these approaches, although more restrictive than the ideal “lack of eventual access”.

An alternative definition might consider α as protected from α_o , if any path from α_o to α goes through at least one internal object. With this definition, o_4 would be protected from o_1 in the heap shown here. However, o_1 can make a call to o_2 , and this call could return o_3 . Once o_1 has direct access to o_3 , it can also get direct access to o_4 . The example justifies our current definition.



4.3 Preservation of Assertions

Program logics require some form of framing, i.e. conditions under which satisfaction of assertions is preserved across program execution. This is the subject of the current Section.

Def. 4.5 which turns an assertion A to the equivalent variable-free form by replacing all free variables from A by their values in σ . Then, Lemma 4.5 says that satisfaction of an assertion is not affected by replacing free variables by their values, nor by changing the state’s continuation.

Definition and Lemma 4.5. For all $M, \sigma, stmt, A$, and \bar{x} where $\bar{x} = Fv(A)$:

- $\sigma[A] \triangleq A[\bar{x}/\bar{x}]_{\sigma/\bar{x}}$
- $M, \sigma \models A \iff M, \sigma \models \sigma[A] \iff M, \sigma[\text{cont} \mapsto stmt] \models A$

We now move to assertion preservation across method call and return.

4.3.1 Stability. In most program logics, satisfaction of variable-free assertions is preserved when pushing/popping frames – i.e. immediately after entering a method or returning from it. But this is not so for our assertions, where protection depends on the heap but also on the range of the top frame. E.g., Fig. 2: $\sigma_2 \not\models \langle o_6 \rangle$, but after pushing a frame, we have $\sigma_3 \models \langle o_6 \rangle$.

Assertions which do not contain $\langle _ \rangle$ are called $Stbl(_)$, while assertions which do not contain $\langle _ \rangle$ in *negative* positions are called $Stb^+(_)$. Fig. 6 shows some examples. Lemma 4.6 says that $Stbl(_)$ assertions are preserved when pushing/popping frames, and $Stb^+(_)$ assertions are preserved when pushing internal frames. C.f. Appendix D for definitions and proofs.

Lemma 4.6. For all states σ , frames ϕ , all assertions A with $Fv(A) = \emptyset$

- $Stbl(A) \implies [M, \sigma \models A \iff M, \sigma \nabla \phi \models A]$
- $Stb^+(A) \wedge M \cdot \bar{M} \models \sigma \nabla \phi \wedge M, \sigma \nabla \phi \models \text{int1} \wedge M, \sigma \models A \implies M, \sigma \nabla \phi \models A$

While Stb^+ assertions are preserved when pushing internal frames, they are *not* necessarily preserved when pushing external frames nor when popping frames (c.f. Ex. 4.7).

Example 4.7. Fig. 2 illustrates that

– Stb^+ not necessarily preserved by External Push: Namely, $\sigma_2 \models \langle o_4 \rangle$, pushing frame ϕ_3 with an external receiver and o_4 as argument gives σ_3 , we have $\sigma_3 \not\models \langle o_4 \rangle$.

– Stb^+ not necessarily preserved by Pop: Namely, $\sigma_3 \models \langle o_6 \rangle$, returning from σ_3 would give σ_2 , and we have $\sigma_2 \not\models \langle o_6 \rangle$.

We work with Stb^+ assertions (the $Stbl$ requirement is too strong). But we need to address the lack of preservation of Stb^+ assertions for external method calls and returns. We do the former through *adaptation* ($\neg \nabla$ in Sect 6.2.2), and the latter through *deep satisfaction* (§7).

4.3.2 Encapsulation. As external code is unknown, it could, in principle, have unlimited effect and invalidate any assertion, and thus make reasoning about external calls impossible. However, because fields are private, assertions which read internal fields only, cannot be invalidated by external execution steps. Reasoning about external calls relies on such *encapsulated* assertions.

Judgment $M \vdash \text{Enc}(A)$ from Def D.4, expresses A looks up the contents of internal objects only, does not contain $\langle _ \rangle \leftarrow _$, and does not contain $\langle _ \rangle$ in negative positions. Lemma 4.8 says that $M \vdash \text{Enc}(A)$ says that any external scoped execution step which involves M and any set of other modules \bar{M} , preserves satisfaction of A .

	$z.f \geq 3$	$\langle x \rangle$	$\neg(\langle x \rangle)$	$\langle y \rangle \leftarrow x$	$\neg(\langle y \rangle \leftarrow x)$
$\text{Stbl}(_)$	✓	✗	✗	✓	✓
$\text{Stb}^+(_)$	✓	✓	✗	✓	✓
$\text{Enc}(_)$	✓	✓	✗	✗	✗

Fig. 6. Comparing $\text{Stbl}(_)$, $\text{Stb}^+(_)$, and $\text{Enc}(_)$ assertions.

Lemma 4.8 (Encapsulation). For all modules M , and assertions A :

- $M \vdash \text{Enc}(A) \implies \forall \bar{M}, \sigma, \sigma'. [M, \sigma \models (A \wedge \text{extl}) \wedge \bar{M} \cdot \bar{M}; \sigma \rightsquigarrow \sigma' \implies M, \sigma' \models \mathcal{T}[A]]$

5 SPECIFICATIONS

We now discuss syntax and semantics of our specifications, and illustrate them through examples.

5.1 Syntax, Semantics, Examples

Definition 5.1 (Specifications Syntax). We define the syntax of specifications, S :

$$\begin{aligned}
 S &::= \forall \bar{x} : \bar{C}. \{A\} \mid \{A\} p C :: m(\bar{y} : \bar{C}) \{A\} \parallel \{A\} \mid S \wedge S \\
 p &::= \text{private} \mid \text{public}
 \end{aligned}$$

In Def. 5.6 later on we describe well-formedness of S , but we first discuss semantics and some examples. We use quadruples involving states: $\bar{M}; M \models \{A\} \sigma \{A'\} \parallel \{A''\}$ says that if σ satisfies A , then any terminating scoped execution of its continuation $(\bar{M} \cdot M; \sigma \rightsquigarrow_{\text{fin}}^* \sigma')$ will satisfy A' , and any intermediate reachable external state $(\bar{M} \cdot M; \sigma \rightsquigarrow^* \sigma'')$ will satisfy the “mid-condition”, A'' .

Definition 5.2. For modules \bar{M}, M , state σ , and assertions A, A' and A'' , we define:

- $\bar{M}; M \models \{A\} \sigma \{A'\} \parallel \{A''\} \triangleq \forall \sigma', \sigma''. [$
 $M, \sigma \models A \implies [\bar{M} \cdot M; \sigma \rightsquigarrow_{\text{fin}}^* \sigma' \implies M, \sigma' \models A'] \wedge$
 $[\bar{M} \cdot M; \sigma \rightsquigarrow^* \sigma'' \implies M, \sigma'' \models (\text{extl} \rightarrow \mathcal{T}[A''])]]$

Example 5.3. Consider $\dots; \dots \models \{A_1\} \sigma_4 \{A_2\} \parallel \{A_3\}$ for Fig. 3. It means that if σ_4 satisfies A_1 , then σ_{23} will satisfy A_2 , while σ_6 - σ_9 , σ_{13} - σ_{17} , and σ_{20} - σ_{21} will satisfy A_3 . It does not imply anything about σ_{24} because $\dots; \sigma_4 \not\rightsquigarrow^* \sigma_{24}$. Similarly, if σ_8 satisfies A_1 then σ_{14} will satisfy A_2 , and $\sigma_8, \sigma_9, \sigma_{13}, \sigma_{14}$ will satisfy A_3 , while making no claims about $\sigma_{10}, \sigma_{11}, \sigma_{12}$, nor about σ_{15} onwards.

Now we define $M \models \forall \bar{x} : \bar{C}. \{A\}$ to mean that if an external state σ satisfies A , then all future external states reachable from σ —including nested calls and returns but *stopping* before returning from the active call in σ —also satisfy A . And $M \models \{A_1\} p D :: m(\bar{y} : \bar{D}) \{A_2\} \parallel \{A_3\}$ means that scoped execution of a call to m from D in states satisfying A_1 leads to final states satisfying A_2 (if it terminates), and to intermediate external states satisfying A_3 .

Definition 5.4 (Semantics of Specifications). We define $M \models S$ by cases over S :

- (1) $M \models \overline{\forall x : \overline{C}}. \{A\} \triangleq \overline{\forall \overline{M}, \sigma. [\overline{M}; M \models \{\text{extl} \wedge \overline{x : \overline{C}} \wedge A\} \sigma \{A\} \parallel \{A\}]}.$
- (2) $M \models \{A_1\} p D :: m(\overline{y : D}) \{A_2\} \parallel \{A_3\} \triangleq$
 $\overline{\forall \overline{M}, \sigma, y_0, \overline{y}. [\sigma. \text{cont} \stackrel{\text{txt}}{=} u := y_0.m(y_1, ..y_n) \implies$
 $\overline{M}; M \models \{y_0 : D, y : \overline{D} \wedge A[y_0/\text{this}]\} \sigma \{A_2[u/\text{res}, y_0/\text{this}]\} \parallel \{A_3\}]}$
- (3) $M \models S \wedge S' \triangleq M \models S \wedge M \models S'$

Fig. 3 in §2.1.2 illustrated the meaning of $\overline{\forall x : \overline{C}}. \{A_0\}$. Moreover, $M_{\text{good}} \models S_2 \wedge S_3 \wedge S_4$, and $M_{\text{fine}} \models S_2 \wedge S_3 \wedge S_4$, while $M_{\text{bad}} \not\models S_2$. We continue with some examples – more in Appendix E.

Example 5.5 (Scoped Invariants and Method Specs). S_5 says that non-null keys are immutable:

$$S_5 \triangleq \overline{\forall a : \text{Account}, k : \text{Key}. \{\text{null} \neq k = a.\text{key}\}}$$

S_9 guarantees that `set` preserves the protectedness of any account, and any key.

$$S_9 \triangleq \{a : \text{Account}, a' : \text{Account} \wedge \langle a \rangle \wedge \langle a' \rangle.\text{key}\} \\ \text{public Account} :: \text{set}(\text{key}' : \text{Key}) \\ \{\langle a \rangle \wedge \langle a' \rangle.\text{key}\} \parallel \{\langle a \rangle \wedge \langle a' \rangle.\text{key}\}$$

Note that a, a' are disjoint from `this` and the formal parameters of `set`. In that sense, a and a' are universally quantified; a call of `set` will preserve protectedness for *all* accounts and their keys.

5.2 Well-formedness

We now define what it means for a specification to be well-formed:

Definition 5.6. *Well-formedness* of specifications, $\vdash S$, is defined by cases on S :

- $\vdash \overline{\forall x : \overline{C}}. \{A\} \triangleq Fv(A) \subseteq \{\overline{x}\} \wedge M \vdash \text{Enc}(\overline{x : \overline{C}} \wedge A).$
- $\vdash \{x : \overline{C'} \wedge A\} p C :: m(\overline{y : \overline{C}}) \{A'\} \parallel \{A''\} \triangleq$
 $[\text{res}, \text{this} \notin \overline{x}, \overline{y} \wedge Fv(A) \subseteq \overline{x}, \overline{y}, \text{this} \wedge Fv(A') \subseteq \overline{x}, \overline{y}, \text{this}, \text{res} \wedge Fv(A'') \subseteq \overline{x}$
 $\wedge \text{Stb}^+(A) \wedge \text{Stb}^+(A') \wedge M \vdash \text{Enc}(\overline{x : \overline{C'}} \wedge A'')]$
- $\vdash S \wedge S' \triangleq \vdash S \wedge \vdash S'.$

Def 5.6's requirements about free variables are relatively straightforward – more in §E.1.1.

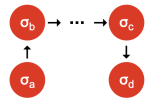
Def 5.6's requirements about encapsulation are motivated by Def. 5.4. If $\overline{x : \overline{C}} \wedge A$ in the scoped invariant were not encapsulated, then it could be invalidated by some external code, and it would be impossible to ever satisfy Def. 5.4(1). Similarly, if a method specification's mid-condition, A'' , could be invalidated by some external code, then it would be impossible to ever satisfy Def. 5.4(2).

Def 5.6's requirements about stability are motivated by our Hoare logic rule for internal calls, [CALL_INT], Fig 8. The requirement $\text{Stb}^+(A)$ for the method's precondition gives that A is preserved when an internal frame is pushed, c.f. Lemma 4.6. The requirement $\text{Stb}^+(A')$ for the method's postcondition gives, in the context of deep satisfaction, that A' is preserved when an internal frame is popped, c.f. Lemma G.42. This is crucial for soundness of [CALL_INT].

5.3 Discussion

Difference with Object and History Invariants. Our scoped invariants are similar to, but different from, history invariants and object invariants. We compare through an example:

Consider σ_a making a call transitioning to σ_b , execution of σ_b 's continuation eventually resulting in σ_c , and σ_c returning to σ_d . Suppose all four states are external, and the module guarantees $\overline{\forall x : \text{Object}. \{A\}}$, and $\sigma_a \not\models A$, but $\sigma_b \models A$. Scoped invariants ensure $\sigma_c \models A$, but allow $\sigma_d \not\models A$.



History invariants [27, 67, 69], instead, consider all future states including any method returns, and therefore would require that $\sigma_d \models A$. Thus, they are, for our purposes, both *unenforceable* and overly

restrictive. Unenforceable: Take $A \triangleq^{\text{txt}} \langle \text{acc} . \text{key} \rangle$, assume in σ_a a path to an external object which has access to $\text{acc} . \text{key}$, assume that path is unknown in σ_b : then, the transition from σ_b to σ_c cannot eliminate that path—hence, $\sigma_d \not\models \langle \text{acc} . \text{key} \rangle$. *Restrictive:* Take $A \triangleq^{\text{txt}} \langle \text{acc} . \text{key} \rangle \wedge a . \text{blnce} \geq b$; then, requiring A to hold in all states from σ_a until termination would prevent all future withdrawals from a , rendering the account useless.

Object invariants [8, 66, 81, 82, 91], on the other hand, expect invariants to hold in all (visible) states, here would require, e.g. that $\sigma_a \models A$. Thus, they are *inapplicable* for us: They would require, e.g., that for all acc , in all (visible) states, $\langle \text{acc} . \text{key} \rangle$, and thus prevent *any* withdrawals from *any* account in *any* state.

Difference between Postconditions and Invariants. In all method specification examples so far, the post-condition and mid-condition were identical. However, this need not be so. Assume a method `tempLeak` defined in `Account`, with an external argument `extArg`, and method body:

```
extArg.m(this.key); this.key:=new Key
```

Then, the assertion $\langle \text{this} . \text{key} \rangle$ is invalidated by the external call `extArg.m(this.key)`, but is established by `this.key:=new Key`. Therefore, $\langle \text{this} . \text{key} \rangle$ is a valid post-condition but not a valid mid-condition. The specification of `tempLeak` could be

```
S_tempLeak   $\triangleq$   { true }
               public Account :: tempLeak(extArg:external)
               {  $\langle \text{this} . \text{key} \rangle$  } || { true }
```

Expressiveness In §E.2 we argue the expressiveness of our approach through a sequence of capability patterns studied in related approaches from the literature [34, 74, 98, 100, 111] and written in our specification language. These approaches are based on temporal logics [74, 98], or on extensions of Coq/Iris [34, 100, 111], and do not offer Hoare logic rules for external calls.

6 HOARE LOGIC

We develop an inference system for adherence to our specifications. We distinguish three phases:

First Phase: We assume an underlying Hoare logic, $M \vdash_{ul} \{A\} \text{stmt} \{A'\}$, and extend it to a logic $M \vdash \{A\} \text{stmt} \{A'\}$ with the expected meaning, i.e. (*) execution of statement *stmt* in a state satisfying *A* will lead to a state satisfying *A'*. These triples only apply to *stmt*'s that do not contain method calls (even internal) – this is so, because method calls may make further calls to external methods. In our extension we introduce judgements which talk about protection.

Second Phase: We develop a logic of quadruples $M \vdash \{A\} \text{stmt} \{A'\} \parallel \{A''\}$. These promise (*) and in addition, that (**) any intermediate external states reachable during execution of that *stmt* satisfy the *mid-condition* *A''*. We incorporate the triples from the first phase, introduce mid-conditions, give the usual substructural rules, and deal with method calls. For internal calls we use the methods' specs. For external calls, we use the module's invariants.

Third Phase: We prove adherence to our specifications. For method specifications we require that the body maps the precondition to the postcondition and preserves the method's mid-condition. For module invariants we require that they are preserved by all public methods of the module.

Preliminaries: The judgement $\vdash M : S$ expresses that *S* is part of *M*'s specification. In particular, it allows *safe renamings*. These renamings are a convenience, akin to the Barendregt convention, and allow simpler Hoare rules – c.f. Sect. 6.3, Def. F.1, and Ex. F.2. We also require an underlying Hoare logic with judgements $M \vdash_{ul} \{A\} \text{stmt} \{A'\}$ – c.f. Ax. F.3.

6.1 First Phase: Triples

In Fig. 7 we introduce our triples, of the form $M \vdash \{A\} \text{stmt} \{A'\}$. These promise, as expected, that any execution of *stmt* in a state satisfying *A* leads to a state satisfying *A'*.

$\frac{\text{[EMBED_UL]} \quad \begin{array}{l} \text{stmt contains no method call} \\ Stbl(A) \ Stbl(A') \quad M \vdash_{ul} \{A\} \ stmt \{A'\} \end{array}}{M \vdash \{A\} \ stmt \{A'\}}$	$\frac{\text{[PROT-NEW]} \quad \begin{array}{l} \text{txt} \\ u \neq x \end{array}}{M \vdash \{true\} \ u = \text{new } C \{ \langle u \rangle \wedge \langle u \rangle \leftarrow x \}}$
$\frac{\text{[PROT-1]} \quad \begin{array}{l} \text{stmt is free of method cals, or assignment to } z \\ M \vdash \{A \wedge e = z\} \ stmt \{e = z\} \end{array}}{M \vdash \{A \wedge \langle e \rangle\} \ stmt \{\langle e \rangle\}}$	$\frac{\text{[PROT-2]} \quad \begin{array}{l} \text{stmt is either } x := y \text{ or } x := y.f \quad \text{txt} \\ z, z' \neq x \\ M \vdash \{A \wedge z = e \wedge z' = e'\} \ stmt \{z = e \wedge z' = e'\} \end{array}}{M \vdash \{A \wedge \langle e \rangle \leftarrow e'\} \ stmt \{\langle e \rangle \leftarrow e'\}}$
$\frac{\text{[PROT-3]} \quad \begin{array}{l} \text{txt} \\ x \neq z \end{array}}{M \vdash \{ \langle y.f \rangle \leftarrow z \} \ x = y.f \{ \langle x \rangle \leftarrow z \}}$	$\frac{\text{[PROT-4]} \quad \begin{array}{l} \text{txt} \\ x \neq z \end{array}}{M \vdash \{ \langle x \rangle \leftarrow z \wedge \langle x \rangle \leftarrow y' \} \ y.f = y' \{ \langle x \rangle \leftarrow z \}}$

Fig. 7. Embedding the Underlying Hoare Logic, and Protection

With rule EMBED_UL in Fig. 7, any triple $\{A\}stmt\{A'\}$ whose statement does not contain a method call, and which can be proven in the underlying Hoare logic, can also be proven in our logic. In PROT-1, we see that protection of an object o is preserved by internal code which does not call any methods: namely any heap modifications will only affect internal objects, and this will not expose new, unmitigated external access to o . PROT-2, PROT-3 and PROT-4 describe the preservation of relative protection. Proofs of soundness for these rules can be found in App. G.5.1.

6.2 Second Phase: Quadruples

6.2.1 Introducing mid-conditions, and substructural rules. We now introduce quadruple rules. Rule MID embeds triples $M \vdash \{A\} s \{A'\}$ into quadruples $M \vdash \{A\} s \{A'\} \parallel \{A''\}$. This is sound, because $stmt$ is guaranteed not to contain method calls (by lemma F.5).

$$\frac{\text{[MID]} \quad M \vdash \{A\} \ stmt \{A'\}}{M \vdash \{A\} \ stmt \{A'\} \parallel \{A''\}}$$

Substructural quadruple rules appear in Fig. 16, and are as expected: Rules SEQU and CONSEQU are the usual rules for statement sequences and consequence, adapted to quadruples. Rule COMBINE combines two quadruples for the same statement into one. Rule ABSURD allows us to deduce anything out of `false` precondition, and CASES allows for case analysis. These rules apply to *any* statements – even those containing method calls.

6.2.2 Adaptation. In the outline of the Hoare proof of the external call in §2.2, we saw that an assertion of the form $\langle x \rangle \leftarrow \bar{y}$ at the call site may imply $\langle x \rangle$ at entry to the call. More generally, the $\neg\forall$ operator adapts an assertion from the view of the callee to that of the caller, and is used in the Hoare logic for method calls. It is defined below.

Definition 6.1. [The $\neg\forall$ operator]

$$\begin{array}{ll} \langle e \rangle \neg\forall \bar{y} \triangleq \langle e \rangle \leftarrow \bar{y} & (A_1 \wedge A_2) \neg\forall \bar{y} \triangleq (A_1 \neg\forall \bar{y}) \wedge (A_2 \neg\forall \bar{y}) \\ (\langle e \rangle \leftarrow \bar{u}) \neg\forall \bar{y} \triangleq \langle e \rangle \leftarrow \bar{u} & (\forall x : C.A) \neg\forall \bar{y} \triangleq \forall x : C.(A \neg\forall \bar{y}) \\ (e : \text{extl}) \neg\forall \bar{y} \triangleq e : \text{extl} & (\neg A) \neg\forall \bar{y} \triangleq \neg(A \neg\forall \bar{y}) \\ e \neg\forall \bar{y} \triangleq e & (e : C) \neg\forall \bar{y} \triangleq e : C \end{array}$$

Only the first equation in Def. 6.1 is interesting: for e to be protected at a callee with arguments \bar{y} , it should be protected from these arguments – thus $\langle e \rangle \neg \bar{y} = \langle e \rangle \leftarrow \bar{y}$. The notation $\langle e \rangle \leftarrow \bar{y}$ stands for $\langle e \rangle \leftarrow y_0 \wedge \dots \wedge \langle e \rangle \leftarrow y_n$, assuming that $\bar{y} = y_0, \dots, y_n$.

Lemma 6.2 states that indeed, \neg adapts assertions from the callee to the caller, and is the counterpart to the ∇ . In particular: (1): \neg turns an assertion into a stable assertion. (2): If the caller, σ , satisfies $A \nabla Rng(\phi)$, then the callee, $\sigma \nabla \phi$, satisfies A . (3): When returning from external states, an assertion implies its adapted version. (4): When calling from external states, an assertion implies its adapted version.

Lemma 6.2. For states σ , assertions A , so that $Stb^+(A)$ and $Fv(A) = \emptyset$, frame ϕ , variables y_0, \bar{y} :

- (1) $Stb(A \neg (y_0, \bar{y}))$
- (2) $M, \sigma \models A \neg Rng(\phi) \implies M, \sigma \nabla \phi \models A$
- (3) $M, \sigma \nabla \phi \models A \wedge \text{extl} \implies M, \sigma \models A \neg Rng(\phi)$
- (4) $M, \sigma \models A \wedge \text{extl} \wedge M \cdot \bar{M} \models \sigma \nabla \phi \implies M, \sigma \nabla \phi \models A \neg Rng(\phi)$

Proofs in Appendix F.5. Example 6.3 demonstrates the need for the `extl` requirement in (3).

Example 6.3 (When returning from internal states, A does not imply $A \neg Rng(\phi)$). In Fig. 2 we have $\sigma_2 = \sigma_1 \nabla \phi_2$, and $\sigma_2 \models \langle o_1 \rangle$, and $o_1 \in Rng(\phi_2)$, but $\sigma_1 \not\models \langle o_1 \rangle \leftarrow o_1$.

6.2.3 Reasoning about calls. is described in Fig. 8. CALL_INT for internal methods, whether public or private; and CALL_EXT_ADAPT and CALL_EXT_ADAPT_STRONG for external methods.

$$\begin{array}{c}
 \text{[CALL_INT]} \\
 \frac{\vdash M : \{A_1\} p C :: m(\overline{x:C}) \{A_2\} \parallel \{A_3\} \quad A'_1 = A_1[y_0, \bar{y}/\text{this}, \bar{x}] \quad A'_2 = A_2[y_0, \bar{y}, u/\text{this}, \bar{x}, \text{res}]}{M \vdash \{y_0 : C, y : \overline{C} \wedge A'_1\} u := y_0.m(y_1, ..y_n) \{A'_2\} \parallel \{A_3\}} \\
 \text{[CALL_EXT_ADAPT]} \\
 \frac{\vdash M : \forall \overline{x:C}. \{A\}}{M \vdash \{y_0 : \text{extl} \wedge \overline{x:C} \wedge A \neg (y_0, \bar{y})\} u := y_0.m(y_1, ..y_n) \{A \neg (y_0, \bar{y})\} \parallel \{A\}} \\
 \text{[CALL_EXT_ADAPT_STRONG]} \\
 \frac{\vdash M : \forall \overline{x:C}. \{A\}}{M \vdash \{y_0 : \text{extl} \wedge \overline{x:C} \wedge A \wedge A \neg (y_0, \bar{y})\} u := y_0.m(y_1, ..y_n) \{A \wedge A \neg (y_0, \bar{y})\} \parallel \{A\}}
 \end{array}$$

Fig. 8. Hoare Quadruples for Internal and External Calls – here \bar{y} stands for y_1, \dots, y_n

For internal calls, we start, as usual, by looking up the method's specification, and substituting the formal by the actual parameters parameters (`this`, \bar{x} by y_0, \bar{y}). CALL_INT is as expected: we require the precondition, and guarantee the postcondition and mid-condition. CALL_INT is applicable whether the method is public or private.

For external calls, we consider the module's invariants. If the module promises to preserve A , i.e. if $\vdash M : \forall \overline{x:D}. \{A\}$, and if its adapted version, $A \neg (y_0, \bar{y})$, holds before the call, then it also holds after the call (CALL_EXT_ADAPT). If, in addition, the un-adapted version also holds before the call, then it also holds after the call (CALL_EXT_ADAPT_STRONG).

Notice that internal calls, CALL_INT, require the *un-adapted* method precondition (i.e. A'_1), while external calls, both CALL_EXT_ADAPT and CALL_EXT_ADAPT_STRONG, require the *adapted* invariant (i.e. $A \neg (y_0, \bar{y})$). This is sound, because internal callees preserve $Stb^+(_)$ -assertions – c.f. Lemma 4.6. On the other hand, external callees do not necessarily preserve $Stb^+(_)$ -assertions – c.f. Ex.

4.7. Therefore, in order to guarantee that A holds upon entry to the callee, we need to know that $A \rightarrow (y_0, \bar{y})$ held at the caller site – *c.f.* Lemma 6.2.

Remember that popping frames does not necessarily preserve $Stb^+(_)$ assertions – *c.f.* Ex. 4.7. Nevertheless, `CALL_INT` guarantees the unadapted version, A , upon return from the call. This is sound, because of our *deep satisfaction* of assertions – more in Sect. 7.

Polymorphic Calls. Our rules do not *directly* address a scenario where the receiver may be internal or external, and where the choice about this is made at runtime. However, such scenarios are *indirectly* supported, through our rules of consequence and case-split. More in Appendix H.6.

Example 6.4 (Proving external calls). We continue our discussion from §2.2 on how to establish the Hoare triple (1) :

$$(1?) \quad \{ \text{buyer} : \text{ext1} \wedge \langle \text{this}.\text{acct}.\text{key} \rangle \leftarrow \text{buyer} \wedge \text{this}.\text{acct}.\text{blnce} = b \} \\ \text{buyer}.\text{pay}(\text{this}.\text{acct}, \text{price}) \\ \{ \text{this}.\text{acct}.\text{blnce} \geq b \} \parallel \{ \langle \text{a}.\text{key} \rangle \wedge \text{a}.\text{blnce} \geq b \}$$

We use S_3 , which says that $\forall a : \text{Account}, b : \text{int}. \{ \langle \text{a}.\text{key} \rangle \wedge \text{a}.\text{blnce} \geq b \}$. We can apply rule `CALL_EXT_ADAPT`, by taking $y_0 \triangleq \text{buyer}$, and $x : \bar{D} \triangleq a : \text{Account}, b : \text{int}$, and $A \triangleq \langle \text{a}.\text{key} \rangle \wedge \text{a}.\text{blnce} \geq b$, and $m \triangleq \text{pay}$, and $\bar{y} \triangleq \text{this}.\text{acct}, \text{price}$, and provided that we can establish that

$$(2?) \quad \langle \text{this}.\text{acct}.\text{key} \rangle \leftarrow (\text{buyer}, \text{this}.\text{acct}, \text{price})$$

holds. Using type information, we obtain that all fields transitively accessible from `this.acct.key`, or `price` are internal or scalar. This implies

$$(3) \quad \langle \text{this}.\text{acct}.\text{key} \rangle \leftarrow \text{this}.\text{acct} \wedge \langle \text{this}.\text{acct}.\text{key} \rangle \leftarrow \text{price}$$

Using then Def. 6.1, we can indeed establish that

$$(4) \quad \langle \text{this}.\text{acct}.\text{key} \rangle \leftarrow (\text{buyer}, \text{this}.\text{acct}, \text{price}) = \langle \text{this}.\text{acct}.\text{key} \rangle \leftarrow \text{buyer}$$

Then, by application of the rule of consequence, (4), and the rule `CALL_EXT_ADAPT`, we can establish (1). More details in §H.3.

6.3 Third phase: Proving adherence to Module Specifications

In Fig. 9 we define the judgment $\vdash M$, which says that M has been proven to be well formed.

$$\begin{array}{c} \text{WELLFRM_MOD} \qquad \text{COMB_SPEC} \\ \frac{\vdash \mathcal{S}pec(M) \quad M \vdash \mathcal{S}pec(M)}{\vdash M} \qquad \frac{M \vdash S_1 \quad M \vdash S_2}{M \vdash S_1 \wedge S_2} \\ \text{METHOD} \\ \frac{\text{mBody}(m, D, M) = p(\overline{y : D})\{ stmt \} \quad M \vdash \{ \text{this} : D, \overline{y : D} \wedge A_1 \} stmt \{ A_2 \wedge A_2 \rightarrow res \} \parallel \{ A_3 \}}{M \vdash \{ A_1 \} p D :: m(\overline{y : D}) \{ A_2 \} \parallel \{ A_3 \}} \\ \text{INVARIANT} \\ \frac{\forall D, m : \text{mBody}(m, D, M) = \text{public}(\overline{y : D})\{ stmt \} \implies M \vdash \{ \text{this} : D, \overline{y : D}, x : \bar{C} \wedge A \wedge A \rightarrow (\text{this}, \bar{y}) \} stmt \{ A \wedge A \rightarrow res \} \parallel \{ A \}}{M \vdash \forall x : \bar{C}. \{ A \}} \end{array}$$

Fig. 9. Methods' and Modules' Adherence to Specification

`WELLFRM_MOD` and `COMB_SPEC` say that M is well-formed if its specification is well-formed (according to Def. 5.6), and if M satisfies all conjuncts of the specification. `METHOD` says that a module satisfies a method specification if the body satisfies the corresponding pre-, post- and

midcondition. In the postcondition we also ask that $A \rightarrow \text{res}$, so that res does not leak any of the values that A promises will be protected. INVARIANT says that a module satisfies a specification $\forall \bar{x} : \overline{C}. \{A\}$, if the method body of each public method has A as its pre-, post- and midcondition. Moreover, the precondition is strengthened by $A \rightarrow (\text{this}, \bar{y})$ – this is sound because the caller is external, and by Lemma 6.2, part (4).

Barendregt In METHOD we implicitly require the free variables in a method’s precondition not to overlap with variables in its body, unless they are the receiver or one of the parameters ($Vs(stmt) \cap Fv(A_1) \subseteq \{\text{this}, y_1, \dots, y_n\}$). And in INVARIANT we require the free variables in A (which are a subset of \bar{x}) not to overlap with the variable in $stmt$ ($Vs(stmt) \cap \bar{x} = \emptyset$). This can easily be achieved through renamings, c.f. Def. F.1.

Example 6.5 (Proving a public method). Consider the proof that $\text{Account} :: \text{set}$ from M_{fine} satisfies S_2 . Applying rule INVARIANT, we need to establish:

$$(5?) \quad \{ \dots a : \text{Account} \wedge \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \Leftarrow (key', key'') \} \\ \text{body_of_set_in_Account_in_}M_{fine}$$

$$\{ \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \rightarrow \text{res} \} \parallel \{ \langle a.\text{key} \rangle \}$$

Given the conditional statement in set , and with the obvious treatment of conditionals (c.f. Fig. 16), among other things, we need to prove for the true -branch that:

$$(6?) \quad \{ \dots \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \Leftarrow (key', key'') \wedge \text{this}.\text{key} = key' \} \\ \text{this}.\text{key} := key'' \\ \{ \langle a.\text{key} \rangle \} \parallel \{ \langle a.\text{key} \rangle \}$$

We can apply case-split (c.f. Fig. 16) on whether $\text{this}=a$, and thus a proof of (7?) and (8?), would give us a proof of (6?):

$$(7?) \quad \{ \dots \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \Leftarrow (key', key'') \wedge \text{this}.\text{key} = key' \wedge \text{this} = a \} \\ \text{this}.\text{key} := key'' \\ \{ \langle a.\text{key} \rangle \} \parallel \{ \langle a.\text{key} \rangle \}$$

and also

$$(8?) \quad \{ \dots \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \Leftarrow (key', key'') \wedge \text{this}.\text{key} = key' \wedge \text{this} \neq a \} \\ \text{this}.\text{key} := key'' \\ \{ \langle a.\text{key} \rangle \} \parallel \{ \langle a.\text{key} \rangle \}$$

If $\text{this}.\text{key} = key' \wedge \text{this} = a$, then $a.\text{key} = key'$. But $\langle a.\text{key} \rangle \Leftarrow key'$ and PROT-NEQ from Fig. 17 give $a.\text{key} \neq key'$. So, by contradiction (c.f. Fig. 16), we can prove (7?). If $\text{this} \neq a$, then we obtain from the underlying Hoare logic that the value of $a.\text{key}$ did not change. Thus, by rule PROT_1, we obtain (8?). More details in §H.5.

On the other hand, set from M_{bad} cannot be proven to satisfy S_2 , because it requires proving

$$(??) \quad \{ \dots \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \Leftarrow (key', key'') \} \\ \text{this}.\text{key} := key'' \\ \{ \{ \langle a.\text{key} \rangle \} \parallel \{ \langle a.\text{key} \rangle \} \}$$

and without the condition $\text{this}.\text{key} = key'$ there is no way we can prove (??).

6.4 Our Example Proven

Using our Hoare logic, we have developed a mechanised proof in Coq, that, indeed, $M_{good} \vdash S_2 \wedge S_3$. This proof is part of the current submission (in a *.zip file), and will be submitted as an artifact with the final version.

Our proof models \mathcal{L}_{ul} , the assertion language, the specification language, and the Hoare logic from §6.1, §6.2, §6.3, §F and Def. 6.1. In keeping with the start of §6, our proof assumes the existence of an underlying Hoare logic, and several, standard, properties of that underlying logic, the assertions logic (e.g. equality of objects implies equality of field accesses) and of type systems

(e.g. fields of objects of different types cannot be aliases of one another). All assumptions are clearly indicated in the associated artifact.

In appendix H, included in the auxiliary material, we outline the main ingredients of that proof.

7 SOUNDNESS

We now give a synopsis of the proof of soundness of the logic from §6, and outline the two most interesting aspects: deep satisfaction, and summarized execution.

Deep Satisfaction We are faced with the problem that assertions are not always preserved when popping the top frame (c.f. Ex. 4.7), while we need to be able to argue that method return preserves post-conditions. For this, we introduce a “deeper” notion of assertion satisfaction, which requires that an assertion not only is satisfied from the viewpoint of the top frame, but also from the viewpoint of all frames from k -th frame onwards: $M, \sigma, k \models A$ says that $\forall j. [k \leq j \leq |\sigma| \Rightarrow M, \sigma[j] \models A]$. Accordingly, we introduce *deep specification satisfaction*, $\bar{M}; M \models_{deep} \{A\} \sigma \{A'\} \parallel \{A''\}$, which promises for all $k \leq |\sigma|$, if $M, \sigma, k \models A$, and if scoped execution of σ 's continuation leads to final state σ' and intermediate external state σ'' , then $M, \sigma', k \models A'$, and $M, \sigma'', k \models A''$ - c.f. App. G.3.

Here how deep satisfaction addresses this problem: Assume state σ_1 right before entering a call, σ_2 and σ_3 at start and end of the call's body, and σ_4 upon return. If a pre-condition holds at σ_1 , then it holds for a $k \leq |\sigma_1|$; hence, if the postcondition holds for k at σ_3 , and because $|\sigma_3| = |\sigma_1| + 1$, it also holds for σ_4 . Deep satisfaction is stronger than shallow (i.e. specification satisfaction as in Def. 5.2).

Lemma 7.1. For all $\bar{M}, M, A, A', A'', \sigma$:

$$\bullet \bar{M}; M \models_{deep} \{A\} \sigma \{A'\} \parallel \{A''\} \implies \bar{M}; M \models \{A\} \sigma \{A'\} \parallel \{A''\}$$

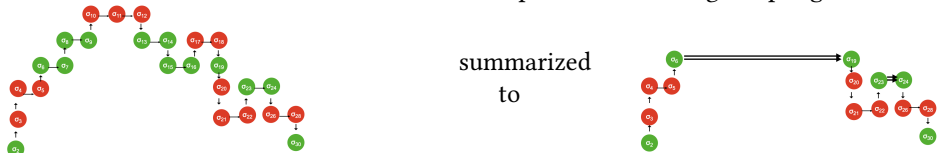
Soundness of the Triples Logic We require the assertion logic, $M \vdash A$, and the underlying Hoare logic, $M \vdash_{ul} \{A\} stmt \{A'\}$, to be sound. Such sound logics do exist. Namely, one can build an assertion logic, $M \vdash A$, by extending a logic which does not talk about protection, through the addition of structural rules which talk about protection; this extension preserves soundness - c.f. App. G.1. Moreover, since the assertions A and A' in $M \vdash_{ul} \{A\} stmt \{A'\}$ may, but need not, talk about protection, one can take a Hoare logic from the literature as the \vdash_{ul} -logic.

We then prove soundness of the rules about protection from Fig. 7, and, based on this, we prove soundness of the inference system for triples - c.f. Appendix G.5.

Theorem 7.2. For module M such that $\vdash M$, and for any assertions A, A', A'' and statement $stmt$:

$$M \vdash \{A\} stmt \{A'\} \implies M \models_{deep} \{A\} stmt \{A'\} \parallel \{A''\}$$

Summarised Execution. Execution of an external call may consist of any number of external transitions, interleaved with calls to public internal methods, which in turn may make any number of further internal calls (public or private), and these, again may call external methods. For the proof of soundness, internal and external transitions use different arguments. For external transitions we consider small steps and argue in terms of preservation of encapsulated properties, while for internal calls, we use large steps, and appeal to the method's specification. Therefore, we define *sumarized executions*, where internal calls are collapsed into one, large step, e.g. below:



Lemma G.28 says that any terminating execution starting in an external state consists of a sequence of external states interleaved with terminating executions of public methods. Lemma

G.29 says that such an execution preserves an encapsulated assertion A provided that all these finalising internal executions also preserve A .

Soundness of the Quadruples Logic Proving soundness of our quadruples requires induction on the execution in some cases, and induction on the derivation of the quadruples in others. We address this through a well-founded ordering that combines both, c.f. Def. G.22 and lemma G.23. Finally, in G.16, we prove soundness:

THEOREM 7.3. *For module M , assertions A, A', A'' , state σ , and specification S :*

$$\begin{aligned} (A) : \vdash M \wedge M \vdash \{A\} \text{ stmt}\{A'\} \parallel \{A''\} &\implies M \models_{\text{deep}} \{A\} \text{ stmt}\{A'\} \parallel \{A''\} \\ (B) : M \vdash S &\implies M \models_{\text{deep}} S \end{aligned}$$

8 CONCLUSION: SUMMARY, RELATED WORK AND FURTHER WORK

Our motivation comes from the OCAP approach to security, whereby object capabilities guard against un-sanctioned effects. Miller [83, 85] advocates *defensive consistency*: whereby “An object is defensively consistent when it can defend its own invariants and provide correct service to its well behaved clients, despite arbitrary or malicious misbehaviour by its other clients.” Defensively consistent modules are hard to design and verify, but make it much easier to make guarantees about systems composed of multiple components [92].

Our Work aims to elucidate such guarantees. We want to formalize and prove that [44]:

Lack of eventual access implies that certain properties will be preserved, even in the presence of external calls.

For this, we had to model the concept of lack of eventual access, determine the temporal scope of the preservation, and develop a Hoare logic framework to formally prove such guarantees.

For lack of eventual access, we introduced protection, a property of all the paths of all external objects accessible from the current stack frame. For the temporal scope of preservation, we developed scoped invariants, which ensure that a given property holds as long as we have not returned from the current method. (top of current stack has not been popped yet). For our Hoare logic, we introduced an adaptation operator, which translates assertions between the caller’s and callee’s frames. Finally, to prove the soundness of our approach, we developed the notion of deep satisfaction, which mandates that an assertion must be satisfied from a particular stack frame onward. Thus, most concepts in this work are *scope-aware*, as they depend on the current stack frame.

With these concepts, we developed a specification language for modules limiting effects, a Hoare Logic for proving external calls, protection, and adherence to specifications, and have proven it sound.

Lack of Eventual Access Efforts to restrict “eventual access” have been extensively explored, with Ownership Types being a prominent example [20, 26]. These types enforce encapsulation boundaries to safeguard internal implementations, thereby ensuring representation independence and defensive consistency [6, 25, 94]. Ownership is fundamental to key systems like Rust’s memory safety [60, 62], Scala’s Concurrency [50, 51], Java heap analyses [54, 88, 99], and plays a critical role in program verification [13, 65] including Spec# [8, 9] and universes [36, 37, 72], Borrowable Fractional Ownership [93], and recently integrated into languages like OCAML [71, 76].

Ownership types are closely related to the notion of protection: both are scoped relative to a frame. However, ownership requires an object to control some part of the path, while protection demands that module objects control the endpoints of paths.

In future work we want to explore how to express protection within Ownership Types, with the primary challenge being how to accommodate capabilities accessible to some external objects while still inaccessible to others. Moreover, tightening some rules in our current Hoare logic (e.g.

Def. 4.4) may lead to a native Hoare logic of ownership. Also, recent approaches like the Alias Calculus [63, 104], Reachability Types [7?] and Capturing Types [12, 17, 118] abstract fine-grained method-level descriptions of references and aliases flowing into and out of methods and fields, and likely accumulate enough information to express protection. Effect exclusion [73] directly prohibits nominated effects, but within a closed, fully-typed world.

Temporal scope of the guarantee Starting with loop invariants[47, 55], property preservation at various granularities and durations has been widely and successfully adapted and adopted [8, 27, 41, 56, 66, 67, 69, 81, 82, 91]. In our work, the temporal scope of the preservation guarantee includes all nested calls, until termination of the currently executing method, but not beyond. We compare with object and history invariants in §3.3.2.

Such guarantees are maintained by the module as a whole. Drossopoulou et al. [43] proposed “holistic specifications” which take an external perspective across the interface of a module. Mackay et al. [74] builds upon this work, offering a specification language based on *necessary* conditions and temporal operators. Neither of these systems support any kind of external calls. Like [43, 74] we propose “holistic specifications”, albeit without temporal logics, and with sufficient conditions. In addition, we introduce protection, and develop a Hoare logic for protection and external calls.

Hoare Logics were first developed in Hoare’s seminal 1969 paper [55], and have inspired a plethora of influential further developments and tools. We shall discuss a few only.

Separation logics [58, 102] reason about disjoint memory regions. Incorporating Separation Logic’s powerful framing mechanisms will pose several challenges: We have no specifications and no footprint for external calls. Because protection is “scope-aware”, expressing it as a predicate would require quantification over all possible paths and variables within the current stack frame. We may also require a new separating conjunction operator. Hyper-Hoare Logics [30, 40] reason about the execution of several programs, and could thus be applied to our problem, if extended to model all possible sequences of calls of internal public methods.

Incorrectness Logic [95] under-approximates postconditions, and thus reasons about the presence of bugs, rather than their absence. Our work, like classical Hoare Logic, over-approximates postconditions, and differs from Hoare and Incorrectness Logics by tolerating interactions between verified code and unverified components. Interestingly, even though earlier work in the space [43, 74] employ *necessary* conditions for effects (*i.e.* under-approximate pre-conditions), we can, instead, employ *sufficient* conditions for the lack of effects (over-approximate postconditions). Incorporating our work into Incorrectness Logic might require under-approximating eventual access, while protection over-approximates it.

Rely-Guarantee [52, 114] and Deny-Guarantee [39] distinguish between assertions guaranteed by a thread, and those a thread can reply upon. Our Hoare quadruples are (roughly) Hoare triples plus the “guarantee” portion of rely-guarantee. When a specification includes a guarantee, that guarantee must be maintained by every “atomic step” in an execution [52], rather than just at method boundaries as in visible states semantics [41, 91, 109]. In concurrent reasoning, this is because shared state may be accessed by another cooperating thread at any time: while in our case, it is because unprotected state may be accessed by an untrusted component within the same thread.

Models and Hoare Logics for the interaction with the the external world Murray [92] made the first attempt to formalise defensive consistency, to tolerate interacting with any untrustworthy object, although without a specification language for describing effects (*i.e.* when an object is correct).

Cassez et al. [21] propose one approach to reason about external calls. Given that external callbacks are necessarily restricted to the module’s public interface, external callsites are replaced with a generated `externalcall()` method that nondeterministically invokes any method in that interface. Rao et al. [101]’s Iris-Wasm is similar. WASM’s modules are very loosely coupled: a

module has its own byte memory and object table. Iris-Wasm ensures models can only be modified via their explicitly exported interfaces.

Swasey et al. [111] designed OCPL, a logic that separates internal implementations (“high values”) from interface objects (“low values”). OCPL supports defensive consistency (called “robust safety” after the security literature [10]) by ensuring low values can never leak high values, and prove object-capability patterns, such as sealer/unsealer, caretaker, and membrane. RustBelt [60] developed this approach to prove Rust memory safety using Iris [61], and combined with RustHorn [78] for the safe subset, produced RustHornBelt [77] that verifies both safe and unsafe Rust programs. Similar techniques were extended to C [105]. While these projects verify “safe” and “unsafe” code, the distinction is about memory safety: whereas all our code is “memory safe” but unsafe / untrusted code is unknown to the verifier.

Devriese et al. [34] deploy step-indexing, Kripke worlds, and representing objects as public/private state machines to model problems including the DOM wrapper and a mashup application. Their distinction between public and private transitions is similar to our distinction between internal and external objects. This stream of work has culminated in VMSL, an Iris-based separation logic for virtual machines to assure defensive consistency [70] and Cerise, which uses Iris invariants to support proofs of programs with outgoing calls and callbacks, on capability-safe CPUs [48], via problem-specific proofs in Iris’s logic. Our work differs from Swasey, Schaefer’s, and Devriese’s work in that they are primarily concerned with ensuring defensive consistency, while we focus on module specifications.

Smart Contracts also pose the problem of external calls. Rich-Ethereum [18] relies on Ethereum contracts’ fields being instance-private and unaliased. Scilla [107] is a minimalistic functional alternative to Ethereum, which has demonstrated that popular Ethereum contracts avoid common contract errors when using Scilla.

The VerX tool can verify specifications for Solidity contracts automatically [98]. VerX’s specification language is based on temporal logic. It is restricted to “effectively call-back free” programs [2, 49], delaying any callbacks until the incoming call to the internal object has finished.

CONSOL [115] provides a specification language for smart contracts, checked at runtime [46]. SCIO* [4], implemented in F*, supports both verified and unverified code. Both CONSOL and SCIO* are similar to gradual verification techniques [28, 119] that insert dynamic checks between verified and unverified code, and contracts for general access control [29, 38, 89].

Programming languages with object capabilities Google’s Caja [87] applies (object-)capabilities [33, 83, 90], sandboxes, proxies, and wrappers to limit components’ access to *ambient* authority. Sandboxing has been validated formally [75]; Many recent languages [19, 53, 103] including Newspeak [16], Dart [15], Grace [11, 59] and Wyvern [79] have adopted object capabilities. Schaefer et al. [106] has also adopted an information-flow approach to ensure confidentiality by construction.

Anderson et al. [3] extend memory safety arguments to “stack safety”: ensuring method calls and returns are well bracketed (aka “structured”), and that the integrity and confidentiality of both caller and callee are ensured, by assigning objects to security classes. Schaefer et al. [106] has also adopted an information-flow approach to ensure confidentiality by construction.

Future work. We will look at the application of our techniques to languages that rely on lexical nesting for access control such as Javascript [84], rather than public/private annotations, languages that support ownership types such as Rust, leveraged for verification [5, 64, 77], and languages from the functional tradition such as OCAML, with features such as ownership and uniqueness [71, 76]. These different language paradigms may lead us to refine our ideas for eventual access, footprints and framing operators. We want to incorporate our techniques into existing program verification tools [28], especially those attempting gradual verification [119].

DATA AVAILABILITY STATEMENT

An extended version of the paper including extensive appendices of full definitions and manual proofs have been uploaded as anonymised auxiliary information with this submission.

The Coq source will be submitted as an artefact to the artefact evaluation process. The code artefact, along with the extended appendices etc will be made permanently available in the ACM Digital Library archive.

REFERENCES

- [1] Gul Agha and Carl Hewitt. 1987. Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming. In *Research Directions in Object-Oriented Programming*, Bruce D. Shriver and Peter Wegner (Eds.). MIT Press, 49–74.
- [2] Elvira Albert, Shelly Grossman, Noam Rinetzky, Clara Rodríguez-Núñez, Albert Rubio, and Mooly Sagiv. 2023. Relaxed Effective Callback Freedom: A Parametric Correctness Condition for Sequential Modules With Callbacks. *IEEE Trans. Dependable Secur. Comput.* 20, 3 (2023), 2256–2273. <https://doi.org/10.1109/TDSC.2022.3178836>
- [3] Sean Noble Anderson, Roberto Blanco, Leonidas Lampropoulos, Benjamin C. Pierce, and Andrew Tolmach. 2023. Formalizing Stack Safety as a Security Property. In *Computer Security Foundations Symposium*. 356–371. <https://doi.org/10.1109/CSF57540.2023.00037>
- [4] Cezar-Constantin Andrici, Ștefan Ciobăcă, Catalin Hritcu, Guido Martínez, Exequiel Rivas, Éric Tanter, and Théo Winterhalter. 2024. Securing Verified IO Programs Against Unverified Code in F. *POPL* 8 (2024), 2226–2259. <https://doi.org/10.1145/3632916>
- [5] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust types for modular specification and verification. *OOPSLA* 3 (2019), 147:1–147:30. <https://doi.org/10.1145/3360573>
- [6] Anindya Banerjee and David A. Naumann. 2005. Ownership Confinement Ensures Representation Independence for Object-oriented Programs. *J. ACM* 52, 6 (Nov. 2005), 894–960. <https://doi.org/10.1145/1101821.1101824>
- [7] Yuyan Bao, Guannan Wei, Oliver Bracevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. 2021. Reachability types: tracking aliasing and separation in higher-order functional programs. *OOPSLA* 5 (2021), 1–32. <https://doi.org/10.1145/3485516>
- [8] Michael Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. 2004. Verification of Object-Oriented Programs with Invariants. *J. Object Technol.* 3, 6 (2004), 27–56. <https://doi.org/10.5381/JOT.2004.3.6.A2>
- [9] Mike Barnett, Rustan Leino, and Wolfram Schulte. 2005. The Spec# Programming System: An Overview. In *CASSIS*, Vol. LNCS3362. 49–69. https://doi.org/10.1007/978-3-540-30569-9_3
- [10] Jesper Bengtson, Kathiekeyan Bhargavan, Cedric Fournet, Andrew Gordon, and S.Maffeis. 2011. Refinement Types for Secure Implementations. *TOPLAS* (2011), 1–45. <https://doi.org/10.1145/1890028.1890031>
- [11] Andrew Black, Kim Bruce, Michael Homer, and James Noble. 2012. Grace: the Absence of (Inessential) Difficulty. In *Onwards*. 85–98. <https://doi.org/10.1145/2384592.2384601>
- [12] Aleksander Boruch-Gruszecki, Martin Odersky, Edward Lee, Ondrej Lhoták, and Jonathan Immanuel Brachthäuser. 2023. Capturing Types. *TOPLAS* 45, 4 (2023), 21:1–21:52. <https://doi.org/10.1145/3618003>
- [13] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. 2003. Ownership types for object encapsulation. In *POPL*. 213–223. <https://doi.org/10.1145/604131.604156>
- [14] John Boyland. 2001. Alias burying: Unique variables without destructive reads. In *S:P&E*. 533–553. <https://doi.org/10.1002/spe.370>
- [15] Gilad Bracha. 2015. *The Dart Programming Language*. Addison-Wesley. 224 pages. <https://dart.dev>
- [16] Gilad Bracha. 2017. The Newspeak Language Specification Version 0.1. (Feb. 2017). <https://newspeaklanguage.org/>
- [17] Jonathan Immanuel Brachthäuser, Philipp Schuster, Edward Lee, and Aleksander Boruch-Gruszecki. 2022. Effects, capabilities, and boxes: from scope-based reasoning to type-based reasoning and back. *OOPSLA* 6 (2022), 1–30. <https://doi.org/10.1145/3527320>
- [18] Christian Bräm, Marco Eilers, Peter Müller, Robin Sierra, and Alexander J. Summers. 2021. Rich specifications for Ethereum smart contract verification. *OOPSLA* 5 (2021), 1–30. <https://doi.org/10.1145/3485523>
- [19] Anton Burtsev, David Johnson, Josh Kunz, Eric Eide, and Jacobus E. van der Merwe. 2017. CapNet: security and least authority in a capability-enabled cloud. In *SoCC*. 128–141. <https://doi.org/10.1145/3127479.3131209>
- [20] Nicholas Cameron, Sophia Drossopoulou, and James Noble. 2013. Understanding Ownership Types with Dependent Types. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. 84–108. https://doi.org/10.1007/978-3-642-36946-9_5
- [21] Franck Cassez, Joanne Fuller, and Horacio Mijail Anton Quiles. 2024. Deductive verification of smart contracts with Dafny. *Int. J. Softw. Tools Technol. Transf.* 26, 2 (2024), 131–145. <https://doi.org/10.1007/S10009-024-00738-1>

- [22] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. 2005. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In *FMCO*. 342–363. https://doi.org/10.1007/11804192_16
- [23] Edwin C. Chan, John Boyland, and William L. Scherlis. 1998. Promises: Limited Specifications for Analysis and Manipulation. In *ICSE*. 167–176. <https://doi.org/10.1109/ICSE.1998.671113>
- [24] Christoph Jentsch. 2016. Decentralized Autonomous Organization to automate governance. (March 2016). <https://download.slock.it/public/DAO/WhitePaper.pdf>
- [25] David G. Clarke, John M. Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *OOPSLA*. 48– 64. <https://doi.org/10.1145/286936.286947>
- [26] David G. Clarke, John M. Potter, and James Noble. 2001. Simple Ownership Types for Object Containment. In *ECOOP*. 53–76. https://doi.org/10.1007/3-540-45337-7_4
- [27] Ernie Cohen, Michal Moskal, Wolfram Schulte, and Stephan Tobies. 2010. Local Verification of Global Invariants in Concurrent Programs. In *CAV*. 480–494. https://doi.org/10.1007/978-3-642-14295-6_42
- [28] David R. Cok and K. Rustan M. Leino. 2022. *Specifying the Boundary Between Unverified and Verified Code*. Chapter 6, 105–128. https://doi.org/10.1007/978-3-031-08166-8_6
- [29] Joseph W. Cutler, Craig Disselkoe, Aaron Eline, Shaobo He, Kyle Headley, Michael Hicks, Kesha Hietala, Eleftherios Ioannidis, John H. Kastner, Anwar Mamat, Darin McAdams, Matt McCutchen, Neha Rungta, Emina Torlak, and Andrew Wells. 2024. Cedar: A New Language for Expressive, Fast, Safe, and Analyzable Authorization. 8, *OOPSLA1* (2024), 670–697. <https://doi.org/10.1145/3649835>
- [30] Thibault Dardinier and Peter Müller. 2024. Hyper Hoare Logic: (Dis-)Proving Program Hyperproperties. In *PLDI*, Vol. 8. 1485–1509. <https://doi.org/10.1145/3656437>
- [31] Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, James Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, and Jonathan Woodruff. 2019. CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment. In *ASPLOS*. 379–393. <https://doi.org/10.1145/3297858.3304042>
- [32] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D’Hondt, and W. De Meuter. 2006. Ambient-Oriented Programming in AmbientTalk. In *ECOOP*. 230–254. https://doi.org/10.1007/11785477_16
- [33] Jack B. Dennis and Earl C. Van Horn. 1966. Programming Semantics for Multiprogrammed Computations. *Comm. ACM* 9, 3 (1966), 143–155. <https://doi.org/10.1145/365230.365252>
- [34] Dominique Devriese, Lars Birkedal, and Frank Piessens. 2016. Reasoning about Object Capabilities with Logical Relations and Effect Parametricity. In *IEEE EuroS&P*. 147–162. <https://doi.org/10.1109/EuroSP.2016.22>
- [35] W. Dietl, S. Drossopoulou, and P. Müller. 2007. Generic Universe Types. In *ECOOP (LNCS, Vol. 4609)*. Springer, 28–53. <http://www.springerlink.com>
- [36] Werner Dietl, Sophia Drossopoulou, and Peter Müller. 2007. Generic Universe Types. In *ECOOP*, Vol. 4609. 28–53. https://doi.org/10.1007/978-3-540-73589-2_3
- [37] W. Dietl and P. Müller. 2005. Universes: Lightweight Ownership for JML. *JOT* 4, 8 (October 2005), 5–32. <https://doi.org/10.5381/jot.2005.4.8.a1>
- [38] Christos Dimoulas, Scott Moore, Aslan Askarov, and Stephen Chong. 2014. Declarative Policies for Capability Control. In *Computer Security Foundations Symposium (CSF)*. 3–17. <https://doi.org/10.1109/CSF.2014.9>
- [39] Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. 2009. Deny-guarantee reasoning. In *ESOP*. 363–377. https://doi.org/10.1007/978-3-642-00590-9_26
- [40] Emanuele D’Ousualdo, Azadeh Farzan, and Derek Dreyer. 2022. Proving hypersafety compositionally. *Proc. ACM Program. Lang.* 6, *OOPSLA2* (2022), 289–314. <https://doi.org/10.1145/3563298>
- [41] S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers. 2008. A Unified Framework for Verification Techniques for Object Invariants. In *ECOOP*. 412–437. https://doi.org/10.1007/978-3-540-70592-5_18
- [42] Sophia Drossopoulou and James Noble. 2013. The need for capability policies. In *FTfJP*. 61–67. <https://doi.org/10.1145/2489804.2489811>
- [43] Sophia Drossopoulou, James Noble, Julian Mackay, and Susan Eisenbach. 2020. Holistic Specifications for Robust Programs. In *FASE*. 420–440. https://doi.org/10.1007/978-3-030-45234-6_21
- [44] Sophia Drossopoulou, James Noble, Mark Miller, and Toby Murray. 2016. Permission and Authority revisited – towards a formalization. In *(FTfJP)*. 1 – 6. <http://dl.acm.org/citation.cfm?id=2955821>
- [45] J. C. Filliatre, L. Gondelman, and A. Pakevich. 2016. The spirit of ghost code. In *Formal Methods System Design*. 1–16. https://doi.org/10.1007/978-3-319-08867-9_1
- [46] Robert Bruce Findler and Matthias Felleisen. 2001. Contract Soundness for object-oriented languages. In *OOPSLA*. 1–15. <https://doi.org/10.1145/504282.504283>
- [47] Robert W. Floyd. 1967. Assigning Meanings to Programs. *Mathematical Aspects of Computer Science* 19 (1967), 19–32. <https://people.eecs.berkeley.edu/~necula/Papers/FloydMeaning.pdf>

- [48] Aina Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, and Lars Birkedal. 2024. Cerise: Program Verification on a Capability Machine in the Presence of Untrusted Code. *J. ACM* 71, 1 (2024), 3:1–3:59. <https://doi.org/10.1145/3623510>
- [49] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzy, Mooly Sagiv, and Yoni Zohar. 2018. Online Detection of Effectively Callback Free Objects with Applications to Smart Contracts. *POPL* (2018). <https://doi.org/10.1145/3158136>
- [50] Philipp Haller. 2024. Lightweight Affine Types for Safe Concurrency in Scala (Keynote). In *Programming*. <https://doi.org/10.1145/3660829.3661033> <https://speakerdeck.com/phaller/towards-safer-lightweight-concurrency-in-scala>.
- [51] Philipp Haller and Alexander Loiko. 2016. LaCasa: lightweight affinity and object capabilities in Scala. In *OOPSLA*. 272–291. <https://doi.org/10.1145/2983990.2984042>
- [52] Ian J. Hayes and Cliff B. Jones. 2018. A Guide to Rely/Guarantee Thinking. In *SETSS 2017*. 1–38. https://doi.org/10.1007/978-3-030-02928-9_1
- [53] Ian J. Hayes, Xi Wu, and Larissa A. Meinicke. 2017. Capabilities for Java: Secure Access to Resources. In *APLAS*. 67–84. https://doi.org/10.1007/978-3-319-71237-6_4
- [54] Trent Hill, James Noble, and John Potter. 2002. Scalable Visualizations of Object-Oriented Systems with Ownership Trees. *J. Vis. Lang. Comput.* 13, 3 (2002), 319–339. <https://doi.org/10.1006/jvlc.2002.0238>
- [55] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Comm. ACM* 12 (1969), 576–580. <https://doi.org/10.1145/363235.363259>
- [56] C. A. R. Hoare. 1974. Monitors: an operating system structuring concept. *Commun. ACM* 17, 10 (1974), 549–557. <https://doi.org/10.1145/355620.361161>
- [57] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *ACM ToPLAS* 23, 3 (2001), 396–450. <https://doi.org/10.1145/503502.503505>
- [58] S. S. Ishtiaq and P. W. O’Hearn. 2001. BI as an assertion language for mutable data structures. In *POPL*. 14–26. <https://doi.org/10.1145/360204.375719>
- [59] Timothy Jones, Michael Homer, James Noble, and Kim B. Bruce. 2016. Object Inheritance Without Classes. In *ECOOP*. 13:1–13:26. <https://doi.org/10.4230/LIPIcs.ECOOP.2016.13>
- [60] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. 2, *POPL* (2017), 66:1–66:34. <https://doi.org/10.1145/3158154>
- [61] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. <https://doi.org/10.1017/S0956796818000151>
- [62] Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language* (2nd ed.). No Starch Press. <https://doc.rust-lang.org/book/>
- [63] Alexander Kogtenkov, Bertrand Meyer, and Sergey Velder. 2015. Alias calculus, change calculus and frame inference. *Sci. Comp. Prog.* 97 (2015), 163–172. <https://doi.org/10.1016/j.scico.2013.11.006>
- [64] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. In *OOPSLA*, Vol. 7. 286–315. <https://doi.org/10.1145/3586037>
- [65] Dirk Leinenbach and Thomas Santen. 2009. Verifying the Microsoft Hyper-V Hypervisor with VCC. In *Formal Methods*. 806–809. https://doi.org/10.1007/978-3-642-05089-3_51
- [66] K. R. M. Leino and P. Müller. 2004. Object Invariants in Dynamic Contexts. In *ECOOP*. 491–516. https://doi.org/10.1007/978-3-540-24851-4_22
- [67] K. Rustan M. Leino and Wolfram Schulte. 2007. Using History Invariants to Verify Observers. In *ESOP*. 80–94. https://doi.org/10.1007/978-3-540-71316-6_7
- [68] Henry M. Levy. 1984. *Capability-Based Computer Systems*. Butterworth-Heinemann. <https://homes.cs.washington.edu/~levy/capabook/>
- [69] Barbara Liskov and Jeanette Wing. 1994. A Behavioral Notion of Subtyping. *ACM ToPLAS* 16, 6 (1994), 1811–1841. <https://www.cs.cmu.edu/~wing/publications/LiskovWing94.pdf>
- [70] Zongyuan Liu, Sergei Stepanenko, Jean Pichon-Pharabod, Amin Timany, Aslan Askarov, and Lars Birkedal. 2023. VMSL: A Separation Logic for Mechanised Robust Safety of Virtual Machines Communicating above FF-A. 7, *PLDI* (2023), 1438–1462. <https://doi.org/10.1145/3591279>
- [71] Anton Lorenzen, Stephen Dolan, Richard A. Eisenberg, and Sam Lindley. 2024. Oxidizing OCaml with Modal Memory Management. In *ICFP*. 448–475. <https://doi.org/10.1145/3674642>
- [72] Y. Lu and J. Potter. 2006. Protecting Representation with Effect Encapsulation.. In *POPL*. 359–371. <https://dl.acm.org/doi/10.1145/1111320.1111069>
- [73] Matthew Lutze, Magnus Madsen, Philipp Schuster, and Jonathan Immanuel Brachthäuser. 2023. With or Without You: Programming with Effect Exclusion. *ICFP* (2023), 448–475. <https://doi.org/10.1145/3607846>

- [74] Julian Mackay, Susan Eisenbach, James Noble, and Sophia Drossopoulou. 2022. *Necessity Specifications for Robustness*. *Proc. ACM Program. Lang.* 6, OOPSLA2, 811–840. <https://doi.org/10.1145/3563317>
- [75] S. Maffei, J.C. Mitchell, and A. Taly. 2010. Object Capabilities and Isolation of Untrusted Web Applications. In *Proc of IEEE Security and Privacy*. 125–140. <https://ieeexplore.ieee.org/document/5504710>
- [76] Daniel Marshall and Dominic Orchard. 2024. Functional Ownership through Fractional Uniqueness. In *OOPSLA*. 1040–1070. <https://doi.org/10.1145/3649848>
- [77] Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. 2022. RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code. In *PLDI*. ACM, 841–856. <https://dl.acm.org/doi/10.1145/3519939.3523704>
- [78] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2021. RustHorn: CHC-based Verification for Rust Programs. *TOPLAS* (2021), 15:1–15:54. <https://doi.org/10.1145/3462205>
- [79] Darya Melicher, Yangqingwei Shi, Alex Potanin, and Jonathan Aldrich. 2017. A Capability-Based Module System for Authority Control. In *ECOOP*. 20:1–20:27. <https://doi.org/10.4230/LIPIcs.ECOOP.2017.20>
- [80] Adrian Mettler, David Wagner, and Tyler Close. 2010. Joe-E a Security-Oriented Subset of Java. In *NDSS*. 357–374. <https://www.ndss-symposium.org/ndss2010/joe-e-security-oriented-subset-java>
- [81] Bertrand Meyer. 1992. Applying "Design by Contract". *Computer* 25, 10 (1992), 40–51. <https://doi.org/10.1109/2.161279>
- [82] B. Meyer. 1992. *Eiffel: The Language*. Prentice Hall.
- [83] Mark Samuel Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Ph. D. Dissertation. Johns Hopkins University, Baltimore, Maryland. <https://papers.agoric.com/assets/pdf/papers/robust-composition.pdf>
- [84] Mark Samuel Miller. 2011. Secure Distributed Programming with Object-capabilities in JavaScript. (Oct. 2011). Talk at Vrije Universiteit Brussel, mobicrant-talks.eventbrite.com.
- [85] Mark Samuel Miller, Tom Van Cutsem, and Bill Tulloch. 2013. Distributed Electronic Rights in JavaScript. In *ESOP*. 1–20. https://doi.org/10.1007/978-3-642-37036-6_1
- [86] Mark Samuel Miller, Chip Morningstar, and Bill Frantz. 2000. Capability-based Financial Instruments: From Object to Capabilities. In *Financial Cryptography*. 349–378. https://doi.org/10.1007/3-540-45472-1_24
- [87] Mark Samuel Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. 2008. Safe active content in sanitized JavaScript. , 26 pages. <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=c2de7d8991bdb875fb06d5e5455b0862f0a2d15b> Google white paper.
- [88] Nick Mitchell. 2006. The Runtime Structure of Object Ownership. In *ECOOP*. 74–98. https://doi.org/10.1007/11785477_5
- [89] Scott Moore, Christos Dimoulas, Robert Bruce Findler, Matthew Flatt, and Stephen Chong. 2016. Extensible access control with authorization contracts. In *OOPSLA*. 214–233. <https://doi.org/10.1145/2983990.2984021>
- [90] James H. Morris Jr. 1973. Protection in Programming Languages. *CACM* 16, 1 (1973), 15–21. <https://doi.org/10.1145/361932.361937>
- [91] P. Müller, A. Poetsch-Heffter, and G. T. Leavens. 2006. Modular Invariants for Layered Object Structures. *Science of Computer Programming* 62 (2006), 253–286. <https://doi.org/10.1016/j.scico.2006.03.001>
- [92] Toby Murray. 2010. *Analysing the Security Properties of Object-Capability Patterns*. Ph. D. Dissertation. University of Oxford. <http://ora.ox.ac.uk/objects/uuid:98b0b6b6-eee1-45d5-b32e-d98d1085c612>
- [93] Takashi Nakayama, Yusuke Matsushita, Ken Sakayori, Ryosuke Sato, and Naoki Kobayashi. 2024. Borrowable Fractional Ownership Types for Verification. In *VMCAI*. 224–246. https://doi.org/10.1007/978-3-031-50521-8_11
- [94] James Noble, John Potter, and Jan Vitek. 1998. Flexible Alias Protection. In *ECOOP*. 158–185. <https://doi.org/10.1007/BFb0054091>
- [95] Peter W. O'Hearn. 2019. Incorrectness Logic. 4, *POPL* (2019), 1–32. <https://doi.org/10.1145/3371078>
- [96] Leo Oswald, Grégory M. Essertel, Xilun Wu, Lilliam I. González Alayón, and Tiark Rompf. 2016. Gentrification gone too far? affordable 2nd-class values for fun and (co-)effect. In *OOPSLA*. 234–251.
- [97] M. Parkinson and G. Bierman. 2005. Separation logic and abstraction. In *POPL*. 247–258. <https://doi.org/10.1145/1040305.1040326>
- [98] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachler-Cohen, and Martin Vechev. 2020. VerX: Safety Verification of Smart Contracts. In *IEEE Symp. on Security and Privacy*. 1661–1677. <https://doi.org/10.1109/SP40000.2020.00024>
- [99] John Potter, James Noble, and David G. Clarke. 1998. The Ins and Outs of Objects. In *Australian Software Engineering Conference*. 80–89. <https://doi.org/10.1109/ASWEC.1998.730915>
- [100] Xiaojia Rao, Aina Linn Georges, Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner, and Lars Birkedal. 2023. Iris-Wasm: Robust and Modular Verification of WebAssembly Programs. In *PLDI*. 1096 – 1120. <https://doi.org/10.1145/3591265>

- [101] Xiaojia Rao, Aina Linn Georges, Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner, and Lars Birkedal. 2023. Iris-Wasm: Robust and Modular Verification of WebAssembly Programs. *PLDI* 7 (2023), 1096–1120. <https://doi.org/10.1145/3591265>
- [102] J. C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. IEEE Computer Society, 55–74. <https://doi.org/10.1109/LICS.2002.1029817>
- [103] Dustin Rhodes, Tim Disney, and Cormac Flanagan. 2014. Dynamic Detection of Object Capability Violations Through Model Checking. In *DLS*. 103–112. <https://doi.org/10.1145/2661088.2661099>
- [104] Victor Rivera and Bertrand Meyer. 2020. AutoAlias: Automatic Variable-Precision Alias Analysis for Object-Oriented Programs. *SN Comp. Sci.* 1, 1 (2020), 12:1–12:15. <https://doi.org/10.1007/s42979-019-0012-1>
- [105] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: automating the foundational verification of C code with refined ownership types. In *PLDI*. 158–174. <https://dl.acm.org/doi/10.1145/3453483.3454036>
- [106] Ina Schaefer, Tobias Runge, Alexander Knüppel, Loek Cleophas, Derrick G. Kourie, and Bruce W. Watson. 2018. Towards Confidentiality-by-Construction. In *ISOLA*. 502–515. https://doi.org/10.1007/978-3-030-03418-4_30
- [107] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan. 2019. Safer Smart Contract Programming with Scilla. In *OOPSLA*, Vol. 3. 1–30. <https://doi.org/10.1145/3360611>
- [108] Randall B. Smith and David M. Ungar. 1995. Programming as an Experience: The Inspiration for Self. In *ECOOP*, Walter G. Olthoff (Ed.), Vol. 952. Springer, 303–330. https://doi.org/10.1007/3-540-49538-X_15
- [109] Alexander J. Summers and Sophia Drossopoulou. 2010. Considerate Reasoning and the Composite Pattern. In *VMCAI*. 328–344. https://doi.org/10.1007/978-3-642-11319-2_24
- [110] Alexander J. Summers, Sophia Drossopoulou, and Peter Müller. 2009. Universe-Type-Based Verification Techniques for Mutable Static Fields and Methods. *J. Object Technol.* 8, 4 (2009), 85–125. <https://doi.org/10.5381/JOT.2009.8.4.A4>
- [111] David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and compositional verification of object capability patterns. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 89:1–89:26. <https://doi.org/10.1145/3133913>
- [112] The Ethereum Wiki. 2018. ERC20 Token Standard. (Dec. 2018). https://theethereum.wiki/w/index.php/ERC20_Token_Standard
- [113] David M. Ungar and Randall B. Smith. 1991. SELF: The Power of Simplicity. *LISP Symb. Comput.* 4, 3 (1991), 187–205. https://doi.org/10.1007/3-540-49538-X_15
- [114] Stephan van Staden. 2015. On Rely-Guarantee Reasoning. In *MPC*. 30–49. https://link.springer.com/chapter/10.1007/978-3-319-19797-5_2
- [115] Guannan Wei, Danning Xie, Wuqi Zhang, Yongwei Yuan, and Zhuo Zhang. 2024. Consolidating Smart Contracts with Behavioral Contracts. In *PLDI*. 965 – 989. <https://doi.org/10.1145/3656416>
- [116] Maurice V. Wilkes and Roger M. Needham. 1980. The Cambridge Model Distributed System. *ACM SIGOPS Oper. Syst. Rev.* 14, 1 (1980), 21–29. <https://doi.org/10.1145/850693.850695>
- [117] Anxhelo Xhebraj, Oliver Bracevac, Guannan Wei, and Tiark Rompf. 2022. What If We Don’t Pop the Stack? The Return of 2nd-Class Values. In *ECOOP*. 15:1–15:29. <https://doi.org/10.4230/LIPLcs.ECOOP.2022.15>
- [118] Yichen Xu and Martin Odersky. 2024. A Formal Foundation of Reach Capabilities. In *Programming*. 134–138. <https://doi.org/10.1145/3660829.3660851>
- [119] Conrad Zimmerman, Jenna DiVincenzo, and Jonathan Aldrich. 2024. Sound Gradual Verification with Symbolic Execution. *POPL* 8, 2547–2576. <https://doi.org/10.1145/3632927>

A APPENDIX TO SECTION 3 – THE PROGRAMMING LANGUAGE \mathcal{L}_{ul}

We introduce \mathcal{L}_{ul} , a simple, typed, class-based, object-oriented language.

A.1 Syntax

The syntax of \mathcal{L}_{ul} is given in Fig. 4¹³. To reduce the complexity of our formal models, as is usually done, CITE - CITE, \mathcal{L}_{ul} lacks many common languages features, omitting static fields and methods, interfaces, inheritance, subsumption, exceptions, and control flow. \mathcal{L}_{ul} and which may be defined recursively.

\mathcal{L}_{ul} modules (M) map class names (C) to class definitions ($ClassDef$). A class definition consists of a list of field definitions, ghost field definitions, and method definitions. Fields, ghost fields, and methods all have types, C ; types are classes. Ghost fields may be optionally annotated as `intrnl`, requiring the argument to have an internal type, and the body of the ghost field to only contain references to internal objects. This is enforced by the limited type system of \mathcal{L}_{ul} . A program state (σ) is a pair of a stack and a heap. The stack is a non-empty list of frames (ϕ), and the heap (χ) is a map from addresses (α) to objects (o). A frame consists of a local variable map and a continuation `.cont` that represents the statements that are yet to be executed (s). A statement is either a field read ($x := y.f$), a field write ($x.f := y$), a method call ($u := y_0.m(\bar{y})$), a constructor call (`new C`), a sequence of statements ($s; s$), or empty (ϵ).

\mathcal{L}_{ul} also includes syntax for expressions e that may be used in writing specifications or the definition of ghost fields.

A.2 Semantics

\mathcal{L}_{ul} is a simple object oriented language, and the operational semantics (given in Fig. 5 and discussed later) do not introduce any novel or surprising features. The operational semantics make use of several helper definitions that we define here.

We provide a definition of reference interpretation in Definition A.1

Definition A.1. For a frame $\phi = (\bar{x} \mapsto \bar{v}, s)$, and a program state $\sigma = (\bar{\phi} \cdot \phi, \chi)$, we define:

- $[x]_{\phi} \triangleq v_i$ if $x = x_i$
- $[x]_{\sigma} \triangleq [x]_{\phi}$
- $[\alpha.f]_{\sigma} \triangleq v_i$ if $\chi(\alpha) = (_; \bar{f} \mapsto \bar{v})$, and $f_i = f$
- $[x.f]_{\sigma} \triangleq [\alpha.f]_{\sigma}$ where $[x]_{\sigma} = \alpha$
- $\phi.\text{cont} \triangleq s$
- $\sigma.\text{cont} \triangleq \phi.\text{cont}$
- $\phi[\text{cont} \mapsto s'] \triangleq (\bar{x} \mapsto \bar{v}, s')$
- $\sigma[\text{cont} \mapsto s'] \triangleq (\bar{\phi} \cdot \phi[\text{cont} \mapsto s'], \chi)$
- $\phi[x' \mapsto v'] \triangleq ((\bar{x} \mapsto \bar{v})[x' \mapsto v'], s)$
- $\sigma[x' \mapsto v'] \triangleq ((\bar{\phi} \cdot (\phi[x' \mapsto v'])), \chi)$
- $\sigma[\alpha \mapsto o] \triangleq ((\bar{\phi} \cdot \phi), \chi[\alpha \mapsto o])$
- $\sigma[\alpha.f' \mapsto v'] \triangleq \sigma[\alpha \mapsto o]$ if $\chi(\alpha) = (C, \bar{f} \mapsto \bar{v})$, and $o = (C; (\bar{f} \mapsto \bar{v})[f' \mapsto v'])$

That is, a variable x , or a field access on a variable $x.f$ has an interpretation within a program state of value v if x maps to v in the local variable map, or the field f of the object identified by x points to v .

Definition A.2 defines the class lookup function an object identified by variable x .

¹³Our motivating example is provided in a slightly richer syntax for greater readability.

Definition A.2 (Class Lookup). For program state $\sigma = (\bar{\phi} \cdot \phi, \chi)$, class lookup is defined as

$$\text{classOf}(\sigma, x) \triangleq C \quad \text{if} \quad \chi(\lfloor x \rfloor_\sigma) = (C, _)$$

Module linking is defined for modules with disjoint definitions:

Definition A.3. For all modules \bar{M} and M , if the domains of \bar{M} and M are disjoint, we define the module linking function as $M \cdot \bar{M} \triangleq M \cup M'$.

That is, their linking is the union of the two if their domains are disjoint.

Definition A.4 defines the method lookup function for a method call m on an object of class C .

Definition A.4 (Method Lookup). For module \bar{M} , class C , and method name m , method lookup is defined as

$$\text{Meth}(\bar{M}, C, m) \triangleq \text{pr method } m \ (\bar{x} : \bar{T}) : T \{ s \}$$

if there exists an M in \bar{M} , so that $M(C)$ contains the definition $\text{pr method } m \ (\bar{x} : \bar{T}) : T \{ s \}$

Definition A.5 looks up all the field identifiers in a given class

Definition A.5 (Fields Lookup). For modules \bar{M} , and class C , fields lookup is defined as

$$\text{fields}(\bar{M}, C) \triangleq \{ f \mid \exists M \in \bar{M}. s.t. M(C) \text{ contains the definition } \text{field } f : T \}$$

We define what it means for two objects to come from the same module

Definition A.6 (Same Module). For program state σ , modules \bar{M} , and variables x and y , we define

$$\text{SameModule}(x, y, \sigma, \bar{M}) \triangleq \exists C, C', M [M \in \bar{M} \wedge C' \in M \wedge \text{classOf}(\sigma, x) = C \wedge \text{classOf}(\sigma, y) = C']$$

As we already said in §3.3.3, we forbid assignments to a method's parameters. To do that, the following function returns the identifiers of the formal parameters of the currently active method.

Definition A.7. For program state σ :

$$\text{Prms}(\sigma, \bar{M}) \triangleq \bar{x} \quad \text{such that} \quad \exists \bar{\phi}, \phi_k, \phi_{k+1}, C, p.$$

$$[\sigma = (\bar{\phi} \cdot \phi_k \cdot \phi_{k+1}, \chi) \wedge \phi_k.\text{cont} = _ := y_0.m(_); _ \wedge$$

$$\text{classOf}((\phi_{k+1}, \chi), \text{this}) \wedge \text{Meth}(\bar{M}, C, m) = p \ C :: m(\bar{x} : _): _ \{ _ \}]$$

While the small-step operational semantics of \mathcal{L}_{ul} is given in Fig. 5, specification satisfaction is defined over an abstracted notion of the operational semantics that models the open world.

An *Initial* program state contains a single frame with a single local variable `this` pointing to a single object in the heap of class `Object`, and a continuation.

Definition A.8 (Initial Program State). A program state σ is said to be an initial state ($\text{Initial}(\sigma)$) if and only if

$$\bullet \sigma = (((\text{this} \mapsto \alpha), s); (\alpha \mapsto (\text{Object}, \emptyset)))$$

for some address α and some statement s .

We provide a semantics for expression evaluation is given in Fig. 10. That is, given a module M and a program state σ , expression e evaluates to v if $M, \sigma, e \hookrightarrow v$. Note, the evaluation of expressions is separate from the operational semantics of \mathcal{L}_{ul} , and thus there is no restriction on field access.

Proof of lemma 3.6 The first assertion is proven by unfolding the definition of $_ \models _$.

The second assertion is proven by case analysis on the execution relation $_, \sigma \twoheadrightarrow \sigma'$. The assertion gets established when we call a method, and is preserved through all the execution steps, because we do not allow assignments to the formal parameters.

End Proof

We now prove lemma B.2:

Proof of lemma B.2

- We first show that $(\bar{M}, \sigma_{sc}); \sigma \rightsquigarrow \sigma' \wedge k < |\sigma|_{sc} \implies \lfloor y \rfloor_{\sigma[k]} = \lfloor y \rfloor_{\sigma'[k]}$ This follows easily from the operational semantics, and the definitions.
- By induction on the earlier part, we obtain that $\bar{M}; \sigma \rightsquigarrow^* \sigma' \wedge k < |\sigma| \implies \lfloor y \rfloor_{\sigma[k]} = \lfloor y \rfloor_{\sigma'[k]}$
- We now show that $\bar{M}; \sigma \rightsquigarrow_{fin}^* \sigma' \wedge y \notin Vs(\sigma.\text{cont}) \implies \lfloor y \rfloor_{\sigma} = \lfloor y \rfloor_{\sigma'}$ by induction on the number of steps, and using the earlier lemma.

End Proof

Lemma A.9 states that initila states are well-formed, and that (2) a pre-existing object, locally reachable after any number of scoped execution steps, was locally reachable at the first step.

Lemma A.9. For all modules \bar{M} , states σ, σ' , and frame ϕ :

- (1) $\text{Initial}(\sigma) \implies \bar{M} \models \sigma$
- (2) $\bar{M}; \sigma \rightsquigarrow^* \sigma' \implies \text{dom}(\sigma) \cap \text{LocRchbl}(\sigma') \subseteq \text{LocRchbl}(\sigma)$

Consider Fig. 3 . Lemma A.9, part 2 promises that any objects locally reachable in σ_{14} which already existed in σ_8 , were locally reachable in σ_8 . However, the lemma is only applicable to scoped execution, and as $\bar{M}; \sigma_8 \rightsquigarrow^* \sigma_{17}$, the lemma does not promise that objects locally reachable in σ_{17} which already existed in σ_8 , were locally accessible in σ_8 – namely it could be that objects are made globally reachable upon method return, during the step from σ_{14} to σ_{15} .

Finally, we define the evaluation of expressions, which, as we already said, represent ghost code.

$$\begin{array}{c}
 M, \sigma, v \hookrightarrow v \quad (\text{E-VAL}) \qquad M, \sigma, x \hookrightarrow \lfloor x \rfloor_{\sigma} \quad (\text{E-VAR}) \qquad \frac{M, \sigma, e \hookrightarrow \alpha}{M, \sigma, e.f \hookrightarrow \lfloor \alpha.f \rfloor_{\sigma}} \quad (\text{E-FIELD}) \\
 \\
 \frac{M, \sigma, e_0 \hookrightarrow \alpha \quad \bar{M}, \sigma, e \hookrightarrow v \quad M(\text{classOf}(\sigma, \alpha)) \text{ contains } \text{ghost } gf(x:T)\{e\} : T' \quad M, \sigma, \lfloor v/x \rfloor e \hookrightarrow v}{M, \sigma, e_0.gf(\bar{v}) \hookrightarrow v} \quad (\text{E-GHOST})
 \end{array}$$

Fig. 10. \mathcal{L}_{ul} Expression evaluation

A.3 M_{ghost} Accounts expressed through ghost fields

We revisit the bank account example, to demonstrate the use of ghost fields. In Fig. 11, accounts belong to banks, and their `blnce` is kept in a ledger. Thus, `account.blnce` is a ghost field which involves a recursive search through that ledger.

```

16181 module Mghost
16192
16203   class Shop ...
16214
16225   class Account
16236     field bank: Bank
16247     field key:Key
16258     public method transfer(dest:Account, key':Key, amt:nat)
16269       if (this.key==key')
16270         this.bank.decBalance(this,amt);
16281         this.bank.incBalance(dest.amt);
16292     public method set(key':Key)
16303       if (this.key==null) this.key=key'
16314     ghost balance(): int
16325       res:=bank.balance(this)
16336
16347   class Bank
16358     field ledger: Ledger
16369     method incBalance(a:Account, amt: nat)
16380       this.ledger.decBalance(a,amt)
16391     private method decBalance(a:Account, amt: nat)
16402       this.ledger.decBalance(a,amt)
16413     ghost balance(acc):int
16424       res:=this.ledger.balance(acc)
16435
16446   class Ledger
16457     acc:Acc
16468     bal:int
16479     next:Ledger
16490     ghost balance(a:Acc):int
16501       if this.acc==a
16512         res:=retrun bal
16523       else
16534         res:=this.next.balance(a)
16545

```

Fig. 11. M_{ghost} – a module with ghost fields

B APPENDIX TO SECTION 3.3 – FUNDAMENTAL CONCEPTS

Lemma B.1 says, essentially, that scoped executions describe the same set of executions as those starting at an initial state¹⁴. For instance, revisit Fig. 3, and assume that σ_6 is an initial state. We have $\bar{M}; \sigma_{10} \dashv\vdash^* \sigma_{14}$ and $\bar{M}; \sigma_{10} \not\sim^* \sigma_{14}$, but also $\bar{M}; \sigma_6 \rightsquigarrow^* \sigma_{14}$.

Lemma B.1. For all modules \bar{M} , state σ_{init} , σ, σ' , where σ_{init} is initial:

- $\bar{M}; \sigma \rightsquigarrow^* \sigma' \implies \bar{M}; \sigma \dashv\vdash^* \sigma'$
- $\bar{M}; \sigma_{init} \dashv\vdash^* \sigma' \implies \bar{M}; \sigma_{init} \rightsquigarrow^* \sigma'$.

Lemma B.2 says that scoped execution does not affect the contents of variables in earlier frames. and that the interpretation of a variable remains unaffected by scoped execution of statements which do not mention that variable. More in Appendix B.

Lemma B.2. For any modules \bar{M} , states σ, σ' , variable y , and number k :

- $\bar{M}; \sigma \rightsquigarrow^* \sigma' \wedge k < |\sigma| \implies \lfloor y \rfloor_{\sigma[k]} = \lfloor y \rfloor_{\sigma'[k]}$
- $\bar{M}; \sigma \rightsquigarrow_{fin}^* \sigma' \wedge y \notin Vs(\sigma.cont) \implies \lfloor y \rfloor_{\sigma} = \lfloor y \rfloor_{\sigma'}$

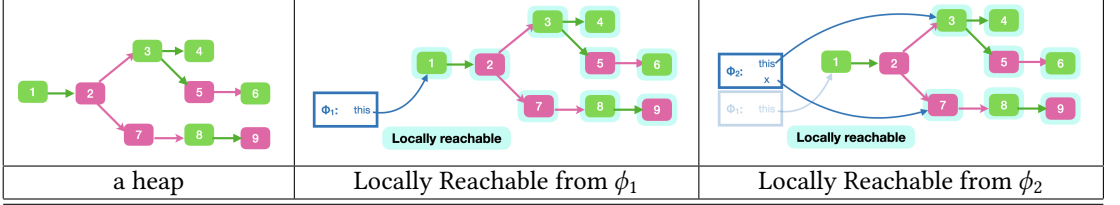


Fig. 12. -Locally Reachable Objects

Fig. 12 illustrates local reachability: In the middle pane the top frame is ϕ_1 which maps `this` to o_1 ; all objects are locally reachable. In the right pane the top frame is ϕ_2 , which maps `this` to o_3 , and x to o_7 ; now o_1 and o_2 are no longer locally reachable.

Proof of lemma B.1

- By unfolding and folding the definitions.
- By unfolding and folding the definitions, and also, by the fact that $|\sigma_{init}|=1$, i.e. minimal.

End Proof

Proof of lemma B.2

- We unfolding the definition of $\bar{M}; \sigma \rightsquigarrow^* \sigma' \bar{M}; \sigma \dashv\vdash^* \sigma'$ and the rules of the operational semantics.
- Take $k = |\sigma|$. We unfold the definition from 3.2, and obtain that $\sigma = \sigma'$ or, $\exists \sigma_1, \dots, \sigma_{n1}. \forall i \in [1..n) [\bar{M}; \sigma_i \dashv\vdash^* \sigma_{i+1} \wedge |\sigma_1| \leq |\sigma_{i+1}| \wedge \sigma = \sigma_1 \wedge \sigma' = \sigma_n]$
Consider the second case. Take any $i \in [1..n)$. Then, by Definition, $k \leq |\sigma|$. If $k = |\sigma_i|$, then we are executing part of $\sigma.prgcont$, and because $y \notin Vs(\sigma.cont)$, we get $\lfloor y \rfloor_{\sigma[i]} = \lfloor y \rfloor_{\sigma_{i+1}[k]}$. If $k = |\sigma_i|$, then we apply the bullet from above, and also obtain $\lfloor y \rfloor_{\sigma[i]} = \lfloor y \rfloor_{\sigma_{i+1}[k]}$
This gives that $\lfloor y \rfloor_{\sigma[k]} = \lfloor y \rfloor_{\sigma'[k]}$. Moreover, because $\bar{M}; \sigma \rightsquigarrow_{fin}^* \sigma'$ we obtain that $|\sigma| = |\sigma'| = k$. Therefore, we have that $\lfloor y \rfloor_{\sigma} = \lfloor y \rfloor_{\sigma'}$.

¹⁴An Initial state's heap contains a single object of class `Object`, and its stack consists of a single frame, whose local variable map is a mapping from `this` to the single object, and whose continuation is any statement. (See Def. A.8)

End Proof

We also prove that in well-formed states ($\models \sigma$), all objects locally reachable from a given frame also locally reachable from the frame below.

Lemma B.3. $\models \sigma \wedge k < |\sigma| \implies \text{LocRchbl}(\sigma[k+1]) \subseteq \text{LocRchbl}(\sigma[k])$

PROOF. By unfolding the definitions: Everything that is in $\sigma[k+1]$ is reachable from its frame, and everything that is reachable from the frame of $\sigma[k+1]$ is also reachable from the frame of $\sigma[k]$. We then apply that $\models \sigma$

□

Proof of lemma 3.6

- (1) By unfolding and folding the definitions. Namely, everything that is locally reachable in σ' is locally reachable through the frame ϕ , and everything in the frame ϕ is locally reachable in σ .
- (2) We require that $\models \sigma$ – as we said earlier, we require this implicitly. Here we apply induction on the execution. Each step is either a method call (in which case we apply the bullet from above), or a return statement (then we apply lemma B.3), or the creation of a new object (in which case reachable set is the same as that from previous state plus the new object), or an assignment to a variable (in which case the locally reachable objects in the new state are a subset of the locally reachable from the old state), or a an assignment to a field. In the latter case, the locally reachable objects are also a subset of the locally reachable objects from the previous state.

End Proof

C APPENDIX TO SECTION 4 – ASSERTIONS

Figure 13 illustrates “protected from” and “protected”. In the first row we highlight in yellow the objects protected from other objects. Thus, all objects except o_6 are protected from o_5 (left pane); all objects except o_8 are protected from o_7 (middle pane); and all objects except o_3 , o_6 , o_7 , and o_8 are protected from o_2 (right pane). Note that o_6 is not protected from o_2 , because o_5 is reachable from o_2 , is external, and has direct access to o_6 .

In the third row of Figure 13 we show three states: σ_1 has top frame ϕ_1 , which has one variable, `this`, pointing to o_1 , while σ_2 has top frame ϕ_2 ; it has two variables, `this` and `x` pointing to o_3 and o_7 , and σ_3 has top frame ϕ_3 ; it has two variables, `this` and `x`, pointing to o_7 and o_3 . We also highlight the protected objects with a yellow halo. Note that o_3 is protected in σ_2 , but is not protected in σ_3 . This is so, because $[\text{this}]_{\sigma_3}$ is external, and o_3 is an argument to the call. As a result, during the call, o_7 may obtain direct access to o_3 .

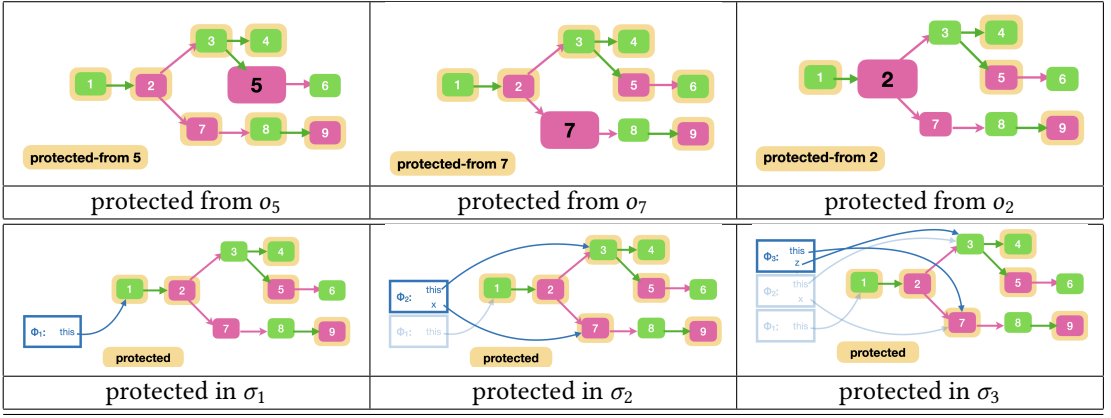


Fig. 13. Protection. Pink objects are external, and green objects are internal.

In order to prove 4.5 from the next appendix, we first formulate and prove the following auxiliary lemma, which allows us to replace any variable x in an extended expression e , by its interpretation

Lemma C.1. For all extended expressions e , addresses α and variables x , so that $x \in \text{dom}(\sigma)$:

$$\bullet \quad M, \sigma, e \hookrightarrow \alpha \iff M, \sigma, e[\llbracket x \rrbracket_\sigma / x] \hookrightarrow \alpha$$

Note that in the above we require that $x \in \text{dom}(\sigma)$, in order to ensure that the replacement $[\llbracket x \rrbracket_\sigma / x]$ is well-defined. On the other hand, we do not require that $x \in \text{Fv}(e)$, because if $x \notin \text{Fv}(e)$, then $e[\llbracket x \rrbracket_\sigma / x] \stackrel{\text{txt}}{=} e$ and the guarantee from above becomes a tautology.

Proof of Lemma C.1 The proof goes by induction on the structure of e – as defined in Def. 4.1 – and according to the expression evaluation rules from Fig. 10. **End of Proof**

D APPENDIX TO SECTION 4.3 – PRESERVATION OF SATISFACTION

Proof of lemma 4.5

We first prove that for any M, A, σ

- (1) To show that $M, \sigma \models A \iff M, \sigma \models A[\lfloor x \rfloor_\sigma / x]$
The proof goes by induction on the structure of A , application of Defs. 4.3, 4.4, and 4.4.
- (2) To show that $M, \sigma \models A \iff M, \sigma[\text{cont} \mapsto \text{stmt}] \models A$
The proof goes by induction on the structure of A , application of Defs. 4.3, 4.4, and 4.4.

The lemma itself then follows from (1) and (2) proven above.

End Proof

In addition to what is claimed in Lemma 4.5, it also holds that

Lemma D.1. $M, \sigma, e \hookrightarrow \alpha \implies [M, \sigma \models A \iff M, \sigma \models A[\alpha/e]]$

PROOF. by induction on the structure of A , application of Defs. 4.3, 4.4, and 4.4, and , and auxiliary lemma C.1.

□

D.1 Stability

We first give complete definitions for the concepts of $Stbl(_)$ and $Stb^+(_)$

Definition D.2. $[Stbl(_)]$ assertions:

$$\begin{aligned} Stbl(\langle e \rangle) &\triangleq \text{false} \\ Stbl(\langle e \rangle \Leftarrow \bar{u}) &= Stbl(e : \text{int1}) = Stbl(e) = Stbl(e : C) \triangleq \text{true} \\ Stbl(A_1 \wedge A_2) &\triangleq Stbl(A_1) \wedge Stbl(A_2) \quad Stbl(\forall x : C.A) = Stbl(\neg A) \triangleq Stbl(A) \end{aligned}$$

Definition D.3 ($Stb^+(_)$). assertions:

$$\begin{aligned} Stb^+(\langle e \rangle) &= Stb^+(\langle e \rangle \Leftarrow \bar{u}) = Stb^+(e : \text{int1}) = Stb^+(e) = Stb^+(e : C) \triangleq \text{true} \\ Stb^+(A_1 \wedge A_2) &\triangleq Stb^+(A_1) \wedge Stb^+(A_2) \\ Stb^+(\forall x : C.A) &\triangleq Stb^+(A) \\ Stb^+(\neg A) &\triangleq Stbl(A) \end{aligned}$$

The definition of $Stb^+(_)$ is less general than would be possible. E.g., $(\langle x \rangle \rightarrow x.f = 4) \rightarrow x.f.3 = 7$ does not satisfy our definition of $Stb^+(_)$. We have given these less general definitions in order to simplify both our definitions and our proofs.

Proof of lemma 4.6 Take any state σ , frame ϕ , assertion A ,

- To show

$$Stbl(A) \wedge Fv(A) = \emptyset \implies [M, \sigma \models A \iff M, \sigma \nabla \phi \models A]$$

By induction on the structure of the definition of $Stbl(A)$.

- To show

$$Stb^+(A) \wedge Fv(A) = \emptyset \wedge M \cdot \bar{M} \models \sigma \nabla \phi \wedge M, \sigma \models A \wedge M, \sigma \nabla \phi \models \text{int1} \implies M, \sigma \nabla \phi \models A$$

By induction on the structure of the definition of $Stb^+(A)$. The only interesting case is when A has the form $\langle e \rangle$. Because $Fv(A) = \emptyset$, we know that $\lfloor e \rfloor_\sigma = \lfloor e \rfloor_{\sigma \nabla \phi}$. Therefore, we assume that $\lfloor e \rfloor_\sigma = \alpha$ for some α , assume that $M, \sigma \models \langle \alpha \rangle$, and want to show that $M, \sigma \nabla \phi \models \langle \alpha \rangle$. From $M \cdot \bar{M} \models \sigma \nabla \phi$ we obtain that $Rng(\phi) \subseteq Rng(\sigma)$. From this, we obtain that $LocRchbl(\sigma \nabla \phi) \subseteq LocRchbl(\sigma)$. The rest follows by unfolding and folding Def. 4.4.

End Proof

D.2 Encapsulation

Proofs of adherence to \mathcal{L}^{spec} specifications hinge on the expectation that some, specific, assertions cannot be invalidated unless some internal (and thus known) computation took place. We call such assertions *encapsulated*. We define the judgement, $M \vdash Enc(A)$, in terms of the judgment $M; \Gamma \vdash Enc(A); \Gamma'$ from Fig. 14. This judgements ensures that any objects whose fields are read in the validation of A are internal, that $\langle _ \rangle \Leftarrow _$ does not appear in A , and that protection assertions (ie $\langle _ \rangle$ or $\langle _ \rangle \Leftarrow _$) do not appear in negative positions in A . The second environment in this judgement, Γ' , is used to keep track of any variables introduces in that judgment, e.g. we would have that

$$M_{good}, \emptyset \vdash Enc(a : Account \wedge k : Key \wedge \langle a.key \rangle \wedge a.key \neq k); (a : Account, k : Key).$$

We assume a type judgment $M; \Gamma \vdash e : \text{int1}$ which says that in the context of Γ , the expression e belongs to a class from M . We also assume that the judgement $M; \Gamma \vdash e : \text{int1}$ can deal with ghostfields – namely, ghost-methods have to be type checked in the context of M and therefore they will only read the state of internal objects. Note that it is possible for $M; \Gamma \vdash Enc(e); \Gamma'$ to hold and $M; \Gamma \vdash e : \text{int1}$ not to hold – c.f. rule ENC_1.

ENC_1	ENC_2	ENC_3
$\frac{M; \Gamma \vdash e : \text{int1} \quad M; \Gamma \vdash Enc(e); \Gamma'}{M; \Gamma \vdash Enc(e.f); \Gamma'}$	$\frac{M; \Gamma \vdash Enc(e); \Gamma'}{M; \Gamma \vdash Enc(e : C); (\Gamma', e : C)}$	$\frac{M; \Gamma \vdash Enc(A); \Gamma' \quad A \text{ does not contain } \langle _ \rangle}{M; \Gamma \vdash Enc(\neg A); \Gamma'}$
ENC_4	ENC_5	ENC_6
$\frac{M; \Gamma \vdash Enc(A_1); \Gamma'' \quad M; \Gamma'' \vdash Enc(A_2); \Gamma'}{M; \Gamma \vdash Enc(A_1 \wedge A_2); \Gamma'}$	$\frac{M; \Gamma, x : C \vdash Enc(A); \Gamma'}{M; \Gamma \vdash Enc(\forall x : C.A); \Gamma'}$	$\frac{M; \Gamma \vdash Enc(e); \Gamma'}{M; \Gamma \vdash Enc(e : \text{ext1}); \Gamma'}$
ENC_7		
$\frac{M; \Gamma \vdash Enc(e); \Gamma'}{M; \Gamma \vdash Enc(\langle e \rangle); \Gamma'}$		

Fig. 14. The judgment $M; \Gamma \vdash Enc(A); \Gamma'$

Definition D.4 (An assertion A is *encapsulated* by module M).

- $M \vdash Enc(A) \triangleq \exists \Gamma. [M; \emptyset \vdash Enc(A); \Gamma]$ as defined in Fig. 14.

To motivate the design of our judgment $M; \Gamma \vdash Enc(A); \Gamma'$, we first give a semantic notion of encapsulation:

Definition D.5. An assertion A is *semantically encapsulated* by module M :

- $M \models Enc(A) \triangleq \forall \bar{M}, \sigma, \sigma'. [M, \sigma \models (A \wedge \text{ext1}) \wedge M \cdot \bar{M}; \sigma \rightsquigarrow \sigma' \implies M, \sigma' \models \sigma[A]]$

More on Def. D.5 If the definition D.5 or in lemma 4.8 we had used the more general execution, $M \cdot \bar{M}; \sigma \twoheadrightarrow \sigma'$, rather than the scoped execution, $M \cdot \bar{M}; \sigma \rightsquigarrow \sigma'$, then fewer assertions would have been encapsulated. Namely, assertions like $\langle x.f \rangle$ would not be encapsulated. Consider, e.g., a heap χ , with objects 1, 2, 3 and 4, where 1, 2 are external, and 3, 4 are internal, and 1 has fields pointing to 2 and 4, and 2 has a field pointing to 3, and 3 has a field f pointing to 4. Take state $\sigma = (\phi_1, \phi_2, \chi)$, where ϕ_1 's receiver is 1, ϕ_2 's receiver is 2, and there are no local variables. We have $\dots \sigma \models \text{ext1} \wedge \langle 3.f \rangle$. We return from the most recent all, getting $\dots; \sigma \twoheadrightarrow \sigma'$ where $\sigma' = (\phi_1, \chi)$; and have $\dots, \sigma' \not\models \langle 3.f \rangle$.

Example D.6. For an assertion $A_{bal} \triangleq a : \text{Account} \wedge a.\text{balance} = b$, and modules M_{bad} and M_{fine} from § 2, we have $M_{bad} \models \text{Enc}(A_{bal})$, and $M_{bad} \models \text{Enc}(A_{bal})$.

Example D.7. Assume further modules, M_{unp} and M_{prt} , which use ledgers mapping accounts to their balances, and export functions that update this map. In M_{unp} the ledger is part of the internal module, while in M_{prt} it is part of the external module. Then $M_{unp} \not\models \text{Enc}(A_{bal})$, and $M_{prt} \models \text{Enc}(A_{bal})$. Note that in both M_{unp} and M_{prt} , the term $a.\text{balance}$ is a ghost field.

Note D.8. Relative protection is not encapsulated, (e.g. $M \not\models \text{Enc}(\langle x \rangle \leftarrow * y)$), even though absolute protection is (e.g. $M \models \text{Enc}(\langle x \rangle)$). Encapsulation of an assertion does not imply encapsulation of its negation; for example, $M \not\models \text{Enc}(\neg \langle x \rangle)$.

More on Def. D.4 This definition is less permissive than necessary. For example $M \not\models \text{Enc}(\neg(\neg \langle x \rangle))$ even though $M \models \text{Enc}(\neg(\neg \langle x \rangle))$. Namely, $\neg(\neg \langle x \rangle) \equiv \langle x \rangle$ and $M \models \text{Enc}(\langle x \rangle)$. A more permissive, sound, definition, is not difficult, but not the main aim of this work. We gave this, less permissive definition, in order to simplify the definitions and the proofs.

Proof of lemma 4.8 This says that $M \vdash \text{Enc}(A)$ implies that $M \vdash \text{Enc}(A)$.

We first prove that

(*) Assertions A_{poor} which do not contain $\langle _ \rangle$ or $\langle _ \rangle \leftarrow * _$ are preserved by any external step. Namely, such an assertion only depends on the contents of the fields of internal objects, and these are not modified by external steps. Such an A_{poor} is defined essentially through

$$A_{poor} ::= e \mid e : C \mid \neg A_{poor} \mid A_{poor} \wedge A_{poor} \mid \forall x : C. A_{poor} \mid e : \text{extl}$$

We can prove (*) by induction on the structure of A_{poor} and case analysis on the execution step.

We then prove Lemma 4.8 by induction on the structure of A .

– The cases ENC_1, ENC_2, and ENC_6 are straight application of (*).

– The case ENC_3 also follows from (*), because any A which satisfies $\text{Enc}(A)$ and which does not contain $\langle _ \rangle$ is an A_{poor} assertion.

– The cases ENC_4 and ENC_5 follow by induction hypothesis.

– The case ENC_7 is more interesting.

We assume that σ is an external state, that $\dots; \sigma \rightsquigarrow \sigma'$, and that $\dots \sigma \models \langle e \rangle$. By definition, the latter means that

(**) no locally reachable external object in σ has a field pointing to e ,
nor is e one of the variables.

We proceed by case analysis on the step $\dots; \sigma \rightsquigarrow \sigma'$.

- If that step was an assignment to a local variable x , then this does not affect $\langle \sigma[e] \rangle$ because in $\lfloor e \rfloor_{\sigma} = \lfloor \sigma[e] \rfloor_{\sigma'}$, and $\dots \sigma' \models x \neq re$.

- If that step was an assignment to an external object's field, of the form $x.f := y$ then this does not affect $\langle \sigma[e] \rangle$ either. This is so, because $\text{Enc}(e)$ gives that $\lfloor e \rfloor_{\sigma} = \lfloor ' \rfloor_{\sigma} \sigma[e]$ – namely the evaluation of e does not read x 's fields, since x is external. And moreover, the assignment $x.f := y$ cannot create a new, unprotected path to e (unprotected means here that the penultimate element in that path is external), because then we would have had in σ an unprotected path from y to e .

- If that step was a method call, then we apply lemma 3.6 which says that all objects reachable in σ' were already reachable in σ .

- Finally, we do not consider method return (i.e. the rule RETURN), because we are looking at $\dots; _ \rightsquigarrow _$ execution steps rather than $\dots; _ \rightarrow _$ steps.

End Proof

E APPENDIX TO SECTION 5 – SPECIFICATIONS

Example E.1 (Badly Formed Method Specifications). S_{9,bad_1} is not a well-formed specification, because A' is not a formal parameter, nor free in the precondition.

$$\begin{aligned}
 S_{9,bad_1} &\triangleq \{ a : \text{Account} \wedge \langle a \rangle \} \\
 &\quad \text{public Account} :: \text{set}(\text{key}' : \text{Key}) \\
 &\quad \{ \langle a \rangle \wedge \langle a'.\text{key} \rangle \} \parallel \{ \text{true} \} \\
 S_{9,bad_2} &\triangleq \{ a : \text{Account} \wedge \langle a \rangle \} \\
 &\quad \text{public Account} :: \text{set}(\text{key}' : \text{Key}) \\
 &\quad \{ \langle a \rangle \wedge \langle a'.\text{key} \rangle \} \parallel \{ \text{this.blnc} \}
 \end{aligned}$$

Example E.2 (More Method Specifications). S_7 below guarantees that `transfer` does not affect the balance of accounts different from the receiver or argument, and if the key supplied is not that of the receiver, then no account's balance is affected. S_8 guarantees that if the key supplied is that of the receiver, the correct amount is transferred from the receiver to the destination. S_9 guarantees that `set` preserves the protectedness of a key.

$$\begin{aligned}
 S_7 &\triangleq \{ a : \text{Account} \wedge a.\text{blnc} = b \wedge (\text{dst} \neq a \neq \text{this} \vee \text{key}' \neq a.\text{key}) \} \\
 &\quad \text{public Account} :: \text{transfer}(\text{dst} : \text{Account}, \text{key}' : \text{Key}, \text{amt} : \text{nat}) \\
 &\quad \{ a.\text{blnc} = b \} \parallel \{ a.\text{blnc} = b \} \\
 S_8 &\triangleq \{ \text{this} \neq \text{dst} \wedge \text{this.blnc} = b \wedge \text{dst.blnc} = b' \} \\
 &\quad \text{public Account} :: \text{transfer}(\text{dst} : \text{Account}, \text{key}' : \text{Key}, \text{amt} : \text{nat}) \\
 &\quad \{ \text{this.blnc} = b - \text{amt} \wedge \text{dst.blnc} = b' + \text{amt} \} \\
 &\quad \parallel \{ \text{this.blnc} = b \wedge \text{dst.blnc} = b' \} \\
 S_9 &\triangleq \{ a : \text{Account} \wedge \langle a.\text{key} \rangle \} \\
 &\quad \text{public Account} :: \text{set}(\text{key}' : \text{Key}) \\
 &\quad \{ \langle a.\text{key} \rangle \} \parallel \{ \langle a.\text{key} \rangle \}
 \end{aligned}$$

E.1 Examples of Semantics of our Specifications

Example E.3. We revisit the specifications given in Sect. 2.1, the three modules from Sect. 2.1.2, and Example E.2

$$\begin{array}{llll}
 M_{\text{good}} \models S_1 & M_{\text{good}} \models S_2 & M_{\text{good}} \models S_3 & M_{\text{good}} \models S_5 \\
 M_{\text{bad}} \models S_1 & M_{\text{bad}} \not\models S_2 & M_{\text{bad}} \not\models S_3 & M_{\text{bad}} \not\models S_5 \\
 M_{\text{fine}} \models S_1 & M_{\text{fine}} \models S_2 & M_{\text{fine}} \not\models S_3 & M_{\text{fine}} \not\models S_5
 \end{array}$$

Example E.4. For Example 5.5, we have $M_{\text{good}} \models S_7$ and $M_{\text{bad}} \models S_7$ and $M_{\text{fine}} \models S_7$. Also, $M_{\text{good}} \models S_8$ and $M_{\text{bad}} \models S_8$ and $M_{\text{fine}} \models S_8$. However, $M_{\text{good}} \models S_9$, while $M_{\text{bad}} \not\models S_9$.

Example E.5. For any specification $S \triangleq \{ A \} p C :: \overline{m(x : \overline{C})} \{ A' \}$ and any module M which does not have a class C with a method m with formal parameter types \overline{C} , we have that $M \models S$. Namely, if a method were to be called with that signature on a C from M , then execution would be stuck, and the requirements from Def. 5.4(3) would be trivially satisfied. Thus, $M_{\text{fine}} \models S_8$.

E.1.1 Free variables in well-formed specifications. We now discuss the requirements about free variables in well-formed specifications as defined in Def. 5.6. In scoped invariants, A may only mention variables introduced by the quantifier, \bar{x} . In method specifications, the precondition, $x : C' \wedge A$, may only mention the receiver, `this`, the formal parameters, \bar{y} , and the explicitly introduced variables, \bar{x} ; it may *not* mention the result `res`. The postcondition, A' , may mention these variables, and in addition, may mention the result, `res`. The mid-condition, A'' is about a state which has at least one more frame than the current method's, and therefore it may not mention `this`, nor \bar{y} , nor `res`.

E.2 Expressiveness

We argue the expressiveness of our approach by comparing with example specifications proposed in [34, 74, 100].

E.2.1 The DOM. This is the motivating example in [34], dealing with a tree of DOM nodes: Access to a DOM node gives access to all its `parent` and `children` nodes, with the ability to modify the node's `property` – where `parent`, `children` and `property` are fields in class `Node`. Since the top nodes of the tree usually contain privileged information, while the lower nodes contain less crucial third-party information, we must be able to limit access given to third parties to only the lower part of the DOM tree. We do this through a `Proxy` class, which has a field `node` pointing to a `Node`, and a field `height`, which restricts the range of `Nodes` which may be modified through the use of the particular `Proxy`. Namely, when you hold a `Proxy` you can modify the `property` of all the descendants of the `height`-th ancestors of the `node` of that particular `Proxy`. We say that `pr` has *modification-capabilities* on `nd`, where `pr` is a `Proxy` and `nd` is a `Node`, if the `pr.height`-th parent of the node at `pr.node` is an ancestor of `nd`.

We specify this property as follows:

$$S_{dom_1} \triangleq \forall nd : \text{DomNode}. \{ \forall pr : \text{Proxy}. [\text{may_modify}(pr, nd) \rightarrow \langle pr \rangle] \}$$

$$S_{dom_2} \triangleq \forall nd : \text{DomNode}, val : \text{PropertyValue}. \\ \{ \forall pr : \text{Proxy}. [\text{may_modify}(pr, nd) \rightarrow \langle pr \rangle] \wedge nd.property = val \}$$

where $\text{may_modify}(pr, nd) \triangleq \exists k. [nd.parent^k = pr.node.parent^{pr.height}]$

Note that S_{dom_2} is strictly stronger than S_{dom_1}

Mackay et al. [74] specify this as:

```
DOMSpec  $\triangleq$  from nd : Node  $\wedge$  nd.property = p to nd.property != p
  onlyIf  $\exists o. [ o : \text{ext1} \wedge$ 
    ( $\exists nd' : \text{Node}. [ \langle o \text{ access } nd' \rangle ] \vee$ 
     $\exists pr : \text{Proxy}, k : \mathbb{N}. [ \langle o \text{ access } pr \rangle \wedge nd.parent^k = pr.node.parent^{pr.height} ] ) ]$ 
```

`DomSpec` states that the `property` of a node can only change if some external object presently has access to a node of the DOM tree, or to some `Proxy` with modification-capabilities to the node that was modified. The assertion $\exists o. [o : \text{ext1} \wedge \langle o \text{ access } pr \rangle]$ is the contrapositive of our $\langle pr \rangle$, but is weaker than that, because it does not specify the frame from which o is accessible. Therefore, `DOMSpec` is a stronger requirement than S_{dom_1} .

E.2.2 DAO. The Decentralized Autonomous Organization (DAO) [24] is a well-known Ethereum contract allowing participants to invest funds. The DAO famously was exploited with a re-entrancy bug in 2016, and lost \$50M. Here we provide specifications that would have secured the DAO against such a bug.

$$S_{dao_1} \triangleq \forall d : \text{DAO}. \{ \forall p : \text{Participant}. [d.ether \geq d.balance(p)] \}$$

$$S_{dao_2} \triangleq \forall d : \text{DAO}. \{ d.ether \geq \sum_{p \in d.participants} d.balance(p) \}$$

The specifications above say the following:

S_{dao_1} guarantees that the DAO holds more ether than the balance of any of its participant's.

S_{dao_2} guarantees that that the DAO holds more ether than the sum of the balances held by DAO's participants.

S_{dao_2} is stronger than S_{dao_1} . They would both have precluded the DAO bug. Note that these specifications do not mention capabilities. They are, essentially, simple class invariants and could have been expressed with the techniques proposed already by [81]. The only difference is that S_{dao_1} and S_{dao_2} are two-state invariants, which means that we require that they are *preserved*, i.e.

if they hold in one (observable) state they have to hold in all successor states, while class invariants are one-state, which means they are required to hold in all (observable) states.¹⁵

We now compare with the specification given in [74]. `DAOSpec1` is similar to S_{dao_1} : it says that no participant's balance may ever exceed the ether remaining in DAO. It is, essentially, a one-state invariant.

```

DAOSpec1  $\triangleq$  from d : DAO  $\wedge$  p : Object
    to d.balance(p) > d.ether
    onlyIf false
  
```

`DAOSpec1`, similarly to S_{dao_1} , in that it enforces a class invariant of DAO, something that could be enforced by traditional specifications using class invariants.

[74] gives one more specification:

```

DAOSpec2  $\triangleq$  from d : DAO  $\wedge$  p : Object
    next d.balance(p) = m
    onlyIf <p calls d.repay(_)>  $\wedge$  m = 0  $\vee$  <p calls d.join(m)>  $\vee$  d.balance(p) = m
  
```

`DAOSpec2` states that if after some single step of execution, a participant's balance is m , then either

- (a) this occurred as a result of joining the DAO with an initial investment of m ,
- (b) the balance is 0 and they've just withdrawn their funds, or
- (c) the balance was m to begin with

E.2.3 ERC20. The ERC20 [112] is a widely used token standard describing the basic functionality of any Ethereum-based token contract. This functionality includes issuing tokens, keeping track of tokens belonging to participants, and the transfer of tokens between participants. Tokens may only be transferred if there are sufficient tokens in the participant's account, and if either they (using the `transfer` method) or someone authorised by the participant (using the `transferFrom` method) initiated the transfer.

For an $e : \text{ERC20}$, the term $e.balance(p)$ indicates the number of tokens in participant p 's account at e . The assertion $e.allowed(p, p')$ expresses that participant p has been authorised to spend moneys from p 's account at e .

The security model in Solidity is not based on having access to a capability, but on who the caller of a method is. Namely, Solidity supports the construct `sender` which indicates the identity of the caller. Therefore, for Solidity, we adapt our approach in two significant ways: we change the meaning of $\langle e \rangle$ to express that e did not make a method call. Moreover, we introduce a new, slightly modified form of two state invariants of the form $\forall x : \overline{C}. \{A\}. \{A'\}$ which expresses that any execution which satisfies A , will preserve A' .

We specify the guarantees of ERC20 as follows:

```

 $\S$ 
 $S_{erc\_2} \triangleq \forall e : \text{ERC20}, p, p' : \text{Participant}, n : \mathbb{N}. \\
    \{ \forall p'. [ (e.allowed(p', p) \rightarrow \langle p' \rangle) ] \}. \{ e.balance(b) = n \}$ 
 $S_{erc\_3} \triangleq \forall e : \text{ERC20}, p, p' : \text{Participant}. \\
    \{ \forall p'. [ (e.allowed(p', p) \rightarrow \langle p' \rangle) ] \}. \{ \neg(e.allowed(p'', p)) \}$ 
  
```

¹⁵This should have been explained somewhere earlier.

The specifications above say the following:

S_{erc_1} guarantees that the the owner of an account is always authorized on that account – this specification is expressed using the original version of two-state invariants.

S_{erc_2} guarantees that any execution which does not contain calls from a participant p' authorized on p 's account will not affect the balance of e 's account. Namely, if the execution starts in a state in which $e.balance(b) = n$, it will lead to a state where $e.balance(b) = n$ also holds.

S_{erc_3} guarantees that any execution which does not contain calls from a participant p' authorized on p 's account will not affect who else is authorized on that account. That is, if the execution starts in a state in which $\neg(e.allowed(p'', p))$, it will lead to a state where $\neg(e.allowed(p'', p))$ also holds.

We compare with the specifications given in [74]: Firstly, `ERC20Spec1` says that if the balance of a participant's account is ever reduced by some amount m , then that must have occurred as a result of a call to the `transfer` method with amount m by the participant, or the `transferFrom` method with the amount m by some other participant.

```

ERC20Spec1  $\triangleq$  from  $e : \text{ERC20} \wedge e.balance(p) = m + m' \wedge m > 0$ 
  next  $e.balance(p) = m'$ 
  onlyIf  $\exists p' p''. [\langle p' \text{ calls } e.transfer(p, m) \rangle \vee$ 
     $e.allowed(p, p'') \geq m \wedge \langle p'' \text{ calls } e.transferFrom(p', m) \rangle]$ 

```

Secondly, `ERC20Spec2` specifies under what circumstances some participant p' is authorized to spend m tokens on behalf of p : either p approved p' , p' was previously authorized, or p' was authorized for some amount $m + m'$, and spent m' .

```

ERC20Spec2  $\triangleq$  from  $e : \text{ERC20} \wedge p : \text{Object} \wedge p' : \text{Object} \wedge m : \text{Nat}$ 
  next  $e.allowed(p, p') = m$ 
  onlyIf  $\langle p \text{ calls } e.approve(p', m) \rangle \vee$ 
     $(e.allowed(p, p') = m \wedge$ 
       $\neg (\langle p' \text{ calls } e.transferFrom(p, \_) \rangle \vee$ 
         $\langle p \text{ calls } e.allowed(p, \_) \rangle)) \vee$ 
     $\exists p''. [e.allowed(p, p') = m + m' \wedge \langle p' \text{ calls } e.transferFrom(p'', m') \rangle]$ 

```

`ERC20Spec1` is related to S_{erc_2} . Note that `ERC20Spec1` is more API-specific, as it expresses the precise methods which caused the modification of the balance.

E.2.4 Wasm, Iris, and the stack. In [100], they consider inter-language safety for Wasm. They develop Iris-Wasm, a mechanized higher-order separation logic mechanized in Coq and the Iris framework. Using Iris-Wasm, with the aim to specify and verify individual modules separately, and then compose them modularly in a simple host language featuring the core operations of the WebAssembly JavaScript Interface. They develop a logical relation that enforces robust safety: unknown, adversarial code can only affect other modules through the functions that they explicitly export. They do not offer however a logic to deal with the effects of external calls.

As a running example, they use a `stack` module, which is an array of values, and exports functions to inspect the stack contents or modify its contents. Such a setting can be expressed in our language through a `stack` and a `modifier` capability. Assuming a predicate $Contents(stack, i, v)$, which expresses that the contents of `stack` at index i is v , we can specify the stack through

$$S_{stack} \triangleq \forall s : \text{Stack}, i : \mathbb{N}, v : \text{Value}. \{ \langle s.\text{modifier} \rangle \wedge \text{Contents}(s, i, v) \}$$

In that work, they provide a tailor-made proof that indeed, when the stack makes an external call, passing only the inspect-capability, the contents will not change. However, because the language is essentially functional, they do not consider the possibility that the external call might already have stored the modifier capability. Moreover, the proof does not make use of a Hoare logic.

E.2.5 Sealer-Unsealer pattern. The sealer-unsealer pattern, proposed by Morris Jr. [90], is a security pattern to enforce data abstraction while interoperating with untrusted code. He proposes a function `makeSeal` which generating pairs of functions (`seal`, `unseal`), such that `seal` takes a value v and returns a low-integrity value v' . The function `unseal` when given v' will return v . But there is no other way to obtain v out of v' except through the use of the `unsealer`. Thus, v' can securely be shared with untrusted code. This pattern has been studied by Swasey et al. [111].

We formulate this pattern here. As we are working with an object oriented rather than a functional language, we assume the existence of a class `DynamicSealer` with two methods, `seal`, and `unseal`. And we define a predicate $\text{Sealed}(v, v', us)$ to express that v has been sealed into v' and can be unsealed using us .

Then, the scoped invariants

$$S_{sealer_1} \triangleq \forall v, v', us : \text{Object}. \{ \langle us \rangle \wedge \text{Sealed}(v, v', us) \}$$

$$S_{sealer_2} \triangleq \forall v, v', us : \text{Object}. \{ \langle v \rangle \wedge \langle us \rangle \wedge \text{Sealed}(v, v', us) \}$$

expresses that the unsealer is not leaked to external code (S_{sealer_1}), and that if the external world has no access to the high-integrity value v nor to the its unsealer us , then it will not get access to the value (S_{sealer_2}).

F APPENDIX TO SECTION 6

F.1 Preliminaries: Specification Lookup, Renamings, Underlying Hoare Logic

Definition F.1 is broken down as follows: $S_1 \stackrel{\text{txt}}{\leq} S_2$ says that S_1 is textually included in S_2 ; $S \sim S'$ says that S is a safe renaming of S' ; $\vdash M : S$ says that S is a safe renaming of one of the specifications given for M .

In particular, a safe renaming of $\forall \bar{x} : \bar{C}. \{A\}$ can replace any of the variables \bar{x} . A safe renaming of $\{A_1\} p D :: m(\bar{y} : \bar{D}) \{A_2\} \parallel \{A_3\}$ can replace the formal parameters (\bar{y}) by actual parameters (\bar{y}') but requires the actual parameters not to include `this`, or `res`, (i.e. $\text{this}, \text{res} \notin \bar{y}'$). – Moreover, it can replace the free variables which do not overlap with the formal parameters or the receiver ($\bar{x} = Fv(A_1) \setminus \{\bar{y}, \text{this}\}$).

Definition F.1. For a module M and a specification S , we define:

- $S_1 \stackrel{\text{txt}}{\leq} S_2 \triangleq S_1 \stackrel{\text{txt}}{=} S_2$, or $S_2 \stackrel{\text{txt}}{=} S_1 \wedge S_3$, or $S_2 \stackrel{\text{txt}}{=} S_3 \wedge S_1$, or $S_2 \stackrel{\text{txt}}{=} S_3 \wedge S_1 \wedge S_4$ for some S_3, S_4 .
- $S \sim S'$ is defined by cases
 - $\forall \bar{x} : \bar{C}. \{A\} \sim \forall \bar{x}' : \bar{C}. \{A'[\bar{x}'/\bar{x}]\}$
 - $\{A_1\} p D :: m(\bar{y} : \bar{D}) \{A_2\} \parallel \{A_3\} \sim \{A'_1\} p D :: m(\bar{y}' : \bar{D}) \{A'_2\} \parallel \{A'_3\}$
 $\triangleq A_1 = A'_1[\bar{y}/\bar{y}'][\bar{x}/\bar{x}'], A_2 = A'_2[\bar{y}/\bar{y}'][\bar{x}/\bar{x}'], A_3 = A'_3[\bar{y}/\bar{y}'][\bar{x}/\bar{x}'], \wedge$
 $\text{this}, \text{res} \notin \bar{y}', \bar{x} = Fv(A_1) \setminus \{\bar{y}, \text{this}\}$
- $\vdash M : S \triangleq \exists S'. [S' \stackrel{\text{txt}}{\leq} \mathcal{Spec}(M) \wedge S' \sim S]$

The restriction on renamings of method specifications that the actual parameters should not to include `this` or `res` is necessary because `this` and `res` denote different objects from the point of the caller than from the point of the callee. It means that we are not able to verify a method call whose actual parameters include `this` or `res`. This is not a serious restriction: we can encode any such method call by preceding it with assignments to fresh local variables, $\text{this}' := \text{this}$, and $\text{res}' := \text{res}$, and using this' and res' in the call.

Example F.2. The specification from Example 5.5 can be renamed as

$$S_{9r} \triangleq \{a1 : \text{Account}, a2 : \text{Account} \wedge \langle a1 \rangle \wedge \langle a2.\text{key} \rangle\}$$

$$\text{public Account} :: \text{set}(\text{nKey} : \text{Key})$$

$$\{\langle a1 \rangle \wedge \langle a2.\text{key} \rangle\} \parallel \{\langle a1 \rangle \wedge \langle a2.\text{key} \rangle\}$$

Axiom F.3. Assume Hoare logic with judgements $M \vdash_{ul} \{A\} \text{stmt} \{A'\}$, with $Stbl(A)$, $Stbl(A')$.

F.2 Types

The rules in Fig. 15 allow triples to talk about the types Rule TYPES-1 promises that types of local variables do not change. Rule TYPES-2 generalizes TYPES-1 to any statement, provided that there already exists a triple for that statement.

$$\begin{array}{c} \text{TYPES-1} \\ \text{stmt contains no method call} \quad \text{stmt contains no assignment to } x \\ \hline M \vdash \{x : C\} \text{stmt} \{x : C\} \\ \\ \text{TYPES-2} \\ \hline M \vdash \{A\} s \{A'\} \parallel \{A''\} \\ \hline M \vdash \{x : C \wedge A\} s \{x : C \wedge A'\} \parallel \{A''\} \end{array}$$

Fig. 15. Types

In TYPES-1 we restricted to statements which do not contain method calls in order to make lemma F.5 valid.

F.3 Second Phase - more

in Fig. 16, we extend the Hoare Quadruples Logic with substructural rules, rules for conditionals, case analysis, and a contradiction rule. For the conditionals we assume the obvious operational semantics, but do not define it in this paper

$$\begin{array}{c}
 \text{[COMBINE]} \\
 \frac{M \vdash \{A_1\} s \{A_2\} \parallel \{A\} \quad M \vdash \{A_3\} s \{A_4\} \parallel \{A\}}{M \vdash \{A_1 \wedge A_3\} s \{A_2 \wedge A_4\} \parallel \{A\}} \\
 \text{[SEQU]} \\
 \frac{M \vdash \{A_1\} s_1 \{A_2\} \parallel \{A\} \quad M \vdash \{A_2\} s_2 \{A_3\} \parallel \{A\}}{M \vdash \{A_1\} s_1; s_2 \{A_3\} \parallel \{A\}} \\
 \text{[CONSEQU]} \\
 \frac{M \vdash \{A_4\} s \{A_5\} \parallel \{A_6\} \quad M \vdash A_1 \rightarrow A_4 \quad M \vdash A_5 \rightarrow A_2 \quad M \vdash A_6 \rightarrow A_3}{M \vdash \{A_1\} s \{A_2\} \parallel \{A_3\}} \\
 \text{[IF_RULE]} \\
 \frac{M \vdash \{A \wedge \text{Cond}\} stmt_1 \{A'\} \parallel \{A''\} \quad M \vdash \{A \wedge \neg \text{Cond}\} stmt_2 \{A'\} \parallel \{A''\}}{M \vdash \{A\} \text{ if } \text{Cond} \text{ then } stmt_1 \text{ else } stmt_2 \{A'\} \parallel \{A''\}} \\
 \text{[ABSURD]} \\
 \frac{M \vdash \{false\} stmt \{A'\} \parallel \{A''\}}{M \vdash \{A \wedge \neg A\} stmt \{A'\} \parallel \{A''\}} \\
 \text{[CASES]} \\
 \frac{M \vdash \{A \wedge A_1\} stmt \{A'\} \parallel \{A''\} \quad M \vdash \{A \wedge A_2\} stmt \{A'\} \parallel \{A''\}}{M \vdash \{A \wedge (A_1 \vee A_2)\} stmt \{A'\} \parallel \{A''\}}
 \end{array}$$

Fig. 16. Hoare Quadruples - substructural rules, and conditionals

F.4 Extend the semantics and Hoare logic to accommodate scalars and conditionals

We extend the notion of protection to also allow it to apply to scalars.

Definition F.4 (Satisfaction of Assertions – Protected From). extending the definition of Def 4.4.

We use α to range over addresses, β to range over scalars, and γ to range over addresses or scalars.

We define $M, \sigma \models \langle \gamma \rangle \leftarrow \times \gamma_o$ as:

- (1) $M, \sigma \models \langle \alpha \rangle \leftarrow \times \alpha_o \triangleq$
 - $\alpha \neq \alpha_o$, and
 - $\forall n \in \mathbb{N}. \forall f_1, \dots, f_n. [\alpha_o.f_1 \dots f_n]_\sigma = \alpha \implies M, \sigma \models [\alpha_o.f_1 \dots f_{n-1}]_\sigma : C \wedge C \in M$
- (2) $M, \sigma \models \langle \gamma \rangle \leftarrow \times \beta_o \triangleq \text{true}$
- (3) $M, \sigma \models \langle \beta \rangle \leftarrow \times \alpha_o \triangleq \text{false}$
- (4) $M, \sigma \models \langle e \rangle \leftarrow \times e_o \triangleq$

$$\exists \gamma, \gamma_o. [M, \sigma, e \hookrightarrow \gamma \wedge M, \sigma, e_o \hookrightarrow \gamma_o \wedge M, \sigma \models \langle \gamma \rangle \leftarrow \times \gamma_o]$$

The definition from above gives rise to further cases of protection; we supplement the triples from Fig. 7 with some further inference rules, given in Fig. 17. Namely, any expression e is protected from a scalar (rules PROT-IN, PROT-BOOL and PROT-STR). Moreover, if starting at some e_o and following any sequence of fields \bar{f} we reach internal objects or scalars (*i.e.* never reach an external object), then any e is protected from e_o (rule PROT_INTL).

Lemma F.5. If $M \vdash \{A\} stmt \{A'\}$, then $stmt$ contains no method calls.

$$\begin{array}{c}
M \vdash e_o : \text{int} \rightarrow \langle e \rangle \leftarrow e_o \quad [\text{PROT-INT}] \qquad M \vdash e_o : \text{bool} \rightarrow \langle e \rangle \leftarrow e_o \quad [\text{PROT-BOOL}] \\
M \vdash e_o : \text{str} \rightarrow \langle e \rangle \leftarrow e_o \quad [\text{PROT-STR}] \qquad M \vdash \langle e \rangle \leftarrow e_o \wedge e : \text{intl} \rightarrow e \neq e' \quad [\text{PROT-NEQ}] \\
\frac{M \vdash A \rightarrow \forall \bar{f}. [e_o.\bar{f} : \text{intl} \vee e_o.\bar{f} : \text{int} \vee e_o.\bar{f} : \text{bool} \vee e_o.\bar{f} : \text{str}]}{M \vdash A \rightarrow \langle e \rangle \leftarrow e_o} \quad [\text{PROT-INTL}]
\end{array}$$

Fig. 17. Protection for Scalar and Internal Types

PROOF. By induction on the rules in Fig. 7.

□

F.5 Adaptation

We now discuss the proof of Lemma 6.2.

Proof of lemma 6.2, part 1

To Show: $\text{Stbl}(A \rightarrow \neg(y_0, \bar{y}))$

By structural induction on A .

End Proof

For parts 2, 3, and 4, we first prove the following auxiliary lemma:

Auxiliary Lemma F.6. For all $\alpha, \bar{\phi}_1, \bar{\phi}_2, \bar{\phi}_2, \phi$ and χ

$$(L1) \quad M, (\bar{\phi}_1, \chi) \models \langle \alpha \rangle \leftarrow \text{Rng}(\phi) \implies M, (\bar{\phi}_2 \cdot \phi, \chi) \models \langle \alpha \rangle$$

$$(L2) \quad M, (\bar{\phi}_1 \cdot \phi, \chi) \models \langle \alpha \rangle \wedge \text{extl} \implies M, (\bar{\phi}_2, \chi) \models \langle \alpha \rangle \leftarrow \text{Rng}(\phi)$$

$$(L3) \quad M, (\bar{\phi}_1 \cdot \phi_1, \chi) \models \langle \alpha \rangle \wedge \text{extl} \wedge \text{Rng}(\phi) \subseteq \text{Rng}(\phi_1) \implies M, (\bar{\phi}_2, \chi) \models \langle \alpha \rangle \leftarrow \text{Rng}(\phi)$$

PROOF.

We first prove (L1):

We define $\sigma_1 \triangleq (\bar{\phi}_1, \chi)$, and $\sigma_2 \triangleq (\bar{\phi}_2 \cdot \phi, \chi)$.

The above definitions imply that:

- (1) $\forall \alpha', \forall \bar{f}. [\lfloor \alpha'.\bar{f} \rfloor_{\sigma_1} = \lfloor \alpha'.\bar{f} \rfloor_{\sigma_2}]$
- (2) $\forall \alpha'. [\text{Rchbl}(\alpha', \sigma_1) = \text{Rchbl}(\alpha', \sigma_2)]$
- (3) $\text{LocRchbl}(\sigma_2) = \bigcup_{\alpha' \in \text{Rng}(\phi)} \text{Rchbl}(\alpha', \sigma_2)$.

We now assume that

$$(4) \quad M, \sigma_1 \models \langle \alpha \rangle \leftarrow \text{Rng}(\phi).$$

and want to show that

$$(A?) \quad M, \sigma_2 \models \langle \alpha \rangle$$

From (4) and by definitions, we obtain that

$$(5) \quad \forall \alpha' \in \text{Rng}(\phi). \forall \alpha'' \in \text{Rchbl}(\alpha', \sigma_1). \forall f. [M, \sigma_1 \models \alpha'' : \text{extl} \rightarrow \alpha''.f \neq \alpha], \quad \text{and also}$$

$$(6) \quad \alpha \notin \text{Rng}(\phi)$$

From (5) and (3) we obtain:

$$(7) \quad \forall \alpha' \in \text{LocRchbl}(\sigma_2). \forall f. [M, \sigma_1 \models \alpha' : \text{extl} \rightarrow \alpha'.f \neq \alpha]$$

From (7) and (1) and (2) we obtain:

$$(8) \quad \forall \alpha' \in \text{LocRchbl}(\sigma_2). \forall f. [M, \sigma_2 \models \alpha' : \text{extl} \rightarrow \alpha'.f \neq \alpha]$$

From (8), by definitions, we obtain

$$(10) \quad M, \sigma_2 \models \langle \alpha \rangle$$

which is (A?).

This completes the proof of (L1).

We now prove (L2):

We define $\sigma_1 \triangleq (\overline{\phi_1} \cdot \phi, \chi)$, and $\sigma_2 \triangleq (\overline{\phi_2}, \chi)$.

The above definitions imply that:

- (1) $\forall \alpha', \forall \bar{f}. [\lfloor \alpha'.\bar{f} \rfloor_{\sigma_1} = \lfloor \alpha'.\bar{f} \rfloor_{\sigma_2}]$
- (2) $\forall \alpha'. [\text{Rchbl}(\alpha', \sigma_1) = \text{Rchbl}(\alpha', \sigma_2)]$
- (3) $\text{LocRchbl}(\sigma_1) = \bigcup_{\alpha' \in \text{Rng}(\phi)} \text{Rchbl}(\alpha', \sigma_1)$.

We assume that

$$(4) \quad M, \sigma_1 \models \langle \alpha \rangle \wedge \text{extl}.$$

and want to show that

$$(A?) \quad M, \sigma_2 \models A \neg \text{Rng}(\phi).$$

From (4), and unfolding the definitions, we obtain:

- (5) $\forall \alpha' \in \text{LocRchbl}(\sigma_1). \forall f. [M, \sigma_1 \models \alpha' : \text{extl} \rightarrow \alpha'.f \neq \alpha], \quad \text{and}$
- (6) $\forall \alpha' \in \text{Rng}(\phi). [\alpha' \neq \alpha].$

From(5), and using (3) and (2) we obtain:

$$(7) \quad \forall \alpha' \in \text{Rng}(\phi). \forall \alpha'' \in \text{Rchbl}(\alpha', \sigma_2). \forall f. [M, \sigma_2 \models \alpha'' : \text{extl} \rightarrow \alpha''.f \neq \alpha]$$

From (5) and (7) and by definitions, we obtain

$$(8) \quad \forall \alpha' \in \text{Rng}(\phi). [\models \alpha \langle \alpha \rangle \leftarrow \alpha'].$$

From (8) and definitions we obtain (A?).

This completes the proof of (L2).

We now prove (L3):

We define $\sigma_1 \triangleq (\overline{\phi_1} \cdot \phi_1, \chi)$, and $\sigma_2 \triangleq (\overline{\phi_2}, \chi)$.

The above definitions imply that:

- (1) $\forall \alpha', \forall \bar{f}. [\lfloor \alpha'.\bar{f} \rfloor_{\sigma_1} = \lfloor \alpha'.\bar{f} \rfloor_{\sigma_2}]$
- (2) $\forall \alpha'. [\text{Rchbl}(\alpha', \sigma_1) = \text{Rchbl}(\alpha', \sigma_2)]$
- (3) $\text{LocRchbl}(\sigma_1) = \bigcup_{\alpha' \in \text{Rng}(\phi_1)} \text{Rchbl}(\alpha', \sigma_1)$.

We assume that

$$(4a) \quad M, \sigma_1 \models \langle \alpha \rangle \wedge \text{extl}, \quad \text{and} \quad (4b) \quad \text{Rng}(\phi) \subseteq \text{Rng}(\phi_1)$$

We want to show that

$$(A?) \quad M, \sigma_2 \models A \neg \text{Rng}(\phi).$$

From (4a), and unfolding the definitions, we obtain:

- (5) $\forall \alpha' \in \text{LocRchbl}(\sigma_1). \forall f. [M, \sigma_1 \models \alpha' : \text{extl} \rightarrow \alpha'.f \neq \alpha], \quad \text{and}$
- (6) $\forall \alpha' \in \text{Rng}(\phi_1). [\alpha' \neq \alpha].$

From(5), and (3) and (2) and (4b) we obtain:

$$(7) \quad \forall \alpha' \in \text{Rng}(\phi). \forall \alpha'' \in \text{Rchbl}(\alpha', \sigma_2). \forall f. [M, \sigma_2 \models \alpha'' : \text{extl} \rightarrow \alpha''.f \neq \alpha]$$

From(6), and (4b) we obtain:

$$(8) \quad \forall \alpha' \in \text{Rng}(\phi_1). [\alpha' \neq \alpha].$$

From (8) and definitions we obtain (A?).

This completes the proof of (L3).

□

Proof of lemma 6.2, part 2

To Show: $(*) \quad M, \sigma \models A \neg Rng(\phi) \implies M, \sigma \nabla \phi \models A$

By induction on the structure of A . For the case where A has the form $\langle \alpha.\bar{f} \rangle$, we use lemma F.6,(L1), taking $\bar{\phi}_1 = \bar{\phi}_2$, and $\sigma \triangleq (\bar{\phi}_1, \chi)$.

End Proof

Proof of lemma 6.2, part 3

To Show $(*) \quad M, \sigma \nabla \phi \models A \wedge \text{extl} \implies M, \sigma \models A \neg Rng(\phi)$

We apply induction on the structure of A . For the case where A has the form $\langle \alpha.\bar{f} \rangle$, we apply lemma F.6,(L2), using $\bar{\phi}_1 = \bar{\phi}_2$, and $\sigma \triangleq (\bar{\phi}_1, \chi)$.

End Proof

Proof of lemma 6.2, part 4

To Show: $(*) \quad M, \sigma \models A \wedge \text{extl} \wedge M \cdot \bar{M} \models \sigma \nabla \phi \implies M, \sigma \nabla \phi \models A \neg Rng(\phi)$

By induction on the structure of A . For the case where A has the form $\langle \alpha.\bar{f} \rangle$, we want to apply lemma F.6,(L3). We take σ to be $(\bar{\phi}_1 \cdot \phi_1, \chi)$, and $\bar{\phi}_2 = \bar{\phi}_1 \cdot \phi_1 \cdot \phi$. Moreover, $M \cdot \bar{M} \models \sigma \nabla \phi$ gives that $Rng(\phi) \subseteq LocRchbl(\sigma_2)$. Therefore, $(*)$ follows by application of lemma F.6,(L3).

End Proof

G APPENDIX TO SECTION 7 – SOUNDNESS OF THE HOARE LOGICS

G.1 Expectations

Axiom G.1. We require a sound logic of assertions ($M \vdash A$), and a sound Hoare logic, *i.e.* that for all $M, A, A', stmt$:

$$\begin{aligned} M \vdash A &\implies \forall \sigma. [M, \sigma \models A]. \\ M \vdash_{ul} \{A\} stmt \{A'\} &\implies M \models \{A\} stmt \{A'\} \end{aligned}$$

The expectation that $M \vdash_{ul} \{A\} stmt \{A'\}$ is sound is not onerous: since the assertions A and A' do not talk about protection, many Hoare logics from the literature could be taken.

On the other hand, in the logic $M \vdash A$ we want to allow the assertion A to talk about protection. Since protection is a novel concept, the literature offers no such logics. Nevertheless, such a logic can be constructed by extending and underlying assertion logic $M \vdash_{ul} A$ which does not talk about protection. We show such an extension in Fig 18.

$$\begin{array}{c} \text{[EXT-1]} \\ \frac{M \vdash_{ul} A}{M \vdash_{ul} A} \\ \text{[EXT-2]} \quad \frac{M \vdash A \rightarrow A'}{M \vdash (A \wedge \langle e \rangle \leftarrow \times e') \rightarrow (A' \wedge \langle e \rangle \leftarrow \times e')} \quad \text{[EXT-3]} \quad \frac{M \vdash A \rightarrow A'}{M \vdash (A \wedge \langle e \rangle) \rightarrow (A' \wedge \langle e \rangle)} \\ \text{[EXT-4]} \\ \frac{M \vdash ((A_1 \vee A_2) \wedge \langle e \rangle \leftarrow \times e') \leftrightarrow ((A_1 \wedge \langle e \rangle \leftarrow \times e') \vee (A_2 \wedge \langle e \rangle \leftarrow \times e'))}{M \vdash ((A_1 \vee A_2) \wedge \langle e \rangle) \leftrightarrow ((A_1 \wedge \langle e \rangle) \vee (A_2 \wedge \langle e \rangle))} \\ \text{[EXT-5]} \\ \frac{M \vdash ((A_1 \vee A_2) \wedge \langle e \rangle) \leftrightarrow ((A_1 \wedge \langle e \rangle) \vee (A_2 \wedge \langle e \rangle))}{M \vdash ((A_1 \vee A_2) \wedge \langle e \rangle) \leftrightarrow ((A_1 \wedge \langle e \rangle) \vee (A_2 \wedge \langle e \rangle))} \end{array}$$

Fig. 18. From $M \vdash_{ul} A$ to $M \vdash A$

The extension shown in in Fig. 18 preserves soundness of the logic:

Lemma G.2. Assume a logic \vdash_{ul} , such that

$$M \vdash_{ul} A \implies \forall \sigma. [M, \sigma \models A].$$

Extend this logic according to the rules in Fig. 18 and in Fig 17, and obtain $M \vdash A$. Then, we have:

$$M \vdash A \implies \forall \sigma. [M, \sigma \models A].$$

PROOF. By induction over the derivation that $M \vdash A$. □

Note that the rules in in Fig. 18 allow the derivation of $M \vdash A$, for which $Stb^+(A)$ does not hold – *e.g.* we can derive $M \vdash \langle e \rangle \rightarrow \langle e \rangle$ through application of rule EXT-3. However, this does not affect soundness of our logic – $Stb^+(_)$ is required only in specifications.

G.2 Deep satisfaction of assertions

Definition G.3. For a state σ , and a number $i \in \mathbb{N}$ with $i \leq |\sigma|$, module M , and assertions A, A' we define:

- $M, \sigma, k \models A \triangleq k \leq |\sigma| \wedge \forall i \in [k \dots |\sigma|]. [M, \sigma[i] \models A[\overline{[z]_\sigma/z}]]$ where $\bar{z} = Fv(A)$.

Remember the definition of $\sigma[k]$, which returns a new state whose top frame is the k -th frame from σ . Namely, $(\phi_1 \dots \phi_i \dots \phi_n, \chi)[i] \triangleq (\phi_1 \dots \phi_i, \chi)$

Lemma G.4. For a states σ, σ' , numbers $k, k' \in \mathbb{N}$, assertions A, A' , frame ϕ and variables \bar{z}, \bar{u} :

- (1) $M, \sigma, |\sigma| \models A \iff M, \sigma \models A$
- (2) $M, \sigma, k \models A \wedge k \leq k' \implies M, \sigma, k' \models A$
- (3) $M, \sigma \models A \wedge Stbl(A) \implies \forall k \leq |\sigma|. [M, \sigma, k \models A]$
- (4) $M \models A \rightarrow A' \implies \forall \sigma. \forall k \leq |\sigma|. [M, \sigma, k \models A \implies M, \sigma, k \models A']$

Proof Sketch

- (1) By unfolding and folding the definitions.
- (2) By unfolding and folding the definitions.
- (3) By induction on the definition of $Stbl(_)$.
- (4) By contradiction: Assume that there exists a σ , and a $k \leq |\sigma|$, such that

$$M, \sigma, k \models A \quad \text{and} \quad \neg(M, \sigma, k \models A')$$

The above implies that

$$\forall i \geq k. [M, \sigma[i] \models A[\overline{[z]_\sigma/z}]], \quad \text{and}$$

$$\exists j \geq k. [M, \sigma[j] \not\models A'[\overline{[z]_\sigma/z}]],$$

where $\bar{z} \triangleq Fv(A) \cup Fv(A')$.

Take $\sigma'' \triangleq \sigma[j]$. Then we have that

$$M, \sigma'' \models A[\overline{[z]_\sigma/z}], \text{ and } M, \sigma'' \not\models A'[\overline{[z]_\sigma/z}].$$

This contradicts $M \models A \rightarrow A'$.

End Proof Sketch

Finally, the following lemma allows us to combine shallow and Deep satisfaction:

Lemma G.5. For states σ, σ' , frame ϕ such that $\sigma' = \sigma \nabla \phi$, and for assertion A , such that $Fv(A) = \emptyset$:

- $M, \sigma, k \models A \wedge M, \sigma' \models A \iff M, \sigma', k \models A$

PROOF. By structural induction on A , and unfolding/folding the definitions. Using also lemma G.19 from later. \square

G.3 Shallow and Deep Semantics of Hoare tuples

Another example demonstrating that assertions at the end of a method execution might not hold after the call:

Example G.6 (Stb^+ not always preserved by Method Return). Assume state σ_a , such that $[this]_{\sigma_a} = o_1$, $[this.f]_{\sigma} = o_2$, $[x]_{\sigma} = o_3$, $[x.f]_{\sigma} = o_2$, and $[x.g]_{\sigma} = o_4$, where o_2 is external and all other objects are internal. We then have $\dots, \sigma_a \models \langle o_4 \rangle$. Assume the continuation of σ_a consists of a method $x.m()$. Then, upon entry to that method, when we push the new frame, we have state σ_b , which also satisfies $\dots, \sigma_b \models \langle o_4 \rangle$. Assume the body of m is $this.f.m1(this.g); this.f := this; this.g := this$, and the external method $m1$ stores in the receiver a reference to the argument. Then, at the end of method execution, and before popping the stack, we have state σ_c , which also satisfies $\dots, \sigma_c \models \langle o_4 \rangle$. However, after we pop the stack, we obtain σ_d , for which $\dots, \sigma_d \not\models \langle o_4 \rangle$.

Definition G.7 (Deep Satisfaction of Quadruples by States). For modules \bar{M} , M , state σ , and assertions A, A' and A''

- $\bar{M}; M \models_{\text{deep}} \{A\} \sigma \{A'\} \parallel \{A''\} \triangleq$
 $\forall k, \bar{z}, \sigma', \sigma''. [M, \sigma, k \models A \implies$
 $[M \cdot \bar{M}; \sigma \rightsquigarrow_{\text{fin}}^* \sigma' \implies M, \sigma', k \models A'] \wedge$
 $[M \cdot \bar{M}; \sigma \rightsquigarrow^* \sigma'' \implies M, \sigma'', k \models (\text{ext l} \rightarrow A''[\bar{z}]_{\sigma/z})]$
 $]$
 where $\bar{z} = Fv(A)$

Lemma G.8. For all $M, \bar{M} A, A', A''$ and σ :

- $\bar{M}; M \models_{\text{deep}} \{A\} \sigma \{A'\} \parallel \{A''\} \implies \bar{M}; M \models \{A\} \sigma \{A'\} \parallel \{A''\}$

We define the *meaning* of our Hoare triples, $\{A\} \text{stmt} \{A'\}$, in the usual way, i.e. that execution of *stmt* in a state that satisfies A leads to a state which satisfies A' . In addition to that, Hoare quadruples, $\{A\} \text{stmt} \{A'\} \parallel \{A''\}$, promise that any external future states scoped by σ will satisfy A'' . We give both a weak and a shallow version of the semantics

Definition G.9 (Deep Semantics of Hoare triples). For modules M , and assertions A, A' we define:

- $M \models \{A\} \text{stmt} \{A'\} \triangleq$
 $\forall \bar{M}. \forall \sigma. [\sigma.\text{cont} \stackrel{\text{txt}}{=} \text{stmt} \implies \bar{M}; M \models \{A\} \sigma \{A'\} \parallel \{true\}]$
- $M \models \{A\} \text{stmt} \{A'\} \parallel \{A''\} \triangleq$
 $\forall \bar{M}. \forall \sigma. [\sigma.\text{cont} \stackrel{\text{txt}}{=} \text{stmt} \implies \bar{M}; M \models \{A\} \sigma \{A'\} \parallel \{A''\}]$
- $M \models_{\text{deep}} \{A\} \text{stmt} \{A'\} \triangleq$
 $\forall \bar{M}. \forall \sigma. [\sigma.\text{cont} \stackrel{\text{txt}}{=} \text{stmt} \implies \bar{M}; M \models_{\text{deep}} \{A\} \sigma \{A'\} \parallel \{true\}]$
- $M \models_{\text{deep}} \{A\} \text{stmt} \{A'\} \parallel \{A''\} \triangleq$
 $\forall \bar{M}. \forall \sigma. [\sigma.\text{cont} \stackrel{\text{txt}}{=} \text{stmt} \implies \bar{M}; M \models_{\text{deep}} \{A\} \sigma \{A'\} \parallel \{A''\}]$

Lemma G.10 (Deep vs Shallow Semantics of Quadruples). For all M, A, A' , and *stmt*:

- $M \models_{\text{deep}} \{A\} \text{stmt} \{A'\} \parallel \{A''\} \implies M \models \{A\} \text{stmt} \{A'\} \parallel \{A''\}$

PROOF. By unfolding and folding the definitions □

G.4 Deep satisfaction of specifications

We now give a Deep meaning to specifications:

Definition G.11 (Deep Semantics of Specifications). We define $M \models_{\text{deep}} S$ by cases:

- (1) $M \models_{\text{deep}} \forall \bar{x} : \bar{C}. \{A\} \triangleq \forall \sigma. [M \models_{\text{deep}} \{\text{ext l} \wedge \bar{x} : \bar{C} \wedge A\} \sigma \{A\} \parallel \{A\}]$
- (2) $M \models_{\text{deep}} \{A_1\} p D :: m(\bar{y} : \bar{D}) \{A_2\} \parallel \{A_3\} \triangleq$
 $\forall y_0, \bar{y}, \sigma [\sigma.\text{cont} \stackrel{\text{txt}}{=} u := y_0.m(y_1, ..y_n) \implies M \models_{\text{deep}} \{A'_1\} \sigma \{A'_2\} \parallel \{A'_3\}]$
 where
 $A'_1 \triangleq y_0 : D, \bar{y} : \bar{D} \wedge A[y_0/\text{this}], A'_2 \triangleq A_2[u/\text{res}, y_0/\text{this}], A'_3 \triangleq A_3[y_0/\text{this}]$
- (3) $M \models_{\text{deep}} S \wedge S' \triangleq M \models_{\text{deep}} S \wedge M \models_{\text{deep}} S'$

Lemma G.12 (Deep vs Shallow Semantics of Quadruples). For all M, S :

- $M \models_{\text{deep}} S \implies M \models S$

G.5 Soundness of the Hoare Triples Logic

Auxiliary Lemma G.13. For any module M , assertions A, A' and A'' , such that $\text{Stb}^+(A)$, and $\text{Stb}^+(A')$, and a statement *stmt* which does not contain any method calls:

$$M \models_{\text{deep}} \{A\} \text{stmt} \{A'\} \implies M \models_{\text{deep}} \{A\} \text{stmt} \{A'\} \parallel \{A''\}$$

PROOF. □

G.5.1 Lemmas about protection.

Definition G.14. $LocRchbl(\sigma, k) \triangleq \{ \alpha \mid \exists i. [k \leq i \leq |\sigma| \wedge \alpha \in LocRchbl(\sigma[i])] \}$

Lemma G.15 guarantees that program execution reduces the locally reachable objects, unless it allocates new ones. That is, any objects locally reachable in the k -th frame of the new state (σ'), are either new, or were locally reachable in the k -th frame of the previous state (σ).

Lemma G.15. For all σ, σ' , and α , where $\models \sigma$, and where $k \leq |\sigma|$:

- $\overline{M} \cdot M; \sigma \rightsquigarrow \sigma' \implies LocRchbl(\sigma', k) \cap \sigma \subseteq LocRchbl(\sigma, k)$
- $\overline{M} \cdot M; \sigma \rightsquigarrow^* \sigma' \implies LocRchbl(\sigma', k) \cap \sigma \subseteq LocRchbl(\sigma, k)$

PROOF.

- If the step is a method call, then the assertion follows by construction. If the step is a local execution in a method, we proceed by case analysis. If it is an assignment to a local variable, then $\forall k. [LocRchbl(\sigma', k) = LocRchbl(\sigma, k)]$. If the step is the creation of a new object, then the assertion holds by construction. If it is a field assignment, say, $\sigma' = \sigma[\alpha_1, f \mapsto \alpha_2]$, then we have that $\alpha_1, \alpha_2 \in LocRchbl(\sigma, |\sigma|)$. And therefore, by Lemma B.3, we also have that $\alpha_1, \alpha_2 \in LocRchbl(\sigma, k)$. All locally reachable objects in σ' were either already reachable in σ or reachable through α_2 . Therefore, we also have that $LocRchbl(\sigma', k) \subseteq LocRchbl(\sigma, k)$. And by definition of $_;$, it is not a method return.
- By induction on the number of steps in $\overline{M} \cdot M; \sigma \rightsquigarrow^* \sigma'$. For the steps that correspond to method calls, the assertion follows by construction. For the steps that correspond to local execution in a method, the assertion follows from the bullet above. For the steps that correspond to method returns, the assertion follows by lemma B.3.

□

Lemma G.16 guarantees that any change to the contents of an external object can only happen during execution of an external method.

Lemma G.16. For all σ, σ' :

- $\overline{M} \cdot M; \sigma \rightsquigarrow \sigma' \wedge \sigma \models \alpha : \text{extl} \wedge [\alpha.f]_\sigma \neq [\alpha.f]_{\sigma'} \implies M, \sigma \models \text{extl}$

PROOF. Through inspection of the operational semantics in Fig. 5, and in particular rule WRITE.

□

Lemma G.17 guarantees that internal code which does not include method calls preserves absolute protection. It is used in the proof of soundness of the inference rule PROT-1.

Lemma G.17. For all σ, σ' , and α :

- $M, \sigma, k \models \langle \alpha \rangle \wedge M, \sigma \models \text{intl} \wedge \sigma.\text{cont}$ contains no method calls $\wedge \overline{M} \cdot M; \sigma \rightsquigarrow \sigma' \implies M, \sigma', k \models \langle \alpha \rangle$
- $M, \sigma, k \models \langle \alpha \rangle \wedge M, \sigma \models \text{intl} \wedge \sigma.\text{cont}$ contains no method calls $\wedge \overline{M} \cdot M; \sigma \rightsquigarrow^* \sigma' \implies M, \sigma', k \models \langle \alpha \rangle$

PROOF.

- Because $\sigma.\text{cont}$ contains no method calls, we also have that $|\sigma'| = |\sigma|$. Let us take $m = |\sigma|$. We continue by contradiction. Assume that $M, \sigma, k \models \langle \alpha \rangle$ and $M, \sigma, k \not\models \langle \alpha \rangle$

Then:

(*) $\forall f. \forall i \in [k..m]. \forall \alpha_o \in LocRchbl(\sigma, i). [M, \sigma \models \alpha_o : \text{extl} \implies [\alpha_o.f]_\sigma \neq \alpha \wedge \alpha_o \neq \alpha]$.

(**) $\exists f. \exists j \in [k..m]. \exists \alpha_o \in LocRchbl(\sigma', j). [M, \sigma' \models \alpha_o : \text{extl} \wedge [\alpha_o.f]_{\sigma'} = \alpha \vee \alpha_o = \alpha]$

We proceed by cases

1st Case $\alpha_o \notin \sigma$, i.e. α_o is a new object. Then, by our operational semantics, it cannot have a field pointing to an already existing object (α), nor can it be equal with α . Contradiction.

2nd Case $\alpha_o \in \sigma$. Then, by Lemma G.15, we obtain that $\alpha_o \in \text{LocRchbl}(\sigma, j)$. Therefore, using (*), we obtain that $\lfloor \alpha_o.f \rfloor_\sigma \neq \alpha$, and therefore $\lfloor \alpha_o.f \rfloor_\sigma \neq \lfloor \alpha_o.f \rfloor_{\sigma'}$. By lemma G.16, we obtain $M, \sigma \models \text{extl}$. Contradiction!

- By induction on the number of steps, and using the bullet above.

□

Lemma G.18. For all σ, σ' , and α :

- $M, \sigma \models \langle \alpha \rangle \leftarrow \alpha_o \wedge \sigma.\text{heap} = \sigma'.\text{heap} \implies M, \sigma' \models \langle \alpha \rangle \leftarrow \alpha_o$

PROOF. By unfolding and folding the definitions.

□

Lemma G.19. For all σ , and $\alpha, \alpha_o, \alpha_1, \alpha_2$:

- $M, \sigma \models \langle \alpha \rangle \leftarrow \alpha_o \wedge M, \sigma \models \langle \alpha \rangle \leftarrow \alpha_1 \implies M, \sigma[\alpha_2, f \mapsto \alpha_1] \models \langle \alpha \rangle \leftarrow \alpha_o$

Definition G.20. • $M, \sigma \models e : \text{intl}^* \triangleq \forall \bar{f}. [M, \sigma \models e.\bar{f} : \text{intl}]$

Lemma G.21. For all σ , and α_o and α :

- $M, \sigma \models \alpha_o : \text{intl}^* \implies M, \sigma \models \langle \alpha \rangle \leftarrow \alpha_o$

Proof Sketch Theorem 7.2 The proof goes by case analysis over the rules from Fig. 7 applied to obtain $M \vdash \{A\} \text{stmt} \{A'\}$:

EMBED_UL By soundness of the underlying Hoare logic (axiom G.1), we obtain that $M \models \{A\} \text{stmt} \{A'\}$.

By axiom F.3 we also obtain that $\text{Stbl}(A)$ and $\text{Stbl}(A')$. This, together with Lemma G.4, part 3, gives us that $M \models_{\text{deep}} \{A\} \text{stmt} \{A'\}$. By the assumption of EXTEND, *stmt* does not contain any method call. Rest follows by lemma G.13.

PROT-NEW By operational semantics, no field of another object will point to u , and therefore u is protected, and protected from all variables x .

PROT-1 by Lemma G.17. The rule premise $M \vdash \{z = e\} \text{stmt} \{z = e\}$ allows us to consider addresses, α , rather than expressions, e .

PROT-2 by Lemma G.18. The rule premise $M \vdash \{z = e \wedge z = e'\} \text{stmt} \{z = e \wedge z = e'\}$ allows us to consider addresses α, α' rather than expressions e, e' .

PROT-3 also by Lemma G.18. Namely, the rule does not change, and $y.f$ in the old state has the same value as x in the new state.

PROT-4 by Lemma G.19.

TYPES-1 Follows from type system, the assumption of TYPES-1 and lemma G.13.

End Proof Sketch

G.6 Well-founded ordering

Definition G.22. For a module M , and modules \bar{M} , we define a measure, $\llbracket A, \sigma, A', A'' \rrbracket_{M, \bar{M}}$, and based on it, a well founded ordering $(A_1, \sigma_1, A_2, A_3) \ll_{M, \bar{M}} (A_4, \sigma_2, A_5, A_6)$ as follows:

- $\llbracket A, \sigma, A', A'' \rrbracket_{M, \bar{M}} \triangleq (m, n)$, where
 - m is the minimal number of execution steps so that $M \cdot \bar{M}; \sigma \rightsquigarrow_{fin}^* \sigma'$ for some σ' , and ∞ otherwise.
 - n is minimal depth of all proofs of $M \vdash \{A\} \sigma.\text{cont} \{A'\} \parallel \{A''\}$.
- $(m, n) \ll (m', n') \triangleq m < m' \vee (m = m' \wedge n < n')$.
- $(A_1, \sigma_1, A_2, A_3) \ll_{M, \bar{M}} (A_4, \sigma_2, A_5, A_6) \triangleq \llbracket A_1, \sigma_1, A_2, A_3 \rrbracket_{M, \bar{M}} \ll \llbracket A_4, \sigma_2, A_5, A_6 \rrbracket_{M, \bar{M}}$

Lemma G.23. For any modules M and \bar{M} , the relation $_ \ll_{M, \bar{M}} _$ is well-founded.

G.7 Public States, properties of executions consisting of several steps

We define a state to be public, if the currently executing method is public.

Definition G.24. We use the form $M, \sigma \models \text{pub}$ to express that the currently executing method is public.¹⁶ Note that pub is not part of the assertion language.

Auxiliary Lemma G.25 (Enclosed Terminating Executions). For modules \bar{M} , states $\sigma, \sigma', \sigma_1$:

- $\bar{M}; \sigma \rightsquigarrow_{fin}^* \sigma' \wedge \bar{M}; \sigma \rightsquigarrow^* \sigma_1 \implies \exists \sigma_2. [\bar{M}; \sigma_1 \rightsquigarrow_{fin}^* \sigma_2 \wedge (\bar{M}, \sigma); \sigma_2 \rightsquigarrow^* \sigma']$

Auxiliary Lemma G.26 (Executing sequences). For modules \bar{M} , statements s_1, s_2 , states $\sigma, \sigma', \sigma''$:

- $\sigma.\text{cont} = s_1; s_2 \wedge \bar{M}; \sigma \rightsquigarrow_{fin}^* \sigma' \wedge \bar{M}; \sigma \rightsquigarrow^* \sigma'' \implies \exists \sigma''. [\bar{M}; \sigma[\text{cont} \mapsto s_1] \rightsquigarrow_{fin}^* \sigma'' \wedge \bar{M}; \sigma''[\text{cont} \mapsto s_2] \rightsquigarrow_{fin}^* \sigma' \wedge [\bar{M}; \sigma[\text{cont} \mapsto s_1] \rightsquigarrow^* \sigma'' \vee \bar{M}; \sigma''[\text{cont} \mapsto s_2] \rightsquigarrow_{fin}^* \sigma']]$

G.8 Summarised Executions

We repeat the two diagrams given in §7.

The diagram opposite shows such an execution: $\bar{M} \cdot M; \sigma_2 \rightsquigarrow_{fin}^* \sigma_{30}$ consists of 4 calls to external objects, and 3 calls to internal objects. The calls to external objects are from σ_2 to σ_3 , from σ_3 to σ_4 , from σ_9 to σ_{10} , and from σ_{16} to σ_{17} . The calls to internal objects are from σ_5 to σ_6 , from σ_7 to σ_8 , and from σ_{21} to σ_{23} .

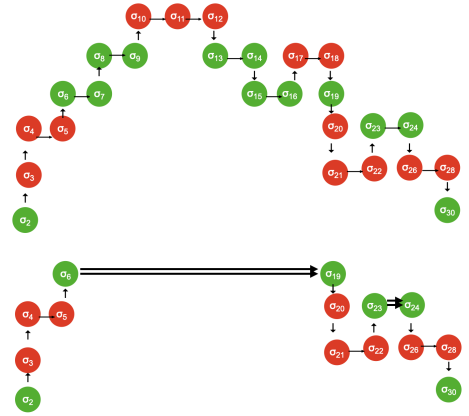
In terms of our example, we want to summarise the execution of the two “outer” internal, public methods into the “large” steps σ_6 to σ_{19} and σ_{23} to σ_{24} . And are not concerned with the states reached from these two public method executions.

In order to express such summaries, Def. G.27 introduces the following concepts:

- $(\bar{M} \cdot M, \sigma_{sc}); \sigma \rightsquigarrow_e^* \sigma'$ execution from σ to σ' scoped by σ_{sc} , involving external states only.
- $(\bar{M} \cdot M); \sigma \rightsquigarrow_p^* \sigma' \text{ pb } \sigma_1$ σ is an external state calling an internal public method, and σ' is the state after return from the public method, and σ_1 is the first state upon entry to the public method.

Continuing with our example, we have the following execution summaries:

- (1) $(\bar{M} \cdot M, \sigma_3); \sigma_3 \rightsquigarrow_e^* \sigma_5$ Purely external execution from σ_3 to σ_5 , scoped by σ_3 .
- (2) $(\bar{M} \cdot M); \sigma_5 \rightsquigarrow_p^* \sigma_{20} \text{ pb } \sigma_6$. Public method call from external state σ_5 into internal state σ_6 returning to σ_{20} . Note that this summarises two internal method executions ($\sigma_6 - \sigma_{19}$, and $\sigma_8 - \sigma_{14}$), and two external method executions ($\sigma_6 - \sigma_{19}$, and $\sigma_8 - \sigma_{14}$).
- (3) $(\bar{M} \cdot M, \sigma_3); \sigma_{20} \rightsquigarrow_e^* \sigma_{21}$.



¹⁶This can be done by looking in the caller’s frame – ie the one right under the topmost frame – the class of the current receiver and the name of the currently executing method, and then looking up the method definition in the module M ; if not defined there, then it is not public.

- (4) $(\bar{M} \cdot M); \sigma_{21} \rightsquigarrow_p^* \sigma_{25} \mathbf{pb} \sigma_{23}$. Public method call from external state σ_{21} into internal state σ_{23} , and returning to external state σ_{25} .
- (5) $(\bar{M} \cdot M, \sigma_3); \sigma_{25} \rightsquigarrow_e^* \sigma_{28}$. Purely external execution from σ_{25} to σ_{28} , scoped by σ_3 .

Definition G.27. For any module M where M is the internal module, external modules \bar{M} , and states $\sigma_{sc}, \sigma, \sigma_1, \dots, \sigma_n$, and σ' , we define:

- $$\begin{aligned}
 (1) \quad & (\bar{M} \cdot M, \sigma_{sc}); \sigma \rightsquigarrow_e^* \sigma' \triangleq \begin{cases} M, \sigma \models \text{extl} \wedge \\ \left[\begin{array}{l} \sigma = \sigma' \wedge |\sigma_{sc}| \leq |\sigma| \wedge |\sigma_{sc}| \leq |\sigma'| \vee \\ \exists \sigma'' [(\bar{M} \cdot M, \sigma_{sc}); \sigma \rightsquigarrow \sigma'' \wedge (\bar{M} \cdot M, \sigma_{sc}); \sigma'' \rightsquigarrow_e^* \sigma'] \end{array} \right] \end{cases} \\
 (2) \quad & (\bar{M} \cdot M); \sigma \rightsquigarrow_p^* \sigma' \mathbf{pb} \sigma_1 \triangleq \begin{cases} M, \sigma \models \text{extl} \wedge \\ \exists \sigma'_1 [(\bar{M} \cdot M, \sigma); \sigma \rightsquigarrow \sigma'_1 \wedge M, \sigma'_1 \models \text{pub} \wedge \\ \bar{M} \cdot M; \sigma_1 \rightsquigarrow_{fin}^* \sigma'_1 \wedge \bar{M} \cdot M; \sigma'_1 \rightsquigarrow \sigma'] \end{cases} \\
 (3) \quad & (\bar{M} \cdot M, \sigma_{sc}); \sigma \rightsquigarrow_{e,p}^* \sigma' \mathbf{pb} \epsilon \triangleq (\bar{M} \cdot M, \sigma_{sc}); \sigma \rightsquigarrow_e^* \sigma'' \\
 (4) \quad & (\bar{M} \cdot M, \sigma_{sc}); \sigma \rightsquigarrow_{e,p}^* \sigma' \mathbf{pb} \sigma_1 \triangleq \exists \sigma'_1, \sigma'_2. \begin{cases} (\bar{M} \cdot M, \sigma_{sc}); \sigma \rightsquigarrow_e^* \sigma'_1 \wedge (\bar{M} \cdot M); \sigma'_1 \rightsquigarrow_p^* \sigma'_2 \mathbf{pb} \sigma_1 \wedge \\ (\bar{M} \cdot M, \sigma_{sc}); \sigma'_2 \rightsquigarrow_e^* \sigma' \end{cases} \\
 (5) \quad & (\bar{M} \cdot M, \sigma_{sc}); \sigma \rightsquigarrow_{e,p}^* \sigma' \mathbf{pb} \sigma_1 \dots \sigma_n \triangleq \exists \sigma'_1. [(\bar{M} \cdot M, \sigma_{sc}); \sigma \rightsquigarrow_{e,p}^* \sigma'_1 \mathbf{pb} \sigma_1 \wedge (\bar{M} \cdot M, \sigma_{sc}); \sigma'_1 \rightsquigarrow_{e,p}^* \sigma' \mathbf{pb} \sigma_2 \dots \sigma_n] \\
 (6) \quad & \bar{M} \cdot M; \sigma \rightsquigarrow_{e,p}^* \sigma' \triangleq \exists n \in \mathbb{N}. \exists \sigma_1, \dots, \sigma_n. (\bar{M} \cdot M, \sigma); \sigma \rightsquigarrow_{e,p}^* \sigma' \mathbf{pb} \sigma_1 \dots \sigma_n
 \end{aligned}$$

Note that $(\bar{M} \cdot M, \sigma_{sc}); \sigma \rightsquigarrow_e^* \sigma'$ implies that σ is external, but does not imply that σ' is external. $(\bar{M} \cdot M, \sigma); \sigma \rightsquigarrow_e^* \sigma'$. On the other hand, $(\bar{M} \cdot M, \sigma_{sc}); \sigma \rightsquigarrow_{e,p}^* \sigma' \mathbf{pb} \sigma_1 \dots \sigma_n$ implies that σ and σ' are external, and $\sigma_1, \dots, \sigma_n$ are internal and public. Finally, note that in part (6) above it is possible that $n = 0$, and so $\bar{M} \cdot M; \sigma \rightsquigarrow_{e,p}^* \sigma'$ also holds when Finally, note that the decomposition used in (5) is not unique, but since we only care for the public states this is of no importance.

Lemma G.28 says that

- (1) Any terminating execution which starts at an external state (σ) consists of a number of external states interleaved with another number of terminating calls to public methods.
- (2) Any execution execution which starts at an external state (σ) and reaches another state (σ') also consists of a number of external states interleaved with another number of terminating calls to public methods, which may be followed by a call to some public method (at σ_2), and from where another execution, scoped by σ_2 reaches σ' .

Auxiliary Lemma G.28. [Summarised Executions] For module M , modules \bar{M} , and states σ, σ' :

If $M, \sigma \models \text{extl}$, then

- (1) $M \cdot \bar{M}; \sigma \rightsquigarrow_{fin}^* \sigma' \implies \bar{M} \cdot M; \sigma \rightsquigarrow_{e,p}^* \sigma'$
- (2) $M \cdot \bar{M}; \sigma \rightsquigarrow^* \sigma' \implies$
 - (a) $\bar{M} \cdot M; \sigma \rightsquigarrow_{e,p}^* \sigma' \vee$
 - (b) $\exists \sigma_c, \sigma_d. [\bar{M} \cdot M; \sigma \rightsquigarrow_{e,p}^* \sigma_c \wedge \bar{M} \cdot M; \sigma_c \rightsquigarrow \sigma_d \wedge M, \sigma_c \models \text{pub} \wedge \bar{M} \cdot M; \sigma_d \rightsquigarrow^* \sigma']$

Auxiliary Lemma G.29. [Preservation of Encapsulated Assertions] For any module M , modules \bar{M} , assertion A , and states $\sigma_{sc}, \sigma, \sigma_1 \dots \sigma_n, \sigma_a, \sigma_b$ and σ' :

If

$$M \vdash \text{Enc}(A) \wedge fv(A) = \emptyset \wedge M, \sigma, k \models A \wedge k \leq |\sigma_{sc}|.$$

Then

- (1) $M, \sigma \models \text{ext1} \wedge (\bar{M} \cdot M, \sigma_{sc}); \sigma \rightsquigarrow \sigma' \implies M, \sigma', k \models A$
- (2) $(\bar{M} \cdot M, \sigma_{sc}); \sigma \rightsquigarrow_e^* \sigma' \implies M, \sigma', k \models A$
- (3) $(\bar{M} \cdot M, \sigma_{sc}); \sigma \rightsquigarrow_{e,p}^* \sigma' \mathbf{pb} \sigma_1 \dots \sigma_n \wedge$
 $\forall i \in [1..n]. \forall \sigma_f. [M, \sigma_i, k \models A \wedge M \cdot \bar{M}; \sigma_i \rightsquigarrow_{fin}^* \sigma_f \implies M, \sigma_f, k \models A]$
 \implies
 $M, \sigma', k \models A$
 \wedge
 $\forall i \in [1..n]. M, \sigma_i, k \models A$
 \wedge
 $\forall i \in [1..n]. \forall \sigma_f. [M \cdot \bar{M}; \sigma_i \rightsquigarrow_{fin}^* \sigma_f \implies M, \sigma_f, k \models A]$

Proof Sketch

- (1) is proven by Def. of $Enc(_)$ and the fact $|\sigma'| \geq |\sigma_{sc}|$ and therefore $k \leq |\sigma'|$. In particular, the step $(\bar{M} \cdot M, \sigma_{sc}); \sigma \rightsquigarrow \sigma'$ may push or pop a frame onto σ . If it pops a frame, then $M, \sigma', k \models A$ holds by definition. If it pushes a frame, then $M, \sigma' \models A$, by lemma 4.8; this gives that $M, \sigma', k \models A$.
- (2) by induction on the number of steps in $(\bar{M} \cdot M, \sigma_{sc}); \sigma \rightsquigarrow_e^* \sigma'$, and using (1).
- (3) by induction on the number of states appearing in $\sigma_1 \dots \sigma_n$, and using (2).

End Proof Sketch

G.9 Sequences, Sets, Substitutions and Free Variables

Our system makes heavy use of textual substitution, textual inequality, and the concept of free variables in assertions.

In this subsection we introduce some notation and some lemmas to deal with these concepts. These concepts and lemmas are by no means novel; we list them here so as to use them more easily in the subsequent proofs.

Definition G.30 (Sequences, Disjointness, and Disjoint Concatenation). For any variables v, w , and sequences of variables \bar{v}, \bar{w} we define:

- $v \in \bar{w} \triangleq \exists \bar{w}_1, \bar{w}_2 [\bar{w} = \bar{w}_1, v, \bar{w}_2]$
- $v \# w \triangleq \neg(v \stackrel{\text{txt}}{=} w)$.
- $\bar{v} \subseteq \bar{w} \triangleq \forall v. [v \in \bar{v} \implies v \in \bar{w}]$
- $\bar{v} \# \bar{w} \triangleq \forall v \in \bar{v}. \forall w \in \bar{w}. [v \# w]$
- $\bar{v} \cap \bar{w} \triangleq \bar{u}$, such that $\forall u. [u \in \bar{v} \cap \bar{w} \Leftrightarrow [u \in \bar{v} \wedge u \in \bar{w}]$
- $\bar{v} \setminus \bar{w} \triangleq \bar{u}$, such that $\forall u. [u \in \bar{v} \setminus \bar{w} \Leftrightarrow [u \in \bar{v} \wedge u \notin \bar{w}]$
- $\bar{v}; \bar{w} \triangleq \bar{v}, \bar{w}$ if $\bar{v} \# \bar{w}$ and undefined otherwise.

Lemma G.31 (Substitutions and Free Variables). For any sequences of variables $\bar{x}, \bar{y}, \bar{z}, \bar{v}, \bar{w}$, a variable w , any assertion A , we have

- (1) $\bar{x}[\bar{y}/\bar{x}] = \bar{y}$
- (2) $\bar{x} \# \bar{y} \implies \bar{y}[\bar{z}/\bar{x}] = \bar{y}$
- (3) $\bar{z} \subseteq \bar{y} \implies \bar{y}[\bar{z}/\bar{x}] \subseteq \bar{y}$
- (4) $\bar{y} \subseteq \bar{z} \implies \bar{y}[\bar{z}/\bar{x}] \subseteq \bar{z}$
- (5) $\bar{x} \# \bar{y} \implies \bar{z}[\bar{y}/\bar{x}] \# \bar{x}$
- (6) $Fv(A[\bar{y}/\bar{x}]) = Fv(A)[\bar{y}/\bar{x}]$
- (7) $Fv(A) = \bar{x}; \bar{v}, Fv(A[\bar{y}/\bar{x}]) = \bar{y}; \bar{w} \implies \bar{v} = (\bar{y} \cap \bar{v}); \bar{w}$
- (8) $\bar{v} \# \bar{x} \# \bar{y} \# \bar{u} \implies w[u/x][v/y] \stackrel{\text{txt}}{=} w[v/y][u/x]$

- (9) $\bar{v} \# \bar{x} \# \bar{y} \# \bar{u} \implies A[\bar{u}/\bar{x}][\bar{v}/\bar{y}] \stackrel{\text{txt}}{=} A[\bar{v}/\bar{y}][\bar{u}/\bar{x}]$
 (10) $(fv(A[\bar{y}/\bar{x}]) \setminus \bar{y}) \# \bar{x}$
 (11)

PROOF. (1) by induction on the number of elements in \bar{x}

(2) by induction on the number of elements in \bar{y}

(3) by induction on the number of elements in \bar{y}

(4) by induction on the number of elements in \bar{y}

(5) by induction on the structure of A

(6) by induction on the structure of A

(7) Assume that

(ass1) $Fv(A) = \bar{x}; \bar{v}$,

(ass2) $Fv(A[\bar{y}/\bar{x}]) = \bar{y}; \bar{w}$

We define:

(a) $\bar{y}_0 \triangleq \bar{v} \cap \bar{y}$, $\bar{v}_2 \triangleq \bar{v} \setminus \bar{y}$, $\bar{y}_1 = \bar{y}_0[\bar{x}/\bar{y}]$

This gives:

(b) $\bar{y}_0 \# \bar{v}_2$

(c) $\bar{v} = \bar{y}_0; \bar{v}_2$

(d) $\bar{y}_1 \subseteq \bar{y}$

(e) $\bar{v}_2[\bar{y}/\bar{x}] = \bar{v}_2$, from assumption and (a) we have $\bar{x} \# \bar{v}_2$ and by Lemma G.31 part (2)

We now calculate

$$\begin{aligned}
 Fv(A[\bar{y}/\bar{x}]) &= (\bar{x}; \bar{v})[\bar{y}/\bar{x}] && \text{by (ass1), and Lemma G.31 part (5).} \\
 &= (\bar{x}; \bar{y}_0; \bar{v}_2)[\bar{y}/\bar{x}] && \text{by (c) above} \\
 &= \bar{x}[\bar{y}/\bar{x}], \bar{y}_0[\bar{y}/\bar{x}], \bar{v}_2[\bar{y}/\bar{x}] && \text{by distributivity of } [\dots] \\
 &= \bar{y}, \bar{y}_1, \bar{v}_2 && \text{by Lemma G.31 part (1), (a), and (e).} \\
 &= \bar{y}; \bar{v}_2 && \text{because (d), and } \bar{y} \# \bar{v}_2 \\
 Fv(A[\bar{y}/\bar{x}]) &= \bar{y}; \bar{w} && \text{by (ass2)}
 \end{aligned}$$

The above gives that $\bar{v}_2 = \bar{w}$. This, together with (a) and (c) give that $\bar{v} = (\bar{y} \cap \bar{v}); \bar{w}$

- (8) By case analysis on whether $w \in \bar{x}$... etc
 (9) By induction on the structure of A , and the guarantee from (8).
 (10) We take a variable sequence \bar{z} such that

(a) $Fv(A) \subseteq \bar{x}; \bar{z}$

This gives that

(b) $\bar{x} \# \bar{z}$

Part (6) of our lemma and (a) give

(c) $Fv(A[\bar{y}/\bar{x}]) \subseteq \bar{y}, \bar{z}$

Therefore

(d) $Fv(A[\bar{y}/\bar{x}]) \setminus \bar{y} \subseteq \bar{z}$

The above, together with (b) conclude the proof

□

Lemma G.32 (Substitutions and Adaptations). For any sequences of variables \bar{x} , \bar{y} , sequences of expressions \bar{e} , and any assertion A , we have

$$\bullet \bar{x} \# \bar{y} \implies (A[\bar{e}/\bar{x}]) \neg \bar{y} \stackrel{\text{txt}}{=} (A \neg \bar{y})[\bar{e}/\bar{x}]$$

PROOF. We first consider A to be $\langle e \rangle_0$, and just take one variable. Then,

$$(\langle e_0 \rangle[e/x]) \neg \bar{y} \stackrel{\text{txt}}{=} \langle e_0[e/x] \rangle \ltimes y,$$

and

$$(\langle e_0 \rangle \neg \bar{y})[e/x] \stackrel{\text{txt}}{=} \langle e_0[e/x] \rangle \neg y[e/x].$$

When $x \# y$ then the two assertions from above are textually equal. The rest follows by induction on the length of \bar{x} and the structure of A . \square

Lemma G.33. For assertion A , variables $\bar{x}, \bar{v}, \bar{y}, \bar{w}, \bar{v}_1$, addresses $\bar{\alpha}_x, \bar{\alpha}_y, \bar{\alpha}_v$, and $\bar{\alpha}_{v_1}$
If

- a. $Fv(A) \stackrel{\text{txt}}{=} \bar{x}; \bar{v}, Fv(A[\bar{y}/\bar{x}]) \stackrel{\text{txt}}{=} \bar{y}; \bar{w},$
- b. $\forall x \in \bar{x}. [x[\bar{y}/\bar{x}][\bar{\alpha}_y/\bar{y}] = x[\bar{\alpha}_x/\bar{x}]]$
- c. $\bar{v} \stackrel{\text{txt}}{=} \bar{v}_1; \bar{w}, \quad \bar{v}_1 \stackrel{\text{txt}}{=} \bar{y} \cap \bar{v}, \quad \bar{\alpha}_{v,1} = \bar{v}_1[\bar{\alpha}_v/\bar{v}]$

then

$$\bullet A[\bar{y}/\bar{x}][\bar{\alpha}_y/\bar{y}] \stackrel{\text{txt}}{=} A[\bar{\alpha}_x/\bar{x}][\bar{\alpha}_{v,1}/\bar{v}_1]$$

PROOF.

From Lemma G.31, part 7, we obtain $(*) \bar{v} = (\bar{y} \cap \bar{v}); \bar{w}$

We first prove that

$$(**) \forall z \in Fv(A) [z[\bar{y}/\bar{x}][\bar{\alpha}_y/\bar{y}] \stackrel{\text{txt}}{=} z[\bar{\alpha}_x/\bar{x}][\bar{\alpha}_{v,1}/\bar{v}_1].$$

Take any arbitrary $z \in Fv(A)$.

Then, by assumptions a. and c., and (*) we have that either $z \in \bar{x}$, or $z \in \bar{v}_1$, or $z \in \bar{w}$.

1st Case $z \in \bar{x}$. Then, there exists some $y_z \in \bar{y}$, and some $\alpha_z \in \bar{\alpha}_y$, such that $z[\bar{y}/\bar{x}] = y_z$ and $y_z[\bar{\alpha}_y/\bar{y}] = \alpha_z$. On the other hand, by part b. we obtain, that $z[\bar{\alpha}_x/\bar{x}] = \alpha_z$. And because $\bar{v}_1 \# \bar{\alpha}_x$ we also have that $\alpha_z[\bar{\alpha}_{v,1}/\bar{v}_1] = \alpha_z$. This concludes the case.

2nd Case $z \in \bar{v}_1$, which means that $z \in \bar{y} \cap \bar{v}$. Then, because $\bar{x} \# \bar{v}$, we have that $z[\bar{y}/\bar{x}] = z$. And because $z \in \bar{y}$, we obtain that there exists a α_z , so that $z[\bar{\alpha}_y/\bar{y}] = \alpha_z$. Similarly, because $\bar{x} \# \bar{v}$, we also obtain that $z[\bar{\alpha}_x/\bar{x}] = z$. And because $\bar{v}_1 \subseteq \bar{y}$, we also obtain that $z[\bar{\alpha}_{v,1}/\bar{v}_1] = z[\bar{\alpha}_y/\bar{y}]$. This concludes the case.

3rd Case $z \in \bar{w}$. From part a. of the assumptions and from (*) we obtain $\bar{w} \# \bar{y} \# \bar{x}$, which implies that $z[\bar{y}/\bar{x}][\bar{\alpha}_y/\bar{y}] = z$. Moreover, (*) also gives that $\bar{w} \# \bar{v}_1$, and this gives that $z[\bar{\alpha}_x/\bar{x}][\bar{\alpha}_{v,1}/\bar{v}_1] = z$. This concludes the case

The lemma follows from (*) and structural induction on A . \square

G.10 Reachability, Heap Identity, and their properties

We consider states with the same heaps ($\sigma \sim \sigma'$) and properties about reachability of an address from another address ($Reach(\alpha, \alpha')_\sigma$).

Definition G.34. For any state σ , addresses α, α' , we define

- $Reach(\alpha, \alpha')_\sigma \triangleq \exists \bar{f}. [\lfloor \alpha. \bar{f} \rfloor_\sigma = \alpha']$
- $\sigma \sim \sigma' \triangleq \exists \chi, \bar{\phi}_1, \bar{\phi}_2. [\sigma = (\bar{\phi}_1, \chi) \wedge \sigma' = (\bar{\phi}_1, \chi)]$

Lemma G.35. For any module M , state σ , addresses $\alpha, \alpha', \alpha''$

- (1) $M, \sigma \models \langle \alpha \rangle \neg \alpha' \wedge Reach(\alpha', \alpha'')_\sigma \implies M, \sigma \models \langle \alpha \rangle \neg \alpha''$
- (2) $\sigma \sim \sigma' \implies [Reach(\alpha, \alpha')_\sigma \iff Reach(\alpha, \alpha')_{\sigma'}]$
- (3) $\sigma \sim \sigma' \implies [M, \sigma \models \langle \alpha \rangle \neg \alpha'' \iff M, \sigma' \models \langle \alpha \rangle \neg \alpha'']$
- (4) $\sigma \sim \sigma' \wedge Fv(A) = \emptyset \wedge Stbl(A) \implies [M, \sigma \models A \iff M, \sigma' \models A]$

PROOF.

- (1) By unfolding/folding the definitions
- (2) By unfolding/folding the definitions

- (3) By unfolding/folding definitions.
- (4) By structural induction on A , and Lemma G.35 part 3.

□

G.11 Preservation of assertions when pushing or popping frames

In this section we discuss the preservation of satisfaction of assertions when calling methods or returning from methods – *i.e.* when pushing or popping frames. Namely, since pushing/popping frames does not modify the heap, these operations should preserve satisfaction of some assertion A , up to the fact that a) passing an object as a parameter of a result might break its protection, and b) the bindings of variables change with pushing/popping frames. To deal with a) upon method call, we require that the fame being pushed or the frame to which we return is internal ($M, \sigma' \models \text{intl}$), or require the adapted version of an assertion (*i.e.* $A \rightarrow \bar{v}$ rather than A). To deal with b) we either require that there are no free variables in A , or we break the free variables of A into two parts, *i.e.* $Fv(A_{in}) = \bar{v}_1; \bar{v}_2$, where the value of \bar{v}_3 in the caller is the same as that of \bar{v}_1 in the called frame. This is described in lemmas G.40 - G.42.

We have four lemmas: Lemma G.40 describes preservation from a caller to an internal called, lemma G.41 describes preservation from a caller to any called, Lemma G.42 describes preservation from an internal called to a caller, and Lemma G.43 describes preservation from an any called to a caller, These four lemmas are used in the soundness proof for the four Hoare rules about method calls, as given in Fig. 8.

In the rest of this section we will first introduce some further auxiliary concepts and lemmas, and then state, discuss and prove Lemmas G.40 - G.42.

Plans for next three subsections Lemmas G.40-G.41 are quite complex, because they deal with substitution of some of the assertions' free variables. We therefore approach the proofs gradually: We first state and prove a very simplified version of Lemmas G.40-G.41, where the assertion (A_{in} or A_{out}) is only about protection and only contains addresses; this is the only basic assertion which is not *Stbl*. We then state a slightly more general version of Lemmas G.40-G.41, where the assertion (A_{in} or A_{out}) is variable-free.

G.12 Preservation of variable-free simple protection when pushing/popping frames

We now move to consider preservation of variable-free assertions about protection when pushing/popping frames

Lemma G.36 (From caller to called - protected, and variable-free). For any address α , addresses $\bar{\alpha}$, states σ, σ' , and frame ϕ .

If $\sigma' = \sigma \nabla \phi$ then

- | | | |
|-------------------------------------------------------------------------------------------------------------------------------------------------------|------------|------------------------------------------------|
| a. $M, \sigma, k \models \langle \alpha \rangle \wedge M, \sigma' \models \text{intl} \wedge \text{Rng}(\phi) \subseteq \text{LocRchbl}(\sigma)$ | \implies | $M, \sigma', k \models \langle \alpha \rangle$ |
| b. $M, \sigma, k \models \langle \alpha \rangle \leftarrow \bar{\alpha} \wedge \text{Rng}(\phi) \subseteq \bar{\alpha}$ | \implies | $M, \sigma' \models \langle \alpha \rangle$ |
| c. $M, \sigma, k \models \langle \alpha \rangle \wedge \langle \alpha \rangle \leftarrow \bar{\alpha} \wedge \text{Rng}(\phi) \subseteq \bar{\alpha}$ | \implies | $M, \sigma', k \models \langle \alpha \rangle$ |

PROOF.

- a. (1) Take any $\alpha' \in \text{LocRchbl}(\sigma')$. Then, by assumptions, we have $\alpha' \in \text{LocRchbl}(\sigma)$. This gives, again by assumptions, that $M, \sigma \models \langle \alpha \rangle \leftarrow \alpha'$. By the construction of σ , and lemma G.35 part 1, we obtain that (2) $M, \sigma' \models \langle \alpha \rangle \leftarrow \alpha'$. From (1) and (2) and because $M, \sigma' \models \text{intl}$ we obtain that $M, \sigma' \models \langle \alpha \rangle$. Then apply lemma G.35 part G.5, and we are done.
- b. By unfolding and folding the definitions, and application of Lemma G.35 part 1.

c. By part G.35 part b. and G.5.

Notice that part G.36 requires that the called (σ') is internal, but parts b. and c. do not.

Notice also that the conclusion in part b. is $M, \sigma' \models \langle \alpha \rangle$ and not $M, \sigma', k \models \langle \alpha \rangle$. This is so, because it is possible that $M, \sigma \models \langle \alpha \rangle \Leftarrow \bar{\alpha}$ but $M, \sigma \not\models \langle \alpha \rangle$.

□

Lemma G.37 (From called to caller – protected, and variable-free). For any states σ, σ' , variable v , address α_v , addresses $\bar{\alpha}$, and statement *stmt*.

If $\sigma' = (\sigma \Delta)[v \mapsto \alpha_v][\text{cont} \mapsto \text{stmt}]$,

then

- a. $M, \sigma, k \models \langle \alpha \rangle \wedge k < |\sigma| \wedge M, \sigma \models \langle \alpha \rangle \Leftarrow \alpha_v \implies M, \sigma', k \models \langle \alpha \rangle$.
- b. $M, \sigma \models \langle \alpha \rangle \wedge \bar{\alpha} \subseteq \text{LocRchbl}(\sigma) \implies M, \sigma', k \models \langle \alpha \rangle \Leftarrow \bar{\alpha}$.

PROOF.

- a. (1) Take any $i \in [k..|\sigma'|]$. Then, by definitions and assumption, we have $M, \sigma[i] \models \langle \alpha \rangle$. Take any $\alpha' \in \text{LocRchbl}(\sigma[i])$. We obtain that $M, \sigma[i] \models \langle \alpha \rangle \Leftarrow \alpha'$. Therefore, $M, \sigma[i] \models \langle \alpha \rangle$. Moreover, $\sigma[i] = \sigma'[i]$, and we therefore obtain (2) $M, \sigma'[i] \models \langle \alpha \rangle$.

(3) Now take a $\alpha' \in \text{LocRchbl}(\sigma')$.

Then, we have that either (A): $\alpha' \in \text{LocRchbl}(\sigma[|\sigma'|])$, or (B): $\text{Reach}(\alpha_r, \alpha')_{\sigma'}$.

In the case of (A): Because $k, |\sigma| = |\sigma'| + 1$, and because $M, \sigma, k \models \langle \alpha \rangle$ we have $M, \sigma \models \langle \alpha \rangle \Leftarrow \alpha'$. Because $\sigma \sim \sigma'$ and Lemma G.35 part 3, we obtain (A') $M, \sigma' \models \langle \alpha \rangle \Leftarrow \alpha'$.

In the case of (B): Because $\sigma \sim \sigma'$ and lemma G.35 part 2, we obtain $\text{Reach}(\alpha_r, \alpha')_{\sigma}$. Then, applying Lemma G.35 part 3 and assumptions, we obtain (B') $M, \sigma' \models \langle \alpha \rangle \Leftarrow \alpha'$.

From (3), (A), (A'), (B) and (B') we obtain: (4) $M, \sigma' \models \langle \alpha \rangle$.

With (1), (2), (4) and Lemma G.35 part 4 we are done.

- b. From the definitions we obtain that $M, \sigma \models \langle \alpha \rangle \Leftarrow \bar{\alpha}$. Because $\sigma \sim \sigma'$ and lemma G.35 part 3, we obtain $M, \sigma' \models \langle \alpha \rangle \Leftarrow \bar{\alpha}$. And because of lemma G.4, part 3, we obtain $M, \sigma', k \models \langle \alpha \rangle \Leftarrow \bar{\alpha}$.

□

G.13 Preservation of variable-free, Stbl^+ , assertions when pushing/popping frames

We now move consider preservation of variable-free assertions when pushing/popping frames, and generalize the lemmas G.36 and G.37

Lemma G.38 (From caller to called - variable-free, and Stbl^+). For any assertion A , addresses $\bar{\alpha}$, states σ, σ' , and frame ϕ .

If $\sigma' = \sigma \nabla \phi$ and $\text{Stb}^+(A)$, and $Fv(A) = \emptyset$,

then

- a. $M, \sigma, k \models A \wedge M, \sigma' \models \text{intl} \wedge \text{Rng}(\phi) \subseteq \text{LocRchbl}(\sigma) \implies M, \sigma', k \models A$
- b. $M, \sigma, k \models A \neg (\bar{\alpha}) \wedge \text{Rng}(\phi) \subseteq \bar{\alpha} \implies M, \sigma' \models A$
- c. $M, \sigma, k \models A \wedge A \neg (\bar{\alpha}) \wedge \text{Rng}(\phi) \subseteq \bar{\alpha} \implies M, \sigma', k \models A$

PROOF.

- a. By Lemma G.36, part G.36 and structural induction on the definition of $\text{Stb}^+(_)$.
- b. By Lemma G.36, part G.36 and structural induction on the definition of $\text{Stb}^+(_)$.
- c. By part b. and Lemma G.4.

□

Lemma G.39 (From called to caller – protected, and variable-free). For any states σ, σ' , variable v , address α_v , addresses $\bar{\alpha}$, and statement $stmt$.

If $\sigma' = (\sigma \Delta)[v \mapsto \alpha_v][\text{cont} \mapsto stmt]$, and $Stb^+(A)$, and $Fv(A) = \emptyset$ then

- a. $M, \sigma, k \models A \wedge k < |\sigma| \wedge M, \sigma \models A \neg \alpha_v \implies M, \sigma', k \models A.$
- b. $M, \sigma \models A \wedge \bar{\alpha} \subseteq LocRchbl(\sigma) \implies M, \sigma', k \models A \neg (\bar{\alpha}).$

PROOF.

- a. By Lemma G.37, part a. and structural induction on the definition of $Stb^+(_)$.
- b. By Lemma G.37, part b. and structural induction on the definition of $Stb^+(_)$.

□

G.14 Preservation of assertions when pushing or popping frames – stated and proven

Lemma G.40 (From caller to internal called). For any assertion A_{in} , states σ, σ' , variables $\bar{v}_1, \bar{v}_2, \bar{v}_3, \bar{v}_4, \bar{v}_6$, and frame ϕ .

If

- (i) $Stb^+(A_{in})$,
- (ii) $Fv(A_{in}) = \bar{v}_1; \bar{v}_2^{17}$, $Fv(A_{in}[\bar{v}_3/\bar{v}_1]) = \bar{v}_3; \bar{v}_4$, $\bar{v}_6 \triangleq \bar{v}_2 \cap \bar{v}_3; \bar{v}_4$,
- (iii) $\sigma' = \sigma \nabla \phi \wedge Rng(\phi) = \lfloor v_3 \rfloor_\sigma$
- (iv) $\lfloor v_1 \rfloor_{\sigma'} = \lfloor v_3 \rfloor_\sigma$,

then

- a. $M, \sigma, k \models A_{in}[\bar{v}_3/\bar{v}_1] \wedge M, \sigma' \models \text{int l} \implies M, \sigma', k \models A_{in}[\lfloor v_6 \rfloor_\sigma / \bar{v}_6]$
- b. $M, \sigma, k \models (A_{in}[\bar{v}_3/\bar{v}_1]) \neg (\bar{v}_3) \implies M, \sigma' \models A_{in}[\lfloor v_6 \rfloor_\sigma / \bar{v}_6].$

Discussion of Lemma. In lemma G.40, state σ is the state right before pushing the new frame on the stack, while state σ' is the state right after pushing the frame on the stack. That is, σ is the last state before entering the method body, and σ' is the first state after entering the method body. A_{in} stands for the method's precondition, while the variables \bar{v}_1 stand for the formal parameters of the method, and \bar{v}_3 stand for the actual parameters of the call. Therefore, \bar{v}_1 is the domain of the new frame, and $\bar{\sigma}v_3$ is its range. The variables \bar{v}_6 are the free variables of A_{in} which are not in \bar{v}_1 – c.f. Lemma G.31 part (7). Therefore if (a.) the callee is internal, and $A_{in}[\bar{v}_3/\bar{v}_1]$ holds at the call point, or if (b.) $(A_{in}[\bar{v}_3/\bar{v}_1]) \neg (\bar{v}_3)$ holds at the call point, then $A_{in}[\dots/\bar{v}_6]$ holds right after pushing ϕ onto the stack. Notice the difference in the conclusion in (a.) and (b.): in the first case we have deep satisfaction, while in the second case we only have shallow satisfaction.

PROOF.

We will use $\bar{\alpha}_1$ as short for $\{\lfloor v_1 \rfloor_{\sigma'}\}$, and $\bar{\alpha}_3$ as short for $\lfloor v_3 \rfloor_\sigma$.

We also define $\bar{v}_{6,1} \triangleq \bar{v}_2 \cap \bar{v}_3$, $\bar{\alpha}_{6,1} \triangleq \bar{v}_{6,1}[\lfloor v_6 \rfloor_\sigma / \bar{v}_6]$

We establish that

$$(*) \quad A_{in}[\bar{v}_3/\bar{v}_1][\lfloor v_3 \rfloor_\sigma / \bar{v}_3] \stackrel{\text{txt}}{=} A_{in}[\bar{\alpha}_1/\bar{v}_1][\bar{\alpha}_{6,1}/\bar{v}_{6,1}]$$

This holds by By Lemma G.33 and assumption (iv) of the current lemma.

And we define $\bar{v}_{6,2} \triangleq \bar{v}_2 \setminus \bar{v}_3$, $\bar{\alpha}_{6,2} \triangleq \bar{v}_{6,2}[\lfloor v_6 \rfloor_\sigma / \bar{v}_6]$.

a. Assume

$$\begin{aligned} M, \sigma, k &\models A_{in}[\bar{v}_3/\bar{v}_1]. && \text{By Lemma 4.5 this implies that} \\ M, \sigma, k &\models A_{in}[\bar{v}_3/\bar{v}_1][\bar{\alpha}_3/\bar{v}_3] && \text{By (*) from above we have} \\ M, \sigma, k &\models A_{in}[\bar{\alpha}_1/\bar{v}_1][\bar{\alpha}_{6,1}/\bar{v}_{6,1}] \end{aligned}$$

¹⁷As we said earlier: this gives also that the variable sequences are pairwise disjoint, i.e. $\bar{v}_1 \# \bar{v}_2$.

The above, and Lemma 4.5 give that

$$M, \sigma, k \models A_{in}[\overline{\alpha_1/v_1}][\overline{\alpha_{6,1}/v_{6,1}}][\overline{\alpha_{6,2}/v_{6,2}}]$$

The assertion above is variable-free. Therefore, by Lemma G.38 part a. we also obtain

$$M, \sigma', k \models A_{in}[\overline{\alpha_1/v_1}][\overline{\alpha_{6,1}/v_{6,1}}][\overline{\alpha_{6,2}/v_{6,2}}]$$

With 4.5 the above gives

$$M, \sigma', k \models A_{in}[\overline{[v_1]_{\sigma'}/v_1}][\overline{[v_6]_{\sigma}/v_6}]$$

By Lemma 4.5, we obtain

$$M, \sigma', k \models A_{in}[\overline{[v_6]_{\sigma}/v_6}]$$

b. Assume

$$M, \sigma, k \models (A_{in}[\overline{v_3/v_1}]) \neg (\overline{v_3}). \quad \text{By Lemma 4.5 this implies that}$$

$$M, \sigma, k \models ((A_{in}[\overline{v_3/v_1}]) \neg (\overline{v_3}))[\overline{\alpha_3/v_3}] \quad \text{which implies that}$$

$$M, \sigma, k \models (A_{in}[\overline{v_3/v_1}][\overline{\alpha_3/v_3}]) \neg (\overline{\alpha_3}) \quad \text{By (*) from above we have}$$

$$M, \sigma, k \models (A_{in}[\overline{\alpha_1/v_1}][\overline{\alpha_{6,1}/v_{6,1}}]) \neg (\overline{\alpha_3})$$

The above, and Lemma 4.5 give that

$$M, \sigma, k \models ((A_{in}[\overline{\alpha_1/v_1}][\overline{\alpha_{6,1}/v_{6,1}}]) \neg (\overline{\alpha_3}))[\overline{\alpha_{6,2}/v_{6,2}}]$$

And the above gives

$$M, \sigma, k \models (A_{in}[\overline{\alpha_1/v_1}][\overline{\alpha_{6,1}/v_{6,1}}][\overline{\alpha_{6,2}/v_{6,2}}]) \neg (\overline{\alpha_3})$$

The assertion above is variable-free. Therefore, by Lemma G.38 part b. we also obtain

$$M, \sigma' \models A_{in}[\overline{\alpha_1/v_1}][\overline{\alpha_{6,1}/v_{6,1}}][\overline{\alpha_{6,2}/v_{6,2}}]$$

We apply Lemma 4.5, and Lemma 4.5, and obtain

$$M, \sigma' \models A_{in}[\overline{[v_6]_{\sigma}/v_6}]$$

□

Lemma G.41 (From caller to any called). For any assertion A_{in} , states σ, σ' , variables $\overline{v_3}$ statement $stmt$, and frame ϕ .

If

- (i) $Stb^+(A_{in})$,
- (ii) $Fv(A_{in}) = \emptyset$,
- (iii) $\sigma' = \sigma \nabla \phi \quad \wedge \quad Rng(\phi) = \overline{[v_3]_{\sigma}}$,

then

- a. $M, \sigma, k \models A_{in} \neg (\overline{v_3}) \implies M, \sigma' \models A_{in}.$
- b. $M, \sigma, k \models (A_{in} \wedge (A_{in} \neg (\overline{v_3}))) \implies M, \sigma', k \models A_{in}$

PROOF. a. Assume that

$$M, \sigma, k \models A_{in} \neg (\overline{v_3})$$

By Lemma 4.5 this implies that

$$M, \sigma, k \models A_{in} \neg (\overline{[v_3]_{\sigma}}).$$

We now have a variable-free assertion, and by Lemma G.38, part b., we obtain

$$M, \sigma' \models A_{in}.$$

b. Assume that

$$M, \sigma, k \models A_{in} \wedge A_{in} \neg (\overline{v_3})$$

By Lemma 4.5 this implies that

$$M, \sigma, k \models A_{in} \wedge A_{in} \neg (\overline{[v_3]_{\sigma}}).$$

We now have a variable-free assertion, and by Lemma G.38, part b., we obtain

$$M, \sigma', k \models A_{in}.$$

□

Discussion of Lemma G.41. In this lemma, as in lemma G.40, σ stands for the last state before entering the method body, and σ' for the first state after entering the method body. A_{in} stands for a module invariant in which all free variables have been substituted by addresses. The lemma is intended for external calls, and therefore we have no knowledge of the method's formal parameters. The variables $\overline{v_3}$ stand for the actual parameters of the call, and therefore $\overline{[v_3]_\sigma}$ is the range of the new frame. Therefore if (a.) the adapted version, $A_{in} \neg \nabla(\overline{v_3})$, holds at the call point, then the unadapted version, A_{in} holds right after pushing ϕ onto the stack. Notice that even though the premise of (a.) requires deep satisfaction, the conclusion promises only weak satisfaction. Moreover, if (b.) the adapted as well as the unadapted version, $A_{in} \wedge A_{in} \neg \nabla(\overline{v_3})$ holds at the call point, then the unadapted version, A_{in} holds right after pushing ϕ onto the stack. Notice the difference in the conclusion in (a.) and (b.): in the first case we have shallow satisfaction, while in the second case we only have deep satisfaction.

Lemma G.42 (From internal called to caller). For any assertion A_{out} , states σ, σ' , variables res, u variable sequences $\overline{v_1}, \overline{v_3}, \overline{v_5}$, and statement $stmt$.

If

- (i) $Stb^+(A_{out})$,
- (ii) $Fv(A_{out}) \subseteq \overline{v_1}$,
- (iii) $\overline{[v_5]_{\sigma'}}, [res]_\sigma \subseteq LocRchbl(\sigma) \wedge M, \sigma' \models \text{intl}$.
- (iv) $\sigma' = (\sigma \Delta)[u \mapsto [res]_\sigma][\text{cont} \mapsto stmt] \wedge \overline{[v_1]_\sigma} = \overline{[v_3]_{\sigma'}}$.

then

- a. $M, \sigma, k \models A_{out} \wedge (A_{out} \neg \nabla res) \wedge |\sigma'| \geq k \implies M, \sigma', k \models A_{out} \overline{[v_3/v_1]}.$
- b. $M, \sigma \models A_{out} \implies M, \sigma', k \models (A_{out} \overline{[v_3/v_1]}) \neg \nabla \overline{v_5}.$

Discussion of Lemma G.42. State σ stands for the last state in the method body, and σ' for the first state after exiting the method call. A_{out} stands for a method postcondition. The lemma is intended for internal calls, and therefore we know the method's formal parameters. The variables $\overline{v_1}$ stand for the formal parameters of the method, and $\overline{v_3}$ stand for the actual parameters of the call. Therefore the formal parameters of the called have the same values as the actual parameters in the caller $\overline{[v_1]_\sigma} = \overline{[v_3]_{\sigma'}}$. Therefore (a.) and (b.) promise that if the postcondition A_{out} holds before popping the frame, then it also holds after popping frame after replacing the the formal parameters by the actual parameters $A_{out} \overline{[v_3/v_1]}$. As in earlier lemmas, there is an important difference between (a.) and (b.): In (a.), we require *deep satisfaction at the called*, and obtain at the deep satisfaction of the *unadapted* version ($A_{out} \overline{[v_3/v_1]}$) at the return point; while in (b.), we only require *shallow satisfaction at the called*, and obtain deep satisfaction of the *adapted* version ($(A_{out} \overline{[v_3/v_1]}) \neg \nabla \overline{v_5}$), at the return point.

PROOF.

We use the following short hands: α as $\overline{[res]_\sigma}$, $\overline{\alpha_1}$ for $\overline{[v_1]_\sigma}$, $\overline{\alpha_5}$ as short for $\overline{[v_5]_{\sigma'}}$.

a. Assume that

$$M, \sigma, k \models A_{out} \wedge A_{out} \neg \nabla res$$

By Lemma 4.5 this implies that

$$M, \sigma, k \models A_{out} \overline{[\alpha_1/v_1]} \wedge (A_{out} \overline{[\alpha_1/v_1]}) \neg \nabla \alpha.$$

We now have a variable-free assertion, and by Lemma G.39, part a., we obtain

$$M, \sigma, k \models A_{out} \overline{[\alpha_1/v_1]}.$$

By Lemma 4.5, and because $\overline{[v_1]_\sigma} = \overline{[v_3]_{\sigma'}}$ this implies that

$$M, \sigma, k \models A_{out} \overline{[v_3/v_1]}.$$

b. Assume that

$$M, \sigma \models A_{out}$$

By Lemma 4.5 this implies that

$$M, \sigma \models A_{out}[\overline{\alpha_1/v_1}]$$

We now have a variable-free assertion, and by Lemma G.39, part b., we obtain

$$M, \sigma', k \models A_{out}[\overline{\alpha_1/v_1}] \neg \overline{v_5}$$

By Lemma 4.5, and because $\overline{v_1}]_{\sigma} = \overline{v_3}]_{\sigma'}$ and $\alpha_5 = \overline{v_5}]_{\sigma'}$, we obtain

$$M, \sigma', k \models A_{out}[\overline{v_3/v_1}] \neg \overline{v_5}$$

□

Lemma G.43 (From any called to caller). For any assertion A_{out} , states σ, σ' , variables res, u variable sequence $\overline{v_5}$, and statement $stmt$.

If

$$(i) Stb^+(A_{out}),$$

$$(ii) Fv(A_{out}) = \emptyset,$$

$$(iii) \overline{v_5}]_{\sigma'}, [res]_{\sigma} \subseteq LocRchbl(\sigma).$$

$$(iv) \sigma' = (\sigma \Delta)[u \mapsto [res]_{\sigma}][\text{cont} \mapsto stmt].$$

then

$$a. M, \sigma \models A_{out}$$

$$b. M, \sigma, k \models A_{out} \wedge |\sigma'| \geq k$$

$$\implies M, \sigma', k \models A_{out} \neg \overline{v_5}.$$

$$\implies M, \sigma', k \models A_{out} \wedge A_{out} \neg \overline{v_5}$$

PROOF.

a. Assume that

$$M, \sigma \models A_{out}$$

Since A_{out} is a variable-free assertion, by Lemma G.39, part a., we obtain

$$M, \sigma', k \models A_{out} \neg \overline{v_5}.$$

By Lemma 4.5, we obtain

$$M, \sigma', k \models A_{out} \neg \overline{v_5}$$

b. Similar argument to the proof of Lemma G.42, part (b).

□

Discussion of lemma G.43. Similarly to lemma G.42, in this lemma, state σ stands for the last state in the method body, and σ' for the first state after exiting the method call. A_{out} stands for a method postcondition. The lemma is meant to apply to external calls, and therefore, we do not know the method's formal parameters, A_{out} is meant to stand for a module invariant where all the free variables have been substituted by addresses – i.e. A_{out} has no free variables. The variables $\overline{v_3}$ stand for the actual parameters of the call. Parts (a.) and (b.) promise that if the postcondition A_{out} holds before popping the frame, then its adapted version also holds after popping frame ($A_{out} \neg \overline{v_5}$). As in earlier lemmas, there is an important difference between (a.) and (b.) In (a.), we require *shallow satisfaction at the called*, and obtain deep satisfaction of the *adapted* version ($A_{out} \neg \overline{v_5}$) at the return point; while in (b.), we require *deep satisfaction at the called*, and obtain deep satisfaction of the *conjunction* of the *unadapted* with the *adapted* version ($A_{out} \wedge A_{out} \neg \overline{v_5}$), at the return point.

G.15 Use of Lemmas G.40-G.41

As we said earlier, Lemmas G.40-G.41 are used to prove the soundness of the Hoare logic rules for method calls.

In the proof of soundness of CALL_INT. we will use Lemma G.40 part (a.) and Lemma G.42 part (a.). In the proof of soundness of CALL_INT_ADAPT we will use Lemma G.40 part (b.) and Lemma

G.42 part (b.). In the proof of soundness of `CALL_EXT_ADAPT` we will use Lemma G.41 part (a.) and Lemma G.43 part (a.). And finally, in the proof of soundness of `CALL_EXT_ADAPT_STRONG` we will use Lemma G.41 part (b.) and Lemma G.43 part (b.).

G.16 Proof of Theorem 7.3 – part (A)

Begin Proof

Take any M, \overline{M} , with

$$(1) \vdash M.$$

We will prove that

$$(*) \forall \sigma, A, A', A''.$$

$$[M \vdash \{A\} \sigma.\text{cont}\{A'\} \parallel \{A''\} \implies M \models_{deep} \{A\} \text{stmt}\{A'\} \parallel \{A''\}].$$

by induction on the well-founded ordering $\ll_{M, \overline{M}}$.

Take $\sigma, A, A', A'', \overline{z}, \overline{w}, \overline{\alpha}, \sigma', \sigma''$ arbitrary. Assume that

$$(2) M \vdash \{A\} \sigma.\text{cont}\{A'\} \parallel \{A''\}$$

$$(3) \overline{w} = Fv(A) \cap \text{dom}(\sigma), \quad \overline{z} = Fv(A) \setminus \text{dom}(\sigma)^{18}$$

$$(4) M, \sigma, k \models A[\overline{\alpha}/\overline{z}]$$

To show

$$(**) \overline{M} \cdot M; \sigma \rightsquigarrow_{fin}^* \sigma' \implies M, \sigma', k \models A'[\overline{\alpha}/\overline{z}]$$

$$(***) \overline{M} \cdot M; \sigma \rightsquigarrow^* \sigma'' \implies M, \sigma'', k \models \text{extl} \rightarrow A''[\overline{\alpha}/\overline{z}][\overline{w}/\overline{w}]$$

We proceed by case analysis on the rule applied in the last step of the proof of (2). We only describe some cases.

MID By Theorem 7.2.

SEQU Therefore, there exist statements stmt_1 and stmt_2 , and assertions A_1, A_2 and A'' , so that $A_1 \stackrel{\text{txt}}{=} A$, and $A_2 \stackrel{\text{txt}}{=} A'$, and $\sigma.\text{cont} \stackrel{\text{txt}}{=} \text{stmt}_1; \text{stmt}_2$. We apply lemma G.26, and obtain that there exists an intermediate state σ_1 . The proofs for stmt_1 and stmt_2 , and the intermediate state σ_1 are in the \ll relation. Therefore, we can apply the inductive hypothesis.

COMBINE by induction hypothesis, and unfolding and folding the definitions

CONSEQU using Lemma G.4 part 4 and axiom G.1

CALL_INT Therefore, there exist $u, y_0, C, \overline{y}, A_{pre}, A_{post}$, and A_{mid} , such that

$$(5) \sigma.\text{cont} \stackrel{\text{txt}}{=} u := y_0.m(\overline{y}),$$

$$(6) \vdash M : \{A_{pre}\} D :: m(\overline{x} : \overline{D}) \{A_{post}\} \parallel \{A_{mid}\},$$

$$(7) A \stackrel{\text{txt}}{=} y_0 : D, \overline{y} : \overline{D} \wedge A_{pre}[y_0, \overline{y}/\text{this}, \overline{x}],$$

$$A' \stackrel{\text{txt}}{=} A_{post}[y_0, \overline{y}, u/\text{this}, \overline{x}, \text{res}],$$

$$A'' \stackrel{\text{txt}}{=} A_{mid}.$$

Also,

$$(8) \overline{M} \cdot M; \sigma \rightsquigarrow \sigma_1,$$

where

$$(8a) \sigma_1 \triangleq (\sigma \nabla (\text{this} \mapsto \lfloor y_0 \rfloor_\sigma, \overline{x} \mapsto \lfloor \overline{y} \rfloor_\sigma) [\text{cont} \mapsto \text{stmt}_m]),$$

$$(8b) \text{mBody}(m, D, M) = \overline{y} : \overline{D} \{ \text{stmt}_m \}.$$

We define the shorthands:

$$(9) A_{pr} \triangleq \text{this} : D, \overline{x} : \overline{D} \wedge A_{pre}.$$

$$(9a) A_{pra} \triangleq \text{this} : D, \overline{x} : \overline{D} \wedge A_{pre} \neg \nabla (y_0, \overline{y}).$$

¹⁸Remember that $\text{dom}(\sigma)$ is the set of variables defined in the top frame of σ

$$(9b) \ A_{poa} \triangleq A_{post} \wedge A_{post} \neg \text{res.}$$

By (1), (6), (7), (9), and definition of $\vdash M$ in Section 6.3 rule METHOD and we obtain

$$(10) \ M \vdash \{ A_{pra} \} \text{stmt}_m \{ A_{poa} \} \parallel \{ A_{mid} \}.$$

From (8) we obtain

$$(11) \ (A_{pra}, \sigma_1, A_{poa}, A_{mid}) \ll_{M, \overline{M}} (A, \sigma, A', A'')$$

In order to be able to apply the induction hypothesis, we need to prove something of the form $\dots \sigma_1 \models A_{pr} [\dots / f\bar{v}(A_{pr}) \setminus \text{dom}(\sigma_1)]$. To that aim we will apply Lemma G.40 part a. on (4), (8a) and (9). For this, we take

$$(12) \ \overline{v}_1 \triangleq \text{this}, \overline{x}, \quad \overline{v}_2 \triangleq Fv(A_{pr}) \setminus \overline{v}_1, \quad \overline{v}_3 \triangleq y_0, \overline{y}, \quad \overline{v}_4 \triangleq Fv(A) \setminus \overline{v}_3$$

These definitions give that

$$(12a) \ A \stackrel{\text{txt}}{=} A_{pr} [\overline{v}_3 / \overline{v}_1],$$

$$(12b) \ Fv(A_{pr}) = \overline{v}_1; \overline{v}_2.$$

$$(12c) \ Fv(A) = \overline{v}_3; \overline{v}_4.$$

With (12a), (12b), (12c), (and Lemma G.31 part (7), we obtain that

$$(12d) \ \overline{v}_2 = \overline{y}_r; \overline{v}_4, \quad \text{where } \overline{y}_r \triangleq \overline{v}_2 \cap \overline{v}_3$$

Furthermore, (8a), and (12) give that:

$$(12e) \ \lfloor v_1 \rfloor_{\sigma_1} = \lfloor v_3 \rfloor_{\sigma}$$

Then, (4), (12a), (12c) and (12f) give that

$$(13) \ M, \sigma, k \models A_{pr} [\overline{v}_3 / \overline{v}_1] [\alpha / z]$$

Moreover, we have that $\overline{z} \# \overline{v}_3$. From Lemma G.31 part (10) we obtain $\overline{z} \# \overline{v}_1$. And, because $\overline{\alpha}$ are addresses while \overline{v}_1 are variables, we also have that $\overline{\alpha} \# \overline{v}_1$. These facts, together with Lemma G.31 part (9) give that

$$(13a) \ A_{pr} [\overline{v}_3 / \overline{v}_1] [\alpha / z] \stackrel{\text{txt}}{=} A_{pr} [\overline{\alpha} / z] [\overline{v}_3 / \overline{v}_1]$$

From (13a) and (13), we obtain

$$(13b) \ M, \sigma, k \models A_{pr} [\overline{\alpha} / z] [\overline{v}_3 / \overline{v}_1]$$

From (4), (8a), (12a)-(12e) we see that the requirements of Lemma G.40 part a. are satisfied where we take A_{in} to be $A_{pr} [\overline{\alpha} / z]$. We use the definition of y_r in (12d), and define

$$(13c) \ \overline{v}_6 \triangleq y_r; (\overline{v}_4 \setminus \overline{z}) \quad \text{which, with (12d) also gives: } \overline{v}_2 = \overline{v}_6; \overline{z}$$

We apply Lemma G.40 part a. on (13b), (13c) and obtain

$$(14a) \ M, \sigma_1, k \models A_{pr} [\overline{\alpha} / z] [\lfloor v_6 \rfloor_{\sigma} / \overline{v}_6].$$

Moreover, we have the $M, \sigma_1 \models \text{int} \perp$. We apply lemma G.42.(G.42), and obtain

$$(14b) \ M, \sigma_1, k \models A_{pr} [\overline{\alpha} / z] [\lfloor v_6 \rfloor_{\sigma} / \overline{v}_6] \wedge A_{pr} \neg (\text{this}, \overline{y}).$$

With similar re-orderings to earlier, we obtain

$$(14b) \ M, \sigma_1, k \models A_{pra} [\overline{\alpha} / z] [\lfloor v_6 \rfloor_{\sigma} / \overline{v}_6].$$

For the proof of (**) as well as for the proof of (***), we will want to apply the inductive hypothesis. For this, we need to determine the value of $Fv(A_{pr}) \setminus \text{dom}(\sigma_1)$, as well as the value of $Fv(A_{pr}) \cap \text{dom}(\sigma_1)$. This is what we do next. From (8a) we have that

$$(15a) \ \text{dom}(\sigma_1) = \{\text{this}, \overline{x}\}.$$

This, with (12) and (12b) gives that

$$(15b) \ Fv(A_{pra}) \cap \text{dom}(\sigma_1) = \overline{v}_1.$$

$$(15c) \ Fv(A_{pra}) \setminus \text{dom}(\sigma_1) = \overline{v}_2.$$

Moreover, (12d) and (13d) give that

$$(15d) \ Fv(A_{pra}) \setminus \text{dom}(\sigma_1) = \overline{z}_2 = \overline{z}; \overline{v}_6.$$

Proving (**). Assume that $\overline{M} \cdot M; \sigma \rightsquigarrow_{fin}^* \sigma'$. Then, by the operational semantics, we obtain that there exists state σ'_1 , such that

$$(16) \overline{M} \cdot M; \sigma_1 \rightsquigarrow_{fin}^* \sigma'_1$$

$$(17) \sigma' = (\sigma'_1 \Delta) [u \mapsto \lfloor \text{res} \rfloor_{\sigma'_1}][\text{cont} \mapsto \epsilon].$$

We now apply the induction hypothesis on (14), (16), (15d), and obtain

$$(18) M, \sigma'_1, k \models (A_{post})[\overline{\alpha/z}][\overline{\lfloor v_6 \rfloor_{\sigma}/v_6}].$$

We now want to obtain something of the form $\dots\sigma' \models \dots A'$. We now want to be able to apply Lemma G.42, part a. on (18). Therefore, we define

$$(18a) A_{out} \triangleq A_{poa}[\overline{\alpha/z}][\overline{\lfloor v_6 \rfloor_{\sigma}/v_6}]$$

$$(18b) \overline{v_{1,a}} \triangleq \overline{v_1}, \text{res}, \quad \overline{v_{3,a}} \triangleq \overline{v_3}, u.$$

The wellformedness condition for specifications requires that $Fv(A_{post}) \subseteq Fv(A_{pr}) \cup \{\text{res}\}$. This, together with (9), (12d) and (18b) give

$$(19a) Fv(A_{out}) \subseteq \overline{v_{1,a}}$$

Also, by (18b), and (17), we have that

$$(19b) \lfloor v_{3,a} \rfloor_{\sigma'} = \lfloor v_{1,a} \rfloor_{\sigma'_1}.$$

From (4) we obtain that $k \leq |\sigma|$. From (8a) we obtain that $|\sigma_1| = |\sigma| + 1$. From (16) we obtain that $|\sigma'_1| = |\sigma_1|$, and from (17) we obtain that $|\sigma'| = |\sigma'_1| - 1$. All this gives that:

$$(19c) k \leq |\sigma'|$$

We now apply Lemma G.42, part a., and obtain

$$(20) M, \sigma', k \models A_{out}[\overline{v_{3,a}/v_{1,a}}].$$

We expand the definition from (18a), and re-order the substitutions by a similar argument as in in step (13a), using Lemma part (9), and obtain

$$(20a) M, \sigma', k \models A_{poa}[\overline{v_{3,a}/v_{1,a}}][\overline{\alpha/z}][\overline{\lfloor v_6 \rfloor_{\sigma}/v_6}].$$

By (20a), (18b), and because by Lemma B.2 we have that $\overline{\lfloor v_6 \rfloor_{\sigma}} = \overline{\lfloor v_6 \rfloor_{\sigma'}}$, we obtain

$$(21) M, \sigma', k \models (A_{poa})[y_0, \overline{y}, u/\text{this}, \overline{x}, \text{res}][\overline{\alpha/z}].$$

With (7) we conclude.

Proving (***). Take a σ'' . Assume that

$$(15) \overline{M} \cdot M; \sigma \rightsquigarrow^* \sigma''$$

$$(16) \overline{M} \cdot M, \sigma'' \models \text{extl}.$$

Then, from (8) and (15) we also obtain that

$$(15) \overline{M} \cdot M; \sigma_1 \rightsquigarrow^* \sigma''$$

By (10), (11) and application of the induction hypothesis on (13), (14c), and (15), we obtain that

$$(\beta') M, \sigma'', k \models A_{mid}[\overline{\alpha/z}][\overline{\lfloor w \rfloor_{\sigma}/w}].$$

and using (7) we are done.

CALL_EXT_ADAPT is in some parts, similar to CALL_INT. We highlight the differences in green .

Therefore, there exist $u, y_0, \overline{C}, D, \overline{y}$, and A_{inv} , such that

$$(5) \sigma.\text{cont} \stackrel{\text{txt}}{=} u := y_0.m(\overline{y}),$$

$$(6) \vdash M : \forall x : \overline{C}. \{A_{inv}\},$$

$$(7) A \stackrel{\text{txt}}{=} y_0 : \text{external}, x : \overline{C} \wedge A_{inv} \neg(y_0, \overline{y}),$$

$$A' \stackrel{\text{txt}}{=} A_{inv} \neg(y_0, \overline{y}),$$

$$A'' \stackrel{\text{txt}}{=} A_{inv}.$$

Also,

$$(8) \quad \overline{M} \cdot M; \sigma \rightsquigarrow \sigma_a,$$

where

$$(8a) \quad \sigma_a \triangleq (\sigma \nabla (\text{this} \mapsto \lfloor y_0 \rfloor_\sigma, \overline{p} \mapsto \lfloor y \rfloor_\sigma) [\text{cont} \mapsto \text{stmt}_m]),$$

$$(8b) \quad \text{mBody}(m, D, \overline{M}) = \overline{p} : \overline{D} \{ \text{stmt}_m \}.$$

$$(8c) \quad D \text{ is the class of } \lfloor y_0 \rfloor_\sigma, \text{ and } D \text{ is external.}$$

By (7), and well-formedness of module invariants, we obtain

$$(9a) \quad Fv(A_{inv}) \subseteq \overline{x},$$

$$(9a) \quad Fv(A) = y_0, \overline{y}, \overline{x}$$

By Barendregt, we also obtain that

$$(10) \quad \text{dom}(\sigma) \# \overline{x}$$

This, together with (3) gives that

$$(10) \quad \overline{z} = \overline{x}$$

From (4), (7) and the definition of satisfaction we obtain

$$(10) \quad M, \sigma, k \models (x : \overline{C} \wedge A_{inv} \nabla y_0, \overline{y}) [\alpha/z].$$

The above gives that

$$(10a) \quad M, \sigma, k \models ((\overline{x} : \overline{C}) [\alpha/z] \wedge (A_{inv} [\alpha/z])) \nabla y_0, \overline{y}.$$

We take A_{in} to be $(\overline{x} : \overline{C}) [\alpha/z] \wedge (A_{inv} [\alpha/z])$, and apply Lemma G.41, part a.. This gives that

$$(11) \quad M, \sigma_a \models (\overline{x} : \overline{C}) [\alpha/z] \wedge A_{inv} [\alpha/z]$$

Proving (**). We shall use the short hand

$$(12) \quad A_o \triangleq \overline{\alpha} : \overline{C} \wedge A_{inv} [\alpha/z].$$

Assume that $\overline{M} \cdot M; \sigma \rightsquigarrow_{fin}^* \sigma'$. Then, by the operational semantics, we obtain that there exists state σ'_b , such that

$$(16) \quad \overline{M} \cdot M; \sigma_a \rightsquigarrow_{fin}^* \sigma_b$$

$$(17) \quad \sigma' = (\sigma_b \Delta) [u \mapsto \lfloor \text{res} \rfloor_{\sigma'_1}][\text{cont} \mapsto \epsilon].$$

By Lemma G.28 part 1, and Def. G.27, we obtain that there exists a sequence of states $\sigma_1, \dots, \sigma_n$, such that

$$(17) \quad (\overline{M} \cdot M, \sigma_a); \sigma_a \rightsquigarrow_{e,p}^* \sigma_b \text{ pb } \sigma_1 \dots \sigma_n$$

By Def. G.27, the states $\sigma_1, \dots, \sigma_n$ are all public, and correspond to the execution of a public method. Therefore, by rule INVARIANT for well-formed modules, we obtain that

$$(18) \quad \forall i \in 1..n.$$

$$[M \vdash \{ \text{this} : D_i, \overline{p}_i : \overline{D}_i, \overline{x} : \overline{C} \wedge A_{inv} \} \sigma_i. \text{cont} \{ A_{inv,r} \} \parallel \{ A_{inv} \}]$$

where D_i is the class of the receiver, \overline{p}_i are the formal parameters, and \overline{D}_i are the types of the formal parameters of the i -th public method, and where we use the shorthand $A_{inv,r} \triangleq A_{inv} \nabla \text{res}$.

Moreover, (17) gives that

$$(19) \quad \forall i \in 1..n. [M \cdot \overline{M}; \sigma \rightsquigarrow^* \sigma_i]$$

From (18) and (19) we obtain

$$(20) \quad \forall i \in [1..n].$$

$$[(\text{this} : D_i, \overline{p}_i : \overline{D}_i, \overline{x} : \overline{C} \wedge A_{inv}, \sigma_i, A_{inv,r}, A_{inv})$$

$$\ll_{M, \overline{M}}$$

$$(A, \sigma, A', A'')]$$

We take

$$(21) \quad k = |\sigma_a| \text{ By application of the induction hypothesis on (20) we obtain that}$$

$$(22) \forall i \in [1..n]. \forall \sigma_f. [M, \sigma_i, k \models A_o \wedge M \cdot \bar{M}; \sigma_i \rightsquigarrow_{fin}^* \sigma_f \implies M, \sigma_f, k \models A_o]$$

We can now apply Lemma G.29, part 3, and because $|\sigma_a| = |\sigma_b|$, we obtain that

$$(23) M, \sigma_b \models A_{inv}[\bar{\alpha}/x]$$

We apply Lemma G.43 part a., and obtain

$$(24) M, \sigma' \models A_{inv}[\bar{\alpha}/x] \neg \forall y_0, \bar{y}$$

And since $A_{inv}[\bar{\alpha}/x] \neg \forall y_0, \bar{y}$ is stable, and by rearranging, and applying (10), we obtain

$$(25) M, \sigma', k \models (A_{inv} \neg \forall y_0, \bar{y})[\bar{\alpha}/z]$$

Apply (7), and we are done.

Proving (**). Take a σ'' . Assume that

$$(12) \bar{M} \cdot M; \sigma \rightsquigarrow^* \sigma''$$

$$(13) \bar{M} \cdot M, \sigma'' \models \text{ext l.}$$

We apply lemma 1, part 2 on (12) and see that there are two cases

1st Case $\bar{M} \cdot M; \sigma_a \rightsquigarrow_{e,p}^* \sigma''$

That is, the execution from σ_a to σ'' goes only through external states. We use (11), and that A_{inv} is encapsulated, and are done with lemma G.29, part 1.

2nd Case for some σ_c, σ_d . we have

$$\bar{M} \cdot M; \sigma_a \rightsquigarrow_{e,p}^* \sigma_c \wedge \bar{M} \cdot M; \sigma_c \rightsquigarrow \sigma_d \wedge M, \sigma_d \models \text{pub} \wedge \bar{M} \cdot M; \sigma_d \rightsquigarrow^* \sigma''$$

We apply similar arguments as in steps (17)-(23) and obtain

$$(14) M, \sigma_c \models A_{inv}[\bar{\alpha}/x]$$

State σ_c is a public, internal state; therefore there exists a Hoare proof that it preserves the invariant. By applying the inductive hypothesis, and the fact that $\bar{z} = \bar{x}$, we obtain:

$$(14) M, \sigma'' \models A_{inv}[\bar{\alpha}/z]$$

CALL_EXT_ADAPT_STRONG is very similar to CALL_EXT_ADAPT. We will summarize the similar steps, and highlight the differences in green .

Therefore, there exist $u, y_0, \bar{C}, D, \bar{y}$, and A_{inv} , such that

$$(5) \sigma.\text{cont} \stackrel{\text{txt}}{=} u := y_0.m(\bar{y}),$$

$$(6) \vdash M : \forall \bar{x} : \bar{C}. \{A_{inv}\},$$

$$(7) A \stackrel{\text{txt}}{=} y_0 : \text{external}, \bar{x} : \bar{C} \wedge A_{inv} \wedge A_{inv} \neg \forall (y_0, \bar{y}),$$

$$A' \stackrel{\text{txt}}{=} A_{inv} \wedge A_{inv} \neg \forall (y_0, \bar{y}),$$

$$A'' \stackrel{\text{txt}}{=} A_{inv}.$$

Also,

$$(8) \bar{M} \cdot M; \sigma \rightsquigarrow \sigma_a,$$

By similar steps to (8a)-(10) from the previous case, we obtain

$$(10a) M, \sigma, k \models A_{inv}[\bar{\alpha}/z] \wedge ((\bar{x} : \bar{C})[\bar{\alpha}/z] \wedge (A_{inv}[\bar{\alpha}/z])) \nabla y_0, \bar{y}.$$

We now apply lemma apply Lemma G.41, part b.. This gives that

$$(11) M, \sigma_a, k \models ((\bar{x} : \bar{C})[\bar{\alpha}/z] \wedge A_{inv}[\bar{\alpha}/z] \wedge ((\bar{x} : \bar{C})[\bar{\alpha}/z]) \nabla y_0, \bar{y}).$$

the rest is similar to earlier cases

End Proof

G.17 Proof Sketch of Theorem 7.3 – part (B)

Proof Sketch By induction on the cases for the specification S . If it is a method spec, then the theorem follows from 7.3. If it is a conjunction, then by inductive hypothesis.

The interesting case is $S \equiv \overline{\forall x : \overline{C}. \{A\}}$.

Assume that $M, \sigma, k \models A[\overline{\alpha/x}]$, that $M, \sigma \models \text{extl}$, that $M \cdot \overline{M}; \sigma \rightsquigarrow^* \sigma'$, and that $M, \sigma \models \text{extl}$. We want to show that $M, \sigma', k \models A[\overline{\alpha/x}]$.

Then, by lemma G.28, we obtain that either

(1) $\overline{M} \cdot M; \sigma \rightsquigarrow_{\text{e},p}^* \sigma'$, or

(2) $\exists \sigma_1, \sigma_2. [\overline{M} \cdot M; \sigma \rightsquigarrow_{\text{e},p}^* \sigma_1 \wedge \overline{M} \cdot M; \sigma_1 \rightsquigarrow \sigma_2 \wedge M, \sigma_2 \models \text{pub} \wedge \overline{M} \cdot M; \sigma_2 \rightsquigarrow^* \sigma']$

In Case (1), we apply G.29, part (3). In order to fulfill the second premise of Lemma G.29, part (3), we make use of the fact that $\vdash M$, apply the rule METHOD, and theorem 7.3. This gives us $M, \sigma', k \models A[\overline{\alpha/x}]$

In Case (2), we proceed as in (1) and obtain that $M, \sigma_1, k \models A[\overline{\alpha/x}]$. Because $M \vdash \text{Enc}(A)$, we also obtain that $M, \sigma_2, k \models A[\overline{\alpha/x}]$. Since we are now executing a public method, and because $\vdash M$, we can apply INVARIANT, and theorem 7.3, and obtain $M, \sigma', k \models A[\overline{\alpha/x}]$

End Proof Sketch

H PROVING LIMITED EFFECTS FOR THE SHOP/ACCOUNT EXAMPLE

In Section 2 we introduced a Shop that allows clients to make purchases through the buy method. The body of this method includes a method call to an unknown external object (`buyer.pay(...)`).

In this section we use our Hoare logic from Section 6 to outline the proof that the buy method does not expose the Shop's Account, its Key, or allow the Account's balance to be illicitly modified.

We outline the proof that $M_{good} \vdash S_2$, and that $M_{fine} \vdash S_2$. We also show why $M_{bad} \not\vdash S_2$.

We rewrite the code of M_{good} and so M_{fine} so that it adheres to the syntax as defined in Fig. 4 (§H.1). We extend the specification S_2 , so that it also makes a specification for the private method set (§H.2). After that, we outline the proofs that $M_{good} \vdash S_2$, and that $M_{fine} \vdash S_2$ (in §H.2), and that $M_{good} \vdash S_3$, and that $M_{fine} \vdash S_3$ (§H.4). These proofs have been mechanized in Coq, and the source code will be submitted as an artefact. We also discuss why $M_{bad} \not\vdash S_2$ (§H.3.2).

H.1 Expressing the Shop example in the syntax from Fig. 4

We now express our example in the syntax of Fig. 4. For this, we add a return type to each of the methods; We turn all local variables to parameter; We add an explicit assignment to the variable `res`; and We add a temporary variable `tmp` to which we assign the result of our void methods. For simplicity, we allow the shorthands `+=` and `-=`. And we also allow definition of local variables, e.g. `int price := ..`

```

1  module Mgood
2    ...
3  class Shop
4    field accnt : Account,
5    field invntry : Inventory,
6    field clients: ..
7
8    public method buy(buyer:external, anItem:Item, price: int,
9      myAccnt: Account, oldBalance: int, newBalance: int, tmp:int) : int
10     price := anItem.price;
11     myAccnt := this.accnt;
12     oldBalance := myAccnt.blnc;
13     tmp := buyer.pay(myAccnt, price)    // external call!
14     newBalance := myAccnt.blnc;
15     if (newBalance == oldBalance+price) then
16       tmp := this.send(buyer,anItem)
17     else
18       tmp := buyer.tell("you have not paid me") ;
19     res := 0
20
21     private method send(buyer:external, anItem:Item) : int
22     ...
23  class Account
24     field blnc : int
25     field key : Key
26
27     public method transfer(dest:Account, key':Key, amt:nat) :int
28     if (this.key==key') then
29       this.blnc-=amt;
30       dest.blnc+=amt
31     else
32       res := 0
33     res := 0

```

```

35784
35795     public method set(key':Key) : int
35806         if (this.key==null) then
35817             this.key:=key'
35828         else
35839             res := 0
35840             res := 0

```

Remember that M_{fine} is identical to M_{good} , except for the method `set`. We describe the module below.

```

35891 module Mfine
35902     ...
35913     class Shop
35924         ... as in Mgood
35935     class Account
35946         field blnce : int
35957         field key : Key
35968
35979     public method transfer(dest:Account, key':Key, amt:nat) :int
35990         ... as in Mgood
35991
35992     public method set(key':Key, k'':Key) : int
35993         if (this.key==key') then
35994             this.key:=key''
35995         else
35996             res := 0
35997             res := 0

```

H.2 Proving that M_{good} and M_{fine} satisfy S_2

We redefine S_2 so that it also describes the behaviour of method `send`. and have:

$$\begin{aligned}
 S_{2a} &\triangleq \{a : \text{Account} \wedge e : \text{external} \wedge \langle a.\text{key} \rangle \leftarrow^* e \\
 &\quad \text{private Shop} :: \text{send}(\text{buyer} : \text{external}, \text{anItem} : \text{Item}) \\
 &\quad \{ \langle a.\text{key} \rangle \leftarrow^* e \} \parallel \{ \langle a.\text{key} \rangle \leftarrow^* e \} \\
 S_{2b} &\triangleq \{a : \text{Account} \wedge a.\text{blnce} = b\} \\
 &\quad \text{private Shop} :: \text{send}(\text{buyer} : \text{external}, \text{anItem} : \text{Item}) \\
 &\quad \{ a.\text{blnce} = b \} \parallel \{ a.\text{blnce} = b \} \\
 S_{2, \text{strong}} &\triangleq S_2 \wedge S_{2a} \wedge S_{2b}
 \end{aligned}$$

For brevity we only show that `buy` satisfies our scoped invariants, as the all other methods of the M_{good} interface are relatively simple, and do not make any external calls.

To write our proofs more succinctly, we will use `ClassId::methId.body` as a shorthand for the method body of `methId` defined in `ClassId`.

Lemma H.1 (M_{good} satisfies $S_{2, \text{strong}}$). $M_{good} \vdash S_{2, \text{strong}}$

PROOF OUTLINE In order to prove that

$$M_{good} \vdash \forall a : \text{Account}. \{ \langle a.\text{key} \rangle \}$$

we have to apply INVARIANT from Fig. 9. That is, for each class C defined in M_{good} , and for each public method m in C , with parameters $\overline{y : D}$, we have to prove that

$$\begin{aligned} M_{good} \vdash & \{ \text{this} : C, \overline{y : D}, a : \text{Account} \wedge \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \Leftarrow (this, \overline{y}) \} \\ & C :: m.\text{body} \\ & \{ \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \Leftarrow \text{res} \} \parallel \{ \langle a.\text{key} \rangle \} \end{aligned}$$

Thus, we need to prove three Hoare quadruples: one for $\text{Shop} :: \text{buy}$, one for $\text{Account} :: \text{transfer}$, and one for $\text{Account} :: \text{set}$. That is, we have to prove that

$$\begin{aligned} (1?) \quad M_{good} \vdash & \{ A_{buy}, a : \text{Account} \wedge \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \Leftarrow \text{Ids}_{buy} \} \\ & \text{Shop} :: \text{buy}.\text{body} \\ & \{ \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \neg \text{res} \} \parallel \{ \langle a.\text{key} \rangle \} \\ (2?) \quad M_{good} \vdash & \{ A_{trns}, a : \text{Account} \wedge \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \Leftarrow \text{Ids}_{trns} \} \\ & \text{Account} :: \text{transfer}.\text{body} \\ & \{ \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \neg \text{res} \} \parallel \{ \langle a.\text{key} \rangle \} \\ (3?) \quad M_{good} \vdash & \{ A_{set}, a : \text{Account} \wedge \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \Leftarrow \text{Ids}_{set} \} \\ & \text{Account} :: \text{set}.\text{body} \\ & \{ \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \neg \text{res} \} \parallel \{ \langle a.\text{key} \rangle \} \end{aligned}$$

where we are using $?$ to indicate that this needs to be proven, and where we are using the shorthands

$$\begin{aligned} A_{buy} & \triangleq \text{this} : \text{Shop}, \text{buyer} : \text{external}, \text{anItem} : \text{Item}, \text{price} : \text{int}, \\ & \text{myAcnt} : \text{Account}, \text{oldBalance} : \text{int}, \text{newBalance} : \text{int}, \text{tmp} : \text{int}. \\ \text{Ids}_{buy} & \triangleq \text{this}, \text{buyer}, \text{anItem}, \text{price}, \text{myAcnt}, \text{oldBalance}, \text{newBalance}, \text{tmp}. \\ A_{trns} & \triangleq \text{this} : \text{Account}, \text{dest} : \text{Account}, \text{key}' : \text{Key}, \text{amt} : \text{nat} \\ \text{Ids}_{trns} & \triangleq \text{this}, \text{dest}, \text{key}', \text{amt} \\ A_{set} & \triangleq \text{this} : \text{Account}, \text{key}' : \text{Key}, \text{key}'' : \text{Key}. \\ \text{Ids}_{set} & \triangleq \text{this}, \text{key}', \text{key}'' . \end{aligned}$$

We will also need to prove that Send satisfies specifications S_{2a} and S_{2b} .

We outline the proof of (1?) in Lemma H.3, and the proof of (2) in Lemma H.4. We do not prove (3), but will prove that set from M_{fine} satisfies S_2 ; shown in Lemma H.5 – ie for module M_{fine} . \square

We also want to prove that M_{fine} satisfies the specification $S_{2, \text{strong}}$.

Lemma H.2 (M_{fine} satisfies $S_{2, \text{strong}}$). $M_{fine} \vdash S_{2, \text{strong}}$

PROOF OUTLINE The proof of

$$M_{fine} \vdash \forall a : \text{Account}. \{ \langle a.\text{key} \rangle \}$$

goes along similar lines to the proof of lemma H.1. Thus, we need to prove the following three Hoare quadruples:

$$\begin{aligned}
 (4?) \quad & M_{fine} \vdash \{ A_{buy}, a : \text{Account} \wedge \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \leftarrow \times \text{Ids}_{buy} \} \\
 & \text{Shop} :: \text{buy.body} \\
 & \{ \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \neg \text{res} \} \parallel \{ \langle a.\text{key} \rangle \} \\
 (5?) \quad & M_{fine} \vdash \{ A_{trns}, a : \text{Account} \wedge \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \leftarrow \times \text{Ids}_{trns} \} \\
 & \text{Account} :: \text{transfer.body} \\
 & \{ \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \neg \text{res} \} \parallel \{ \langle a.\text{key} \rangle \} \\
 (6?) \quad & M_{fine} \vdash \{ A_{set}, a : \text{Account} \wedge \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \leftarrow \times \text{Ids}_{set} \} \\
 & \text{Account} :: \text{set.body} \\
 & \{ \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \neg \text{res} \} \parallel \{ \langle a.\text{key} \rangle \}
 \end{aligned}$$

The proof of (4?) is identical to that of (1?); the proof of (5?) is identical to that of (2?). We outline the proof (6?) in Lemma H.5 in §H.2. □

Lemma H.3 ($\text{Shop} :: \text{buy}$ satisfies S_2).

$$\begin{aligned}
 (1) \quad & M_{good} \vdash \{ A_{buy} a : \text{Account} \wedge \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \leftarrow \times \text{Ids}_{buy} \} \\
 & \text{Shop} :: \text{buy.body} \\
 & \{ \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \neg \text{res} \} \parallel \{ \langle a.\text{key} \rangle \}
 \end{aligned}$$

PROOF OUTLINE We will use the shorthand $A_1 \triangleq A_{buy}, a : \text{Account}$. We will split the proof into 1) proving that statements 10, 11, 12 preserve the protection of $a.\text{key}$ from the buyer, 2) proving that the external call

1st Step: proving statements 10, 11, 12

We apply the underlying Hoare logic and prove that the statements on lines 10, 11, 12 do not affect the value of $a.\text{key}$, ie that for a $z \notin \{\text{price}, \text{myAcnt}, \text{oldBalance}\}$, we have

$$\begin{aligned}
 (10) \quad & M_{good} \vdash_{ul} \{ A_1 \wedge z = a.\text{key} \} \\
 & \text{price} := \text{anItem.price}; \\
 & \text{myAcnt} := \text{this.acnt}; \\
 & \text{oldBalance} := \text{myAcnt.blnc}; \\
 & \{ z = a.\text{key} \}
 \end{aligned}$$

We then apply EMBED_UL, PROT-1 and PROT-2 and COMBINE and and TYPES-2 on (10) and use the shorthand $\text{stmts}_{10,11,12}$ for the statements on lines 10, 11 and 12, and obtain:

$$\begin{aligned}
 (11) \quad & M_{good} \vdash \{ A_1 \wedge \langle a.\text{key} \rangle \wedge \langle \text{buyer} \rangle \leftarrow \times a.\text{key} \} \\
 & \text{stmts}_{10,11,12} \\
 & \{ \langle a.\text{key} \rangle \wedge \langle \text{buyer} \rangle \leftarrow \times a.\text{key} \}
 \end{aligned}$$

We apply MID on (11) and obtain

$$(12) \quad M_{good} \vdash \{ A_1 \wedge \langle a.key \rangle \Leftarrow \times buyer \} \\ \text{stmts}_{10,11,12} \\ \{ A_1 \wedge \langle a.key \rangle \wedge \langle buyer \rangle \Leftarrow \times a.key \} \parallel \\ \{ \langle a.key \rangle \}$$

2nd Step: Proving the External Call

We now need to prove that the external method call `buyer.pay(this.accnt, price)` protects the key. i.e.

$$(13?) \quad M_{good} \vdash \{ A_1 \wedge \langle a.key \rangle, \wedge \langle a.key \rangle \Leftarrow \times buyer \} \\ \text{tmp} := \text{buyer.pay}(\text{myAccnt}, \text{price}) \\ \{ A_1 \wedge \langle a.key \rangle \wedge \langle buyer \rangle \Leftarrow \times a.key \} \parallel \\ \{ \langle a.key \rangle \}$$

We use that $M \vdash \forall a : \text{Account}. \{ \langle a.key \rangle \}$ and obtain

$$(14) \quad M_{good} \vdash \{ \text{buyer} : \text{external}, \langle a.key \rangle \wedge \langle a.key \rangle \Leftarrow \times (\text{buyer}, \text{myAccnt}, \text{price}) \} \\ \text{tmp} := \text{buyer.pay}(\text{myAccnt}, \text{price}) \\ \{ \langle a.key \rangle \wedge \langle a.key \rangle \Leftarrow \times (\text{buyer}, \text{myAccnt}, \text{price}) \} \parallel \\ \{ \langle a.key \rangle \}$$

Moreover by the type declarations and the type rules, we obtain that all objects directly or indirectly accessible from `myAccnt` are internal or scalar. This, together with PROT-INTL, gives that

$$(15) \quad M_{good} \vdash A_1 \longrightarrow \langle a.key \rangle \Leftarrow \times \text{myAccnt}$$

Similarly, because `price` is a `nat`, and because of PROT-INT₁, we obtain

$$(16) \quad M_{good} \vdash A_1 \longrightarrow \langle a.key \rangle \Leftarrow \times \text{price}$$

We apply CONSEQU on (15), (16) and (14) and obtain (13)!

□

Lemma H.4 (`transfer` satisfies S_2).

$$(2) \quad M_{good} \vdash \{ A_{trns}, a : \text{Account} \wedge \langle a.key \rangle \wedge \langle a.key \rangle \Leftarrow \times \text{Ids}_{trns} \} \\ \text{Account} :: \text{transfer.body} \\ \{ \langle a.key \rangle \wedge \langle a.key \rangle \rightarrow \forall \text{res} \} \parallel \{ \langle a.key \rangle \}$$

PROOF OUTLINE

To prove (2), we will need to prove that

```

(21?)  $M_{good} \vdash \{ Atrns, a : \text{Account} \wedge \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \nleftrightarrow Idstrns \}$ 
    if (this.key==key') then
      this.blnc:=this.blnc-amt
      dest.blnc:=dest.blnc+amt
    else
      res:=0
    res:=0
     $\{ \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \neg \forall res \} \parallel \{ \langle a.\text{key} \rangle \}$ 

```

Using the underlying Hoare logic we can prove that no account's key gets modified, namely

```

(22)  $M_{good} \vdash_{ul} \{ Atrns, a : \text{Account} \wedge \langle a.\text{key} \rangle$ 
    if (this.key==key') then
      this.blnc:=this.blnc-amt
      dest.blnc:=dest.blnc+amt
    else
      res:=0
    res:=0
     $\{ \langle a.\text{key} \rangle \}$ 

```

Using (22) and [PROT-1], we obtain

```

(23)  $M_{good} \vdash \{ Atrns, a : \text{Account} \wedge z = a.\text{key} \}$ 
    if (this.key==key') then
      this.blnc:=this.blnc-amt
      dest.blnc:=dest.blnc+amt
    else
      res:=0
    res:=0
     $\{ z = a.\text{key} \}$ 

```

Using (23) and [EMBED-UL], we obtain

(24) $M_{good} \vdash \{ A_{trns}, a : \text{Account} \wedge z = a.\text{key} \}$
 if (this.key==key') then
 this.blnc:=this.blnc-amt
 dest.blnc:=dest.blnc+amt
 else
 res:=0
 res:=0
 {z = a.key} || {z = a.key}

[PROT_INT] and the fact that z is an int gives us that $\langle a.\text{key} \rangle \neg \text{res}$. Using [TYPES], and [PROT_INT] and [CONSEQU] on (24) we obtain (21?).

□

We want to prove that this public method satisfies the specification $S_{2, \text{strong}}$, namely

Lemma H.5 (set satisfies S_2).

(6) $M_{fine} \vdash \{ A_{set} \wedge \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \leftarrow \times \text{Ids}_{set} \}$
 if (this.key==key') then
 this.key:=key"
 else
 res:=0
 res:=0
 { $\langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \neg \text{res}$ } || { $\langle a.\text{key} \rangle$ }

PROOF OUTLINE We will be using the shorthand $A_2 \triangleq a : \text{Account}, A_{set}$.

To prove (6), we will use the SEQUENCE rule, and we want to prove

(61?) $M_{fine} \vdash \{ A_2 \wedge \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \leftarrow \times \text{Ids}_{set} \}$
 if (this.key==key') then
 this.key:=key"
 else
 res:=0
 { $A_2 \wedge \langle a.\text{key} \rangle$ } || { $\langle a.\text{key} \rangle$ }

and that

(62?) $M_{fine} \vdash \{ A_2 \wedge \langle a.\text{key} \rangle \}$
 res:=0
 { $\langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \neg \text{res}$ } || { $\langle a.\text{key} \rangle$ }

(62?) follows from the types, and PROT-INT₁, the fact that $a.\text{key}$ did not change, and PROT-1.

We now want to prove (61?). For this, will apply the IF-RULE. That is, we need to prove that

(63?) $M_{fine} \vdash \{ A_2 \wedge \langle a.key \rangle \wedge \langle a.key \rangle \Leftarrow Ids_{set} \wedge this.key = key' \}$
 $this.key := key''$
 $\{ \langle a.key \rangle \} \parallel \{ \langle a.key \rangle \}$

and that

(64?) $M_{fine} \vdash \{ A_2 \wedge \langle a.key \rangle \wedge \langle a.key \rangle \Leftarrow Ids_{set} \wedge this.key \neq key' \}$
 $res := 0$
 $\{ \langle a.key \rangle \} \parallel \{ \langle a.key \rangle \}$

(64?) follows easily from the fact that $a.key$ did not change, and PROT-1.

We look at the proof of (63?). We will apply the CASES rule, and distinguish on whether $a.key = this.key$. That is, we want to prove that

(65?) $M_{fine} \vdash \{ A_2 \wedge \langle a.key \rangle \wedge \langle a.key \rangle \Leftarrow Ids_{set} \wedge this.key = key' \wedge this.key = a.key \}$
 $this.key := key''$
 $\{ \langle a.key \rangle \} \parallel \{ \langle a.key \rangle \}$

and that

(66?) $M_{fine} \vdash \{ A_2 \wedge \langle a.key \rangle \wedge \langle a.key \rangle \Leftarrow Ids_{set} \wedge this.key = key' \wedge this.key \neq a.key' \}$
 $this.key := key''$
 $\{ \langle a.key \rangle \} \parallel \{ \langle a.key \rangle \}$

We can prove (65?) through application of ABSURD, PROTNEQ, and CONSEQU, as follows

(61c) $M_{fine} \vdash \{ false \}$
 $this.key := key''$
 $\{ \langle a.key \rangle \} \parallel \{ \langle a.key \rangle \}$

By PROTNEQ, we have $M_{fine} \vdash \langle a.key \rangle \Leftarrow key' \longrightarrow a.key \neq key'$, and therefore obtain

(61d) $M_{fine} \vdash \dots \wedge \langle a.key \rangle \Leftarrow Ids_{set} \wedge this.key = a.key \wedge this.key = key' \longrightarrow false$

We apply CONSEQU on (61c) and (61d) and obtain (61aa?).

We can prove (66?) by proving that $this.key \neq a.key$ implies that $this \neq a$ (by the underlying Hoare logic), which again implies that the assignment $this.key := \dots$ leaves the value of $a.key$ unmodified. We apply PROT-1, and are done.

□

H.3 Showing that M_{bad} does not satisfy S_2 nor S_3

H.3.1 M_{bad} does not satisfy S_2 . M_{bad} does not satisfy S_2 . We can argue this semantically (as in §H.3.2), and also in terms of the proof system (as in H.3.3).

H.3.2 $M_{bad} \not\models S_2$. The reason is that M_{bad} exports the public method `set`, which updates the key without any checks. So, it could start in a state where the key of the account was protected, and then update it to something not protected.

In more detail: Take any state σ , where $M_{bad}, \sigma \models a_0 : \text{Account}, k_0 : \text{Key} \wedge \langle a_0.\text{key} \rangle$. Assume also that $M_{bad}, \sigma \models \text{extl}$. Finally, assume that the continuation in σ consists of $a_0.\text{set}(k_0)$. Then we obtain that $M_{bad}, \sigma \rightsquigarrow^* \sigma'$, where $\sigma' = \sigma[a_0.\text{key} \mapsto k_0]$. We also have that $M_{bad}, \sigma' \models \text{extl}$, and because k_0 is a local variable, we also have that $M_{bad}, \sigma' \not\models \langle k_0 \rangle$. Moreover, $M_{bad}, \sigma' \models a_0.\text{krey} = k_0$. Therefore, $M_{bad}, \sigma' \not\models \langle a_0.\text{key} \rangle$.

H.3.3 $M_{bad} \not\models S_2$. In order to prove that $M_{bad} \vdash S_2$, we would have needed to prove, among other things, that `set` satisfied S_2 , which means proving that

```
(ERR_1?)   $M_{bad} \vdash \{ \text{this} : \text{Account}, k' : \text{Key}, a : \text{Account} \wedge \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \Leftarrow \{ \text{this}, k' \} \}$ 
            $\text{this}.\text{key} := k';$ 
            $\text{res} := 0$ 
            $\{ \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \Leftarrow \text{res} \} \parallel \{ \dots \}$ 
```

However, we cannot establish (ERR_1?). Namely, when we take the case where `this = a`, we would need to establish, that

```
(ERR_2?)   $M_{bad} \vdash \{ \text{this} : \text{Account}, k' : \text{Key} \wedge \langle \text{this}.\text{key} \rangle \wedge \langle \text{this}.\text{key} \rangle \Leftarrow \{ \text{this}, k' \} \}$ 
            $\text{this}.\text{key} := k'$ 
            $\{ \langle \text{this}.\text{key} \rangle \} \parallel \{ \dots \}$ 
```

And there is no way to prove (ERR_2?). In fact, (ERR_2?) is not sound, for the reasons outlined in §H.3.2.

H.3.4 M_{bad} does not satisfy S_3 . We have already argued in Examples 2.3 and 6.5 that M_{bad} does not satisfy S_3 , by giving a semantic argument – i.e we are in state where $\langle a_0.\text{key} \rangle$, and execute $a_0.\text{set}(k_1); a_0.\text{transfer}(\dots k_1)$.

Moreover, if we attempted to prove that `set` satisfies S_3 , we would have to show that

```
(ERR_3?)   $M_{bad} \vdash \{ \text{this} : \text{Account}, k' : \text{Key}, a : \text{Account}, b : \text{int} \wedge$ 
            $\langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \Leftarrow \{ \text{this}, k' \} \wedge a.\text{blnce} \geq b \}$ 
            $\text{this}.\text{key} := k';$ 
            $\text{res} := 0$ 
            $\{ \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \Leftarrow \text{res} \wedge a.\text{blnce} \geq b \} \parallel \{ \dots \}$ 
```

which, in the case of `a = this` would imply that

```
(ERR_4?)   $M_{bad} \vdash \{ \text{this} : \text{Account}, k' : \text{Key}, b : \text{int} \wedge$ 
            $\langle \text{this}.\text{key} \rangle \wedge \langle \text{this}.\text{key} \rangle \Leftarrow \{ \text{this}, k' \} \wedge \text{this}.\text{blnce} \geq b \}$ 
            $\text{this}.\text{key} := k'$ 
            $\{ \langle \text{this}.\text{key} \rangle \} \parallel \{ \dots \}$ 
```

And (ERR_4?) cannot be proven and does not hold.

H.4 Demonstrating that $M_{good} \vdash S_3$, and that $M_{fine} \vdash S_3$

H.5 Extending the specification S_3

As in §H.2, we redefine S_3 so that it also describes the behaviour of method `send`. and have:

$$S_{3,strong} \triangleq S_3 \wedge S_{2a} \wedge S_{2b}$$

Lemma H.6 (module M_{good} satisfies $S_{3,strong}$). $M_{good} \vdash S_{3,strong}$

PROOF OUTLINE In order to prove that

$$M_{good} \vdash \forall a : \text{Account}, b : \text{int}. \{ \langle a.\text{key} \rangle \wedge a.\text{blnce} \geq b \}$$

we have to apply INVARIANT from Fig. 9. That is, for each class C defined in M_{good} , and for each public method m in C , with parameters $\overline{y} : \overline{D}$, we have to prove that they satisfy the corresponding quadruples.

Thus, we need to prove three Hoare quadruples: one for `Shop :: buy`, one for `Account :: transfer`, and one for `Account :: set`. That is, we have to prove that

$$\begin{aligned} (31?) \quad & M_{good} \vdash \{ A_{buy}, a : \text{Account}, b : \text{int} \wedge \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \Leftarrow \text{Ids}_{buy} \wedge a.\text{blnce} \geq b \} \\ & \quad \text{Shop} :: \text{buy.body} \\ & \quad \{ \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \neg \text{res} \wedge a.\text{blnce} \geq b \} \parallel \{ \langle a.\text{key} \rangle \wedge a.\text{blnce} \geq b \} \\ (32?) \quad & M_{good} \vdash \{ A_{trns}, a : \text{Account}, b : \text{int} \wedge \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \Leftarrow \text{Ids}_{trns} \wedge a.\text{blnce} \geq b \} \\ & \quad \text{Account} :: \text{transfer.body} \\ & \quad \{ \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \neg \text{res} \wedge a.\text{blnce} \geq b \} \parallel \{ \langle a.\text{key} \rangle \wedge a.\text{blnce} \geq b \} \\ (33?) \quad & M_{good} \vdash \{ A_{set}, a : \text{Account}, b : \text{int} \wedge \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \Leftarrow \text{Ids}_{set} \wedge a.\text{blnce} \geq b \} \\ & \quad \text{Account} :: \text{set.body} \\ & \quad \{ \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \neg \text{res} \wedge a.\text{blnce} \geq b \} \parallel \{ \langle a.\text{key} \rangle \wedge a.\text{blnce} \geq b \} \end{aligned}$$

where we are using $?$ to indicate that this needs to be proven, and where we are using the shorthands A_{buy} , Ids_{buy} , A_{trns} , Ids_{trns} , A_{set} as defined earlier.

□

The proofs for M_{fine} are similar.

We outline the proof of (31?) in Lemma H.7. We outline the proof of (32?) in Lemma H.8.

H.5.1 Proving that `Shop :: buy` from M_{good} satisfies $S_{3,strong}$ and also S_4 .

Lemma H.7 (function $M_{good} :: \text{Shop} :: \text{buy}$ satisfies $S_{3,strong}$ and also S_4).

$$\begin{aligned} (31) \quad & M_{good} \vdash \{ A_{buy}, a : \text{Account}, b : \text{int}, \wedge \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \Leftarrow \text{Ids}_{buy} \wedge a.\text{blnce} \geq b \} \\ & \quad \text{Shop} :: \text{buy.body} \\ & \quad \{ \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \neg \text{res} \wedge a.\text{blnce} \geq b \} \parallel \{ \langle a.\text{key} \rangle \wedge a.\text{blnce} \geq b \} \end{aligned}$$

PROOF OUTLINE Note that (31) is a proof that $M_{good} :: \text{Shop} :: \text{buy}$ satisfies $S_{3,strong}$ and also hat $M_{good} :: \text{Shop} :: \text{buy}$ satisfies S_4 . This is so, because application of [METHOD] on S_4 gives us exactly the proof obligation from (31).

This proof is similar to the proof of lemma H.3, with the extra requirement here that we need to argue about balances not decreasing. To do this, we will leverage the assertion about balances given in S_3 .

We will use the shorthand $A_1 \triangleq A_{buy}, a : \text{Account}, b : \text{int}$. We will split the proof into 1) proving that statements 10, 11, 12 preserve the protection of `a.key` from the buyer, 2) proving that the external call

1st Step: proving statements 10, 11, 12

We apply the underlying Hoare logic and prove that the statements on lines 10, 11, 12 do not affect the value of `a.key` nor that of `a.blnc`. Therefore, for a $z, z' \notin \{\text{price}, \text{myAcnt}, \text{oldBalance}\}$, we have

(40) $M_{good} \vdash_{ul} \{ A_1 \wedge z = a.key \wedge z' = a.blnc \}$
 $\quad \text{price} := \text{anItem.price};$
 $\quad \text{myAccnt} := \text{this.accnt};$
 $\quad \text{oldBalance} := \text{myAccnt.blnc};$
 $\quad \{ z = a.key \wedge z' = a.blnc \}$

We then apply EMBED_UL, PROT-1 and PROT-2 and COMBINE and and TYPES-2 on (10) and use the shorthand $\text{stmts}_{10,11,12}$ for the statements on lines 10, 11 and 12, and obtain:

(41) $M_{good} \vdash \{ A_1 \wedge \langle a.key \rangle \wedge \langle \text{buyer} \rangle \Leftarrow a.key \wedge z' = a.blnc \}$
 $\quad \text{stmts}_{10,11,12}$
 $\quad \{ \langle a.key \rangle \wedge \langle \text{buyer} \rangle \Leftarrow a.key \wedge z' = a.blnc \}$

We apply MID on (11) and obtain

(42) $M_{good} \vdash \{ A_1 \wedge \langle a.key \rangle \Leftarrow \text{buyer} \wedge z' = a.blnc \}$
 $\quad \text{stmts}_{10,11,12}$
 $\quad \{ A_1 \wedge \langle a.key \rangle \wedge \langle \text{buyer} \rangle \Leftarrow a.key \wedge z' = a.blnc \} \parallel$
 $\quad \{ \langle a.key \rangle \wedge z' = a.blnc \}$

2nd Step: Proving the External Call

We now need to prove that the external method call $\text{buyer.pay}(\text{this.accnt}, \text{price})$ protects the key, and does not decrease the balance, i.e.

(43?) $M_{good} \vdash \{ A_1 \wedge \langle a.key \rangle \wedge \langle a.key \rangle \Leftarrow \text{buyer} \wedge z' = a.blnc \}$
 $\quad \text{tmp} := \text{buyer.pay}(\text{myAccnt}, \text{price})$
 $\quad \{ A_1 \wedge \langle a.key \rangle \wedge \langle \text{buyer} \rangle \Leftarrow a.key \wedge a.blnc \geq z' \} \parallel$
 $\quad \{ \langle a.key \rangle \wedge a.blnc \geq z' \}$

We use that $M \vdash \forall a : \text{Account}, b : \text{int}, \{ \langle a.key \rangle \wedge a.blnc \geq z' \}$ and obtain

(44) $M_{good} \vdash \{ \text{buyer} : \text{external}, \langle a.key \rangle \wedge \langle a.key \rangle \Leftarrow (\text{buyer}, \text{myAccnt}, \text{price}) \wedge z' \geq a.blnc \}$
 $\quad \text{tmp} := \text{buyer.pay}(\text{myAccnt}, \text{price})$
 $\quad \{ \langle a.key \rangle \wedge \langle a.key \rangle \Leftarrow (\text{buyer}, \text{myAccnt}, \text{price}) \wedge z' \geq a.blnc \} \parallel$
 $\quad \{ \langle a.key \rangle \wedge z' \geq a.blnc \}$

In order to obtain (43?) out of (44), we use the type system and type declarations and obtain that all objects transitively reachable from myAccnt or price are scalar or internal. Thus, we apply PROT-INTL, and obtain

(45) $M_{good} \vdash A_1 \wedge \langle a.key \rangle \longrightarrow \langle a.key \rangle \Leftarrow \text{myAccnt}$
(46) $M_{good} \vdash A_1 \wedge \langle a.key \rangle \longrightarrow \langle a.key \rangle \Leftarrow \text{price}$
(47) $M_{good} \vdash A_1 \wedge z' = a.blnc \longrightarrow z' \geq a.blnc$

We apply CONSEQU on (44), (45), (46) and (47) and obtain (43)!

3rd Step: Proving the Remainder of the Body

We now need to prove that lines 15-19 of the method preserve the protection of $a.key$, and do not decrease $a.balance$. We outline the remaining proof in less detail.

We prove the internal call on line 16, using the method specification for `send`, using S_{2a} and S_{2b} , and applying rule [CALL_INT], and obtain

$$(48) \quad M_{good} \vdash \{ \text{buyer} : \text{external}, \text{item} : \text{Item} \wedge \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \Leftarrow^* (\text{buyer} \wedge z' = a.\text{blnce}) \} \\ \quad \text{tmp} := \text{this.send}(\text{buyer}, \text{Item}) \\ \quad \{ \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \Leftarrow^* \text{buyer} \wedge z' = a.\text{blnce} \} \parallel \\ \quad \{ \langle a.\text{key} \rangle \wedge z' = a.\text{blnce} \}$$

We now need to prove that the external method `callbuyer.tell("You have not paid me")` also protects the key, and does not decrease the balance. We can do this by applying the rule about protection from strings [PROR_STR], the fact that $M_{good} \vdash S_3$, and rule [CALL_EXTL_ADAPT] and obtain:

$$(49) \quad M_{good} \vdash \{ \text{buyer} : \text{external}, \text{item} : \text{Item} \wedge \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \Leftarrow^* (\text{buyer} \wedge z' / \geq a.\text{blnce}) \} \\ \quad \text{tmp} := \text{buyer.tell}(\text{"You have not paid me"}) \\ \quad \{ \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \Leftarrow^* \text{buyer} \wedge z' \geq a.\text{blnce} \} \parallel \\ \quad \{ \langle a.\text{key} \rangle \wedge z' \geq a.\text{blnce} \}$$

We can now apply [IF_RULE], and [CONSEQ] on (49) and (50), and obtain

$$(50) \quad M_{good} \vdash \{ \text{buyer} : \text{external}, \text{item} : \text{Item} \wedge \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \Leftarrow^* (\text{buyer} \wedge z' \geq a.\text{blnce}) \} \\ \quad \text{if...then} \\ \quad \text{tmp} := \text{this.send}(\text{buyer}, \text{anItem}) \\ \quad \text{else} \\ \quad \text{tmp} := \text{buyer.tell}(\text{"You have not paid me"}) \\ \quad \{ \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \Leftarrow^* \text{buyer} \wedge z' \geq a.\text{blnce} \} \parallel \\ \quad \{ \langle a.\text{key} \rangle \wedge z' \geq a.\text{blnce} \}$$

The rest follows through application of [PROT_INT], and [SEQ].

□

Lemma H.8 (function $M_{good} :: \text{Account} :: \text{transfer}$ satisfies S_3).

$$(32) \quad M_{good} \vdash \{ A_{trns}, a : \text{Account}, b : \text{int} \wedge \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \Leftarrow^* \text{Ids}_{trns} \wedge a.\text{blnce} \geq b \} \\ \quad \text{Account} :: \text{transfer.body} \\ \quad \{ \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \Leftarrow^* \neg \text{res} \wedge a.\text{blnce} \geq b \} \parallel \{ \langle a.\text{key} \rangle \wedge a.\text{blnce} \geq b \}$$

PROOF OUTLINE We will use the shorthand $stmts_{28-33}$ for the statements in the body of `transfer`. We will prove the preservation of protection, separately from the balance not decreasing when the key is protected.

For the former, applying the steps in the proof of Lemma H.4, we obtain

$$(21) \quad M_{good} \vdash \{ A_{trns}, a : \text{Account} \wedge \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \Leftarrow^* \text{Ids}_{trns} \} \\ \quad stmts_{28-33} \\ \quad \{ \langle a.\text{key} \rangle \wedge \langle a.\text{key} \rangle \Leftarrow^* \neg \text{res} \} \parallel \{ \langle a.\text{key} \rangle \}$$

For the latter, we rely on the underlying Hoare logic to ensure that no balance decreases, except perhaps that of the receiver, in which case its key was not protected. Namely, we have that

$$(71) \quad M_{good} \vdash l \{ \text{Atrns}, a : \text{Account} \wedge a.\text{blnce} = b \wedge (\text{this} \neq a \vee \text{prgthis.key} \neq \text{key}') \} \\ \text{stmts}_{28-33} \\ \{a.\text{blnce} \geq b\}$$

We apply rules EMBED_UL and MID on (71), and obtain

$$(72) \quad M_{good} \vdash \{ \text{Atrns}, a : \text{Account} \wedge a.\text{blnce} = b \wedge (\text{this} \neq a \vee \text{prgthis.key} \neq \text{key}') \} \\ \text{stmts}_{28-33} \\ \{a.\text{blnce} \geq b\} \parallel \{a.\text{blnce} \geq b\}$$

Moreover, we have

$$(73) \quad M_{good} \vdash \langle a.\text{key} \rangle \leftarrow^* \text{Ids}_{trns} \rightarrow \langle a.\text{key} \rangle \leftarrow^* \text{key}' \\ (74) \quad M_{good} \vdash \langle a.\text{key} \rangle \leftarrow^* \text{key}' \rightarrow a.\text{key} \neq \text{key}' \\ (75) \quad M_{good} \vdash a.\text{key} \neq \text{key}' \rightarrow a \neq \text{this} \vee \text{this.key} \neq \text{key}'$$

normalize

Applying (73), (74), (75) and CONSEQ on (72) we obtain:

$$(76) \quad M_{good} \vdash \{ \text{Atrns}, a : \text{Account} \wedge a.\text{blnce} = b \wedge \langle a.\text{key} \rangle \leftarrow^* \text{Ids}_{trns} \} \\ \text{stmts}_{28-33} \\ \{a.\text{blnce} \geq b\} \parallel \{a.\text{blnce} \geq b\}$$

We combine (72) and (76) through COMBINE and obtain (32). □

H.6 Dealing with polymorphic function calls

The case split rules together with the rule of consequence allow our Hoare logic to formally reason about polymorphic calls, where the receiver may be internal or external.

We demonstrate this through an example where we may have an external receiver, or a receiver from a class C . Assume we had a module M with a scoped invariant (as in A), and an internal method specification as in (B).

$$(A) \quad M \vdash \forall y_1 : D. \{A\} \\ (B) \quad M \vdash \{A_1\} \text{p } C::m(y_1 : D) \{A_2\} \parallel \{A_3\}$$

Here p may be `private` or `public`; the argument applies either way.

Assume also implications as in (C)-(H)

$$(C) \quad M \vdash A_0 \rightarrow A \neg(y_0, y_1) \\ (D) \quad M \vdash A \neg(y_0, y_1) \rightarrow A_4 \\ (E) \quad M \vdash A \rightarrow A_5 \\ (F) \quad M \vdash A_0 \rightarrow A_1[y_0/\text{this}] \\ (G) \quad M \vdash A_2[y_0, u/\text{this}, \text{res}] \rightarrow A_4 \\ (H) \quad M \vdash A_3 \rightarrow A_5$$

Then, by application of CALL_EXT_ADAPT on (A) we obtain (I)

$$(I) \quad M \vdash \{ y_0 : \text{external}, y_1 : D \wedge A \neg(y_0, y_1) \} u := y_0.m(y_1) \{ A \neg(y_0, y_1) \} \parallel \{ A \}$$

By application of the rule of consequence on (I) and (C), (D), and (E), we obtain

$$(J) \quad M \vdash \{ y_0 : \text{external}, y_1 : D \wedge A_0 \} u := y_0.m(y_1) \{ A_4 \} \parallel \{ A_5 \}$$

Then, by application of [CALL_INTL] on (B) we obtain (K)

(K) $M \vdash \{ y_0 : C, y_1 : D \wedge A_1[y_0/\text{this}] \} u := y_0.m(y_1) \{ A_2[y_0, u/\text{this}, \text{res}] \} \parallel \{ A_3 \}$

By application of the rule of consequence on (K) and (F), (G), and (H), we obtain

(L) $M \vdash \{ y_0 : C, y_1 : D \wedge A_0 \} u := y_0.m(y_1) \{ A_4 \} \parallel \{ A_5 \}$

By case split, [CASES], on (J) and (L), we obtain

(polymorphic) $M \vdash \{ (y_0 : \text{external} \vee y_0 : C), y_1 : D \wedge A_0 \} u := y_0.m(y_1) \{ A_4 \} \parallel \{ A_5 \}$