Reasoning about External Calls

ANONYMOUS AUTHOR(S)

In today's complex software, internal, trusted, code is tightly intertwined with external, untrusted, code. By definition, internal code does not trust external code. From an internal perspective, the effects of outgoing calls to external code – *external calls* — are necessarily unknown and unlimited.

Nevertheless, the effects of external calls can be *tamed* if internal code is programmed defensively, *i.e.* to ensure particular effects cannot happen. Tamed effects allow us to prove that internal code preserves assertions about internal and external objects, even in the presence of outgoing calls and callbacks.

This paper addresses the specification and verification of internal code that makes external calls, using encapsulation and object capabilities to tame effects. We propose new assertions for access to capabilities, new specifications for tamed effects, and a Hoare logic to verify that a module satisfies its tamed effects specification, even while making external calls. We illustrate the approach though 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. In today's complex software, internal, trusted, code is tightly intertwined with external, untrusted, code: external code calls into internal code, internal code calls out to external code and external code even calls 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. External code may even have been written by an attacker trying to subvert or destroy the whole system.

In this code sketch, method `m1`'s code is trusted, method `m2` takes an untrusted parameter `untrst`, and then at line 6 calls an unknown external method `unkn` passing itself as an argument. The challenge is: what can that method call do? what effects will it 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) //external call
7     ... trusted code ...
```

Tamed effects. In practice, not all external calls will have unlimited effects. If the programming language supports encapsulation (*e.g.* no address forging, private fields, *etc.*) then internal modules can be written *defensively* [76], to ensure that external calls have only limited effects on internal components. For example, a defensive implementation of the DAO [23] can ensure that (a) no external object can cause the DAO's balance to fall below the sum of the balances of its subsidiary accounts, and (b) no external object can cause reduction of the balance of the DAO unless the causing object is one of the DAO's account holders.

We say a module has *tamed an effect*, when no outgoing external call can trigger that effect. While the literature has explored external calls [41, 63, 102, 105], "robust safety" [1, 42, 89], *etc.*, to our knowledge, there is no widely accepted term limiting the range of effects resulting from external calls. Tamed effects help mitigate the uncertainty associated with outgoing external calls. With tamed effects, we can ensure that specified properties established before an outgoing external call will be satisfied afterward.

2025. ACM 2475-1421/2025/1-ART
https://doi.org/

Taming of effects may be *conditional* or *unconditional*. For example, (a) is unconditional: the balance of the DAO is always, unconditionally, kept above the sum of the balances of its accounts. On the other hand, (b) is conditional: reduction is possible, but only if the causing, external, object is an account holder. Reasoning about unconditional taming of effects typically requires only an adaptation of techniques from object invariants [8, 36, 60, 83, 101]. Reasoning about conditional taming of effects requires rather more — hence the topic of this paper.

Effects tamed by capabilities. To tame effects in their code, programmers rely on various kinds of encapsulation – *e.g.* if address forging were possible, the range of potential effects from external calls would be unlimited. In addition, to conditionally tame effects, programmers often employ the object capability model (OCAP)[76] – or capability model for short. Capabilities are transferable rights that allow the performance of one or more operations on a specific object. They are *necessary* conditions for causing effects; callers can only produce effects if they possess the required capabilities. For example, a signatory can withdraw funds from a DAO only if they hold a "withdraw" capability for that specific account within that particular DAO.

Our remit: Specification and Verification for tamed effects. In this paper we demonstrate how to reason about code which tames effects, including those tamed by capabilities. We can specify effects to be tamed, and then we can *prove* that a module has indeed tamed the effects we've specified.

Recent work has developed logics to prove properties of programs employing object capabilities. Swasey et al. [102] develop a logic to prove that code employing object capabilities for encapsulation preserves invariants for intertwined code, but without external calls. Devriese et al. [30] can describe and verify invariants about multi-object structures and the availability and exercise of object capabilities. Similarly, Liu et al. [63] propose a separation logic to support formal modular reasoning about communicating VMs, including in the presence of unknown VMs. Rao et al. [92] specify WASM modules, and prove that adversarial code can only affect other modules through the functions that they explicitly export. Cassez et al. [21] handle external calls by replacing them through an unbounded number of calls to the module's public methods.

The approaches above do not aim to support indirect, eventual access to capabilities. Drossopoulou et al. [37] and Mackay et al. [67] do describe such access; 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 such specifications. Rather than relying on problem-specific, custom-made proofs, we propose a Hoare logic that addresses access to capabilities, external calls, and the module's tamed effects.

This paper's contributions. (1) assertions to describe access to capabilities, (2) a specification language to describe taming of effects, 3) a Hoare logic to reason about external calls and to prove that modules satisfy their tamed effects 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 contains essential concepts for our study. Sect. 5 and Sect 7 give syntax and semantics of assertions, and specifications, while Sect. 6 discusses preservation of satisfaction of assertions. Sect. 8 develops Hoare triples and quadruples to prove external calls, and that a module adheres to its tamed effects specifications. Sect. 9 outlines our proof of soundness of the Hoare logic. Sect. 10 summarises the Coq proof of our running example (the source code will be submitted as an artefact). Sect. 11 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 language similar to Joe-E [73] 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 $\overline{M}$. `Private` methods may only 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 types. [1]

We are concerned with guarantees made in an *open* setting; that is, our internal module M must be programmed so that its execution, together with any external modules $\overline{M}$ will satisfy these guarantees. M must ensure these guarantees are satisfied whenever the $\overline{M}$ *external* modules are executing, yet without relying on any assumptions about $\overline{M}$'s code (beyond the programming language's semantics)[2]. The internal module may break these guarantees temporarily, so long as they are reestablished before (re)entry to an external module.

`Shop` – illustrating tamed effects

The challenge when calling a method on an external object, is that we have no specification for that method. For illustration, consider the following, internal, module M_{shop}, and assume that it includes the classes `Item`, `Shop`, `Account`, and `Inventory`. Classes `Inventory` and `Item` have the expected functionality. `Account`s hold a balance and have a key. With access to an `Account`, one can pay money into it, and with access to an account and its key, one can withdraw money from it. Implementations of such a class appear in the next section. `Shop` has a public method `buy` whose formal parameter `buyer` is an `external` object.

```
1  module M_shop
2    ...
3    class Shop
4      field accnt:Account, invntry:Inventory, clients:[external]
5      public method buy(buyer:external, anItem:Item)
6        int price = anItem.price
7        int oldBlnce = this.accnt.blnce
8        buyer.pay(this.accnt, price)     // external call!
9        if (this.accnt.blnce == oldBlnce+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 critical point 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 its 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? The `Shop` hopes, but cannot be sure, that at line 9 it will have received money; but it wants to be certain that the `buyer` can not use this opportunity to access the shop's account to drain its money. Can `Shop` be certain?

[1] As 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 [73].

[2] This is a critical distinction from e.g. cooperative approaches such as rely/guarantee [46, 104].

(A) If prior to the call of `buy`, the `buyer` has no (eventual) access to the account's key, and
(B) If 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 has eventual access to the account's key, then
(C) The external call on line 8 will not 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. In that example, we relied on two yet-to-be-defined concepts: (A) "eventual access" and (B) tamed effects (e.g., no money withdrawn unless certain conditions are met). Therefore, we need to address the following three challenges:

1^{st} Challenge The specification of "eventual access".
2^{nd} Challenge The specification of tamed effects,
3^{rd} Challenge A Hoare Logic for external calls, and for adherence to tamed effect specifications.

2.1 1^{st} Challenge: eventual access

Assume we have a guarantee that no external object o_e can cause an effect E unless it has direct access to the capability object o_{cap}. To ensure o_e will not cause E, we must ensure that o_e not only lacks access now but also cannot gain access in the future. We call this "lack of eventual access."

We approximate "lack of eventual access" through *protection*, defined befow, and illustrated in Fig. 1.

Protection Object o is *protected from* o', formally $\langle o \rangle \leftarrow\!\!\!\!\times\, o'$, if the penultimate object on any path from o to o' is internal. Object o is *protected*, formally $\langle o \rangle$, if o is protected from all external objects transitively accessible from the currently executing method call. – *c.f.* Def. 5.4.

If the internal module never passes o to any external object (*i.e.* never leaks o,) then if o is protected from o_e now, it will remain protected from o_e, and if o is protected now, it will remain protected. Going back to the original question, if o_e is protected from all locally accessible external objects, effect E is guaranteed not to occur.

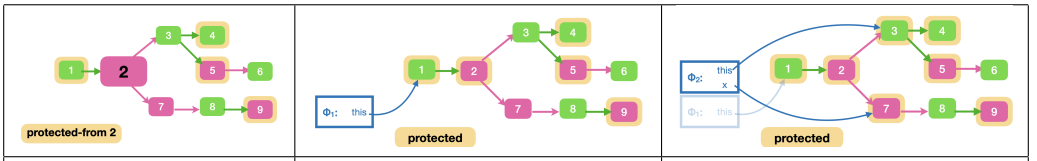


Fig. 1. Protected from and Protected. Pink and green squares are external and internal objects, respectively. Connecting straight arrows indicate fields. Blue boxes are frames on the stack. Protected objects are highlighted in yellow. The left pane shows a heap with objects o_1-o_9. Here o_1, o_4, o_5 and o_9 are protected from o_2. The middle pane shows the same heap and a stack with frame ϕ_1 whose receiver, `this`, points to o_1. Here o_1, o_2, o_4, o_5 and o_9 are protected. The right pane shows the same configuration, but after pushing frame ϕ_2, whose receiver, `this`, and local variable, `x`, point to o_3 and o_7, respectively. Here o_3 and o_7 are protected, in addition to the objects protected in the previous pane.

2.2 2^{nd} Challenge: Specification of tamed effects

How can we express the guarantee that effects are tamed? In particular, when such effects can be the outcome of the execution of more than one method? Traditional, per-method PRE/POST conditions cannot guarantee that two or more of our methods won't interact to produce an untamed effect. We build on the concept of history invariants [26, 61, 62] and define

Scoped invariants $\forall \overline{x : C}.\{A\}$ expresses that if a state σ has objects $\overline{x}$ of class $\overline{C}$, and satisfies A, then all σ's *scoped future states* will also satisfy A. The scoped future contains all states which can be reached through any steps, including further method calls and returns, but stopping before returning from the call active in σ [3] – *c.f.* Def 4.2. For σ and its scoped future we only consider external states – *c.f.* Def 7.5.

Example 2.1. The following scoped invariants

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

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

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

Scoped invariants are *conditional*: They ensure that assertions are *preserved*, but unlike object invariants, they do not guarantee that they always hold. *E.g.*, `buy` cannot assume $\langle a.\mathtt{key} \rangle$ holds on entry, but guarantees that if it holds on exit.

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

$S_4 \triangleq \{ \langle \mathtt{this.accnt.key} \rangle \not\leftrightarrow \mathtt{buyer} \wedge \mathtt{this.accnt.blnce} = b \}$
`public Shop :: buy(buyer : external, anItem : Item)`
$\{ \mathtt{this.accnt.blnce} \geq b \}$

S_4 guarantees that if the key was protected from `buyer` before the call, then the balance will not decrease. It does *not* guarantee `buy` will only be called when $\langle \mathtt{this.accnt.key} \rangle \not\leftrightarrow \mathtt{buyer}$ 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 of a class `Account` from [67] as part of our internal module M_{shop}. To differentiate, we rename M_{shop} as $\mathtt{M}_{good}$, $\mathtt{M}_{bad}$, or $\mathtt{M}_{fine}$. All use the same `transfer` method for withdrawing money.

```
1 module M_good
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:int)
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 $\mathtt{M}_{bad}$ and $\mathtt{M}_{fine}$ which differ from $\mathtt{M}_{good}$ only in their `set` methods. Whereas $\mathtt{M}_{good}$'s key is immutable, $\mathtt{M}_{bad}$ allows any client to reset an account's key at any time, and $\mathtt{M}_{fine}$ requires the existing key in order to change it.

```
1 M_bad
2 public method set(key':Key)
3   this.key=key'
```

```
1 M_fine
2 public method set(key',key'':Key)
3   if (this.key==key')  this.key=key''
```

Thus, in all three modules, the key is an object capability which *enables* the withdrawal of the money. Moreover, in $\mathtt{M}_{good}$ and $\mathtt{M}_{fine}$, the key is a capability used to *tame* withdrawal of money, preventing those without it from getting the money from the account. Crucially, in $\mathtt{M}_{bad}$ the key *does not tame* withdrawal of money. Using $\mathtt{M}_{bad}$, 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)
```

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

is enough to drain `acc` in $\texttt{M}_{bad}$ without knowing the key. Even though `transfer` in $\texttt{M}_{bad}$ is "safe" when considered in isolation, it is not safe when considered in conjunction with other methods from the same module.

Modules $\texttt{M}_{good}$ and $\texttt{M}_{fine}$ satisfy S_2 and S_3, while $\texttt{M}_{bad}$ satisfies neither S_2 nor S_3. No module satisfies S_1.

2.3 3^{rd} Challenge: A Hoare logic

We now move to the verification of S_4. The challenge is how to reason about the external call on line 8. We aim to establish a Hoare triple of the form:

$$\{\ \texttt{buyer}:\texttt{extl} \wedge \langle\texttt{this.accnt.key}\rangle \leftrightarrow\!\!\!\!\times\ \texttt{buyer} \wedge \texttt{this.accnt.blnce} = b\ \}$$
$$\texttt{buyer.pay(this.accnt,price)}$$
$$\{\ \texttt{this.accnt.blnce} \geq b\ \}$$

The intuitive reasoning is as follows: if the shop's account's key is protected from `buyer` (A from earlier), and the module satisfies S_3 (B), then after the call, the account's balance will not decrease (C). However, application of S_3 is not straightforward. It requires $\langle\texttt{a.key}\rangle \wedge \ldots$, but the call's precondition only guarantees $\langle\texttt{this.accnt.key}\rangle \leftrightarrow\!\!\!\!\times\ \texttt{buyer}$.

While we do not know whether `a.key` is protected during execution of `buy`[4], we can be certain it is protected during execution of `pay`. This is so, because the objects accessible during `pay` are those visible from its arguments (*i.e.* `buyer` and `price`).

We define the adaptation operator $\text{–}\triangledown$, which translates an assertion from the viewpoint of the called function to that of the caller. Specifically, $A\text{–}\triangledown\overline{y}$ ensures that A holds when the variables $\overline{y}$ (where $\overline{y}$ stands for $y_1, \ldots, y_n$) have been pushed onto a new frame. For example, $(\langle e\rangle)\text{–}\triangledown\overline{y} = \langle e\rangle \leftrightarrow\!\!\!\!\times\ \overline{y}$ for any term e (see Def. 6.5 and Lemma 6.6). In this case, we have:

$$(\langle\texttt{this.accnt.key}\rangle)\text{–}\triangledown\{\texttt{buyer},\texttt{price}\} = \langle\texttt{this.accnt.key}\rangle \leftrightarrow\!\!\!\!\times\ (\texttt{buyer},\texttt{price}).$$

and with this, we *can* apply S_3. Below a Hoare logic rule dealing with external calls - *c.f.*. Fig.7.

$$\frac{\forall\overline{x:D}.\{A\}\ \text{ is part of } M\text{'s specification}}{\{\ y_0:\texttt{extl} \wedge \overline{x:D} \wedge A\text{–}\triangledown(y_0,\overline{y})\ \}\ u := y_0.m(\overline{y})\ \{\ A\text{–}\triangledown(y_0,\overline{y})\ \}\ \ldots}$$

To develop our logic, we take a Hoare logic which does not have the concept of protection. We extend it through rules talking about protection, and internal and external calls – *c.f.* Figs. 6 - 7. A module is well-formed, if its invariants are well-formed, its public methods preserve its invariants, and all methods satisfy their specifications - *c.f.* Fig. 8. An invariant is well-formed if it is *encapsulated*, *i.e.* can only be invalidated by internal code – *c.f.* Def. 6.10. A method preserves an assertion if it preserves it from pre- to post-states and also in any intermediate external state. Our extension preserves soundness of the Hoare logic – *c.f.* Thms. 9.2, 9.3.

Summary

In our threat model, external objects can execute arbitrary code, invoke any public internal methods, potentially access any other external object, and may collude with one another in any conceivable way. Our specifications are conditional: they do not guarantee that certain effects will never occur, but they ensure that specific effects will only happen if certain conditions were met prior to the execution of the external code.

The key ingredients of our work are: a) the concepts of protection ($\langle x\rangle \leftrightarrow\!\!\!\!\times\ y$ and $\langle x\rangle$), b) scoped invariants ($\forall\overline{x:D}.\{A\}$), and c) the adaptation operator ($\text{–}\triangledown$). In the remaining sections, we discuss all this in more detail.

[4]For instance, one of the clients may have access to it.

3 THE UNDERLYING PROGRAMMING LANGUAGE $\mathscr{L}_{ul}$

3.1 $\mathscr{L}_{ul}$ syntax and runtime configurations

This work is based on $\mathscr{L}_{ul}$, a small, 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. Wrt to encapsulation and capability safety, $\mathscr{L}_{ul}$ supports private fields, private and public methods, unforgeable addresses, and no ambient authority (no static methods, no address manipulation). It has a simple concept of module with module-private fields and methods, described in Sect. 3.2. The definition of $\mathscr{L}_{ul}$ can be found in Appendix A.[5].

A $\mathscr{L}_{ul}$ state, σ, consists of a heap χ, and a stack. A stack is a sequence of frames, $\phi_1 \cdot \ldots \cdot \phi_n$. A frame, ϕ, consists of a local variable map and a continuation, *i.e.* a sequence of statements to be executed. The top frame in a state $(\phi_1 \cdot \ldots \cdot \phi_n, \chi)$ 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, α, α', α_1, ... when more formal.
- x, x', y, z, u, v, w are variables.
- $\alpha \in \sigma$ means that α is defined in the heap of σ, and $x \in \sigma$ means that x is defined in the top frame of σ. Conversely, $\alpha \notin \sigma$ and $x \notin \sigma$ have the obvious meanings. $\lfloor \alpha \rfloor_\sigma$ is α; and $\lfloor x \rfloor_\sigma$ is the value to which x is mapped in the top-most frame of σ's stack, and $\lfloor e.f \rfloor_\sigma$ looks up in σ's heap the value of f for the object $\lfloor e \rfloor_\sigma$.
- $\phi[x \mapsto \alpha]$ 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, ... 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.
- $\phi.\texttt{cont}$ is the continuation of frame ϕ, and $\sigma.\texttt{cont}$ is the continuation in the top frame.
- $text_1 \overset{\text{txt}}{=} text_2$ expresses that $text_1$ and $text_2$ are textually equal.
- We define the depth of a stack as $|\phi_1 \ldots \phi_n| \triangleq n$, For states, $|(\overline{\phi}, \chi)| \triangleq |\overline{\phi}|$. The operator $\sigma[k]$ truncates the stack up to the k-th frame: $(\phi_1 \ldots \phi_k \ldots \phi_n, \chi)[k] \triangleq (\phi_1 \ldots \phi_k, \chi)$
- $Vs(stmt)$ returns the variables which appear in $stmt$. For example, $Vs(u := y.f) = \{u, y\}$.

3.2 $\mathscr{L}_{ul}$ Execution

$\mathscr{L}_{ul}$ execution is described by a small steps operational semantics of the shape $\overline{M}; \sigma \dashrightarrow \sigma'$ – *c.f.* Fig. 11. $\overline{M}$ stands for one or more modules, where a module, M, maps class names to class definitions.

The semantics enforces dynamically a simple form of module-wide privacy: Fields may be read or written only if the class of the object whose field is being read or written, and the class of the object which is reading or writing belong to the same module. Private methods may be called only if the class of the receiver (the object whose method is being called), and the class of the caller (the object which is calling) belong to the same module. Public methods may always be called.

The semantics is unsurprising : In σ, the top frame's continuation contains the statement to be executed next. Statements may assign to variables, allocate new objects, perform field reads and writes on objects, and call methods on those objects. 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. Methods are expected to store their return values in the variable `res`. When the continuation is empty (ϵ), the frame is popped and the value of `res` [6] from the popped frame is stored in the variable map of the top

[5]The examples in this paper are using a slightly richer syntax for greater readability.

[6]`res` is implicit like `this`

frame. Wlog, to simplify some proofs we require, as in Kotlin, that method bodies do not assign to formal parameters, , *c.f.* Def. A.6.

Fig. 2 illustrates such $\dashrightarrow$ executions, where we distinguish steps within the same call ($\rightarrow$), entering a method ($\uparrow$), returning from a method ($\downarrow$). Thus, $\sigma_8 \rightarrow \sigma_9$ indicates that $\overline{M}; \sigma_8 \dashrightarrow \sigma_9$ is a step within the same call, $\sigma_9 \uparrow \sigma_{10}$ indicates that $\overline{M}; \sigma_9 \dashrightarrow \sigma_{10}$ is a method entry, with $\sigma_{12} \downarrow \sigma_{13}$ the corresponding return. In general, $\overline{M}; \sigma \dashrightarrow^* \sigma'$ may involve any number of calls or returns: *e.g.* $\overline{M}; \sigma_8 \dashrightarrow^* \sigma_{12}$ involves one call and no return, while $\overline{M}; \sigma_{10} \dashrightarrow^* \sigma_{15}$, involves no calls and two returns.

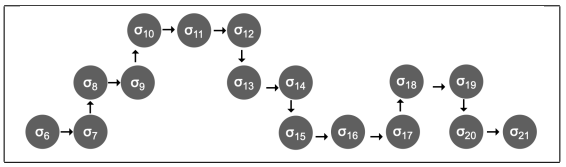


Fig. 2. $\rightarrow$: step within the same method; $\uparrow$: entering a method; $\downarrow$: returning from a method

4 FUNDAMENTAL CONCEPTS

The semantics of our assertions language is based on three fundamental concepts built on $\mathscr{L}_{ul}$: method calls and returns, scoped execution, and locally reachable objects.

4.1 Method Calls and Returns

Method calls and returns are critical for our work. They are characterized through pushing/popping frames on the stack. The operator $\sigma \triangledown \overline{\phi}$ pushes frame ϕ onto the stack of σ, while operator $\sigma\ \vartriangle$ pops a frame of σ's stack and updates the continuation and variable map.

Definition 4.1. Given a state σ, and a frame ϕ, we define

- $\sigma \triangledown \phi \triangleq (\overline{\phi} \cdot \phi, \chi)$ if $\sigma = (\overline{\phi}, \chi)$.
- $\sigma\ \vartriangle \triangleq (\overline{\phi} \cdot (\phi_n[\texttt{cont} \mapsto stmt][x \mapsto \lfloor \texttt{res} \rfloor_{\phi_n}]), \chi)$ if $\sigma = (\overline{\phi} \cdot \phi_n \cdot \phi_{n+1}, \chi)$, and $\phi_n(\texttt{cont}) \overset{\text{txt}}{=} x := y_0.m(\overline{y}); stmt$

Consider Fig. 2 again: $\sigma_8 = \sigma_7 \triangledown \phi$ for some ϕ, and σ_{15}=$\sigma_{14}\ \vartriangle$ – thus σ_8 is a called state for σ_7, and σ_{15} is the return state from σ_{14}.

4.2 Scoped Execution

Scoped invariants, *c.f.* §2.2, ensure that if a state σ satisfies A, then all future states reachable from σ—including nested method calls and returns but *stopping* before the return from the active call in σ—will also satisfy A. For example, let σ make an external call, transitioning to σ_1, execution of σ_1's continuation results in σ_2, and σ_2 returns to σ'. Suppose the module guarantees $\forall \overline{x}.\{A\}$, and $\sigma \not\models A$, but $\sigma_1 \models A$. Scoped invariants require $\sigma_2 \models A$, but allow $\sigma' \not\models A$.

History invariants [26, 61, 62], instead, allow future states to contain the return from the active call, and thus, would require that $\sigma' \not\models A$. Thus, they are, for our purposes, both *unenforceable* and overly *restrictive*. *Unenforceable*: Take $A \overset{\text{txt}}{=} \langle a.\texttt{key} \rangle$, assume in σ a path to an external object which has access to $a.\texttt{key}$, assume that path is unknown in σ_1: then, the transition from σ_1 to σ_2 cannot eliminate that path—hence, $\sigma' \not\models \langle a.\texttt{key} \rangle$. *Restrictive*: Take $A \overset{\text{txt}}{=} \langle a.\texttt{key} \rangle \wedge a.\texttt{blnce} \geq b$; then, requiring A to hold in all states from σ_1 until termination would prevent all future withdrawals from a, rendering the account useless.

Object invariants [8, 60, 74, 75, 83], on the other hand, require the invariant to hold in all (visible) states, and thus, are equally *inapplicable* for us: They would require, *e.g.*, that for all objects a, in all (visible) states, $\langle a.\texttt{key} \rangle$, and thus prevent *any* withdrawals from *any* account in *any* state.

Having established the difference between scoped, history and object invariants, only scoped have a semantics that permits any steps but stops before returning from the current active call. In order to give semantics to scoped invariants (later, in Def. 7.5), we need a new definition of execution, called *scoped execution.*

Definition 4.2 (Scoped Execution). :

- $\overline{M};\ \sigma \rightsquigarrow \sigma' \quad \triangleq \quad \overline{M}; \sigma \dashrightarrow \sigma' \ \wedge\ |\sigma| \leq |\sigma'|$
- $\overline{M};\ \sigma_1 \rightsquigarrow^* \sigma_n \quad \triangleq \quad \sigma_1 = \sigma_n \ \vee\ \exists \sigma_2, ...\sigma_{n-1}. \forall i \in [1..n)[\ \overline{M}; \sigma_i \dashrightarrow \sigma_{i+1} \ \wedge\ |\sigma_1| \leq |\sigma_{i+1}|\ ]$
- $\overline{M};\ \sigma \rightsquigarrow^*_{fin} \sigma' \quad \triangleq \quad \overline{M};\ \sigma \rightsquigarrow^* \sigma' \ \wedge\ |\sigma| = |\sigma'| \ \wedge\ \sigma'.\texttt{cont} = \epsilon$

Consider Fig. 2: Here $|\sigma_8| \leq |\sigma_9|$ and thus $\overline{M};\ \sigma_8 \rightsquigarrow \sigma_9$. Also, $\overline{M}; \sigma_{14} \dashrightarrow \sigma_{15}$ but $|\sigma_{14}| \not\leq |\sigma_{15}|$ (this step returns from the active call in σ_{14}), and hence $\overline{M};\ \sigma_{14} \not\rightsquigarrow \sigma_{15}$. Finally, even though $|\sigma_8| = |\sigma_{18}|$ and $\overline{M}; \sigma_8 \dashrightarrow^* \sigma_{18}$, we have $\overline{M};\ \sigma_8 \not\rightsquigarrow^* \sigma_{18}$: in This is so, because the execution $\overline{M}; \sigma_8 \dashrightarrow^* \sigma_{18}$ goes through the step $\overline{M}; \sigma_{14} \dashrightarrow \sigma_{15}$ 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.*,, $\overline{M};\ \sigma_9 \rightsquigarrow^* \sigma_{13}$, even though $\overline{M};\ \sigma_{12} \not\rightsquigarrow \sigma_{13}$. Nevertheless, Lemma 4.3 says, essentially, that scoped executions describe the same set of executions as those starting at an initial state[7]. For instance, revisit Fig. 2, and assume that σ_6 is an initial state. We have $\overline{M}; \sigma_{10} \dashrightarrow^* \sigma_{14}$ and $\overline{M};\ \sigma_{10} \not\rightsquigarrow^* \sigma_{14}$, but also $\overline{M};\ \sigma_6 \rightsquigarrow^* \sigma_{14}$.

Lemma 4.3. For all modules $\overline{M}$, state σ_{init}, σ, σ', where σ_{init} is initial:

- $\overline{M};\ \sigma \rightsquigarrow^* \sigma' \implies \overline{M}; \sigma \dashrightarrow^* \sigma'$
- $\overline{M}; \sigma_{init} \dashrightarrow^* \sigma' \implies \overline{M};\ \sigma_{init} \rightsquigarrow^* \sigma'$.

Lemma 4.4 says that scoped execution does not affect the contents of variables in earlier scopes. 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 4.4. For any modules $\overline{M}$, states σ, σ', variable y, and number k:

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

4.3 Locally Reachable Objects

A central concept to our work is protection, which we will define in Sect. 5.2: It requires that no external locally reachable object can have unmitigated access to that object. An object α is *locally reachable*, $\alpha \in LocRchbl(\sigma)$, if it is reachable from the top frame on the stack of σ.

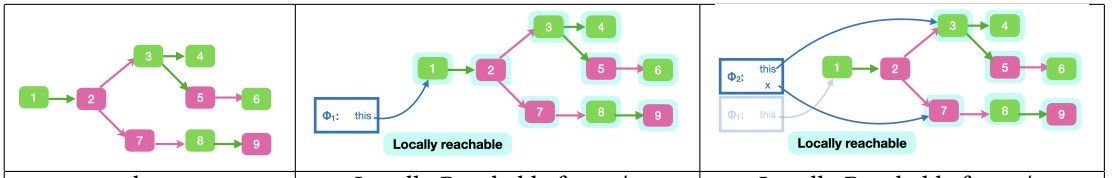
Definition 4.5. We define

- $LocRchbl(\sigma) \quad \triangleq \quad \{\ \alpha \ \mid\ \exists n \in \mathbb{N}. \exists f_1, ...f_n, x.[\ \lfloor x.f_1...f_n \rfloor_\sigma = \alpha\ ]$

We illustrate these concepts in Fig. 3: 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.

Lemma 4.6 says that (1) any object which is locally reachable right after pushing a frame was also locally reachable before pushing that frame, and (2) A pre-existing object, locally reachable after any number of scoped execution steps, was locally reachable at the first step.

[7] An *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.7)



a heap | Locally Reachable from ϕ_1 | Locally Reachable from ϕ_2

Fig. 3. A heap, two stacks, and Locally Reachable Objects. Distinction objects into green/pink explained later.

Lemma 4.6. For all modules $\overline{M}$, states σ, σ', and frame ϕ:

(1) $\sigma' = \sigma \triangledown \overline{\phi} \ \wedge \ Rng(\phi) \subseteq LocRchbl(\sigma) \implies LocRchbl(\sigma') \subseteq LocRchbl(\sigma)$
(2) $\overline{M}; \sigma \rightsquigarrow^* \sigma' \implies dom(\sigma) \cap LocRchbl(\sigma') \subseteq LocRchbl(\sigma)$

Consider Fig. 2. Lemma 4.6, 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 $\overline{M}; \sigma_8 \not\rightsquigarrow^* \sigma_{17}$, 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}.

5 ASSERTIONS

Our assertions can be standard as well as *object-capability*. The standard assertions include the values of fields, implication, quantification etc, as well as ghost fields; the latter can represent user-defined predicates. The object capability assertions express restrictions of an object's eventual authority on some other object.

Definition 5.1. Expressions, e, and assertions, A, are defined as follows:

$e \quad ::= \quad \texttt{true} \mid \alpha \mid x \mid e.f \mid e.f(\overline{e})$ [8]

$A \quad ::= \quad e \mid e : C \mid \neg A \mid A \wedge A \mid \forall x : C.A \mid e : \texttt{extl} \mid \langle e \rangle \not\leftrightarrow e \mid \langle e \rangle$

$Fv(A)$ returns the free variables in A; for example, $Fv(a{:}Account \wedge \forall b : int.[a.\texttt{blnce} = b]) = \{a\}$. Here f stands for a field, or a ghost field, but not a method – *i.e.* no side-effects.[9]

Definition 5.2 (Shorthands). We write $e : \texttt{intl}$ for $\neg(e : \texttt{extl})$, and $\texttt{extl}$. resp. $\texttt{intl}$ for $\texttt{this} : \texttt{extl}$ resp. $\texttt{this} : \texttt{intl}$. Forms as $A \longrightarrow A'$, $A \vee A'$, 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 5.3, 5.4, and 5.4.

M is used to look up the definitions of ghost fields, and to find class definitions to determine whether an object is external.

5.1 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. As expressions in $\mathscr{L}_{ul}$ may be recursively defined, their evaluation need not terminate. Nevertheless, the logic of A remains classical because recursion is restricted to expressions, and not generally to assertions.

[8] Addresses in assertions as *e.g.* in $\alpha.blnce > 700$, are useful when giving semantics to universal quantifiers *c.f.* Def. 5.3.(5), when the local map changes *e.g.* upon call and return, and in general, for scoped invariants, *c.f.* Def. 7.5.

[9] The syntax does not distinguish between fields and ghost fields. *E.g.*, `a.blnce` may, in some modules (*e.g.* in $\texttt{M}_{good}$), be a field lookup, while in others (e.g. when balance is defined though an entry in a lookup table) may execute a ghost function.

Definition 5.3 (Satisfaction of Assertions – first part)**.** We define satisfaction of an assertion A by a state σ with module M as:

(1) $M, \sigma \models e \triangleq M, \sigma, e \hookrightarrow \texttt{true}$
(2) $M, \sigma \models e : C \triangleq M, \sigma, e \hookrightarrow \alpha \wedge \mathit{classOf}(\alpha, \sigma) = C$
(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) $M, \sigma \models e : \texttt{extl} \triangleq \exists C.[\, M, \sigma \models e : C \wedge C \notin M \,]$

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 mto look up the definitions of ghost fields, and to determine whether objects are external.

5.2 Semantics of Assertions - second part

In the object capabilities model [76], *access* to a capability (called *permission* in [76] is a necessary precondition for producing a given effect; as expressed by the principle that "authority (to cause an effect) implies eventual permission" [38]. As in §2, and also [67], if no external object has eventual access for a given capability, then the corresponding effect cannot occur. Specifically, we say that o *has eventual access to* o', to mean that o either currently has or will acquire direct access to o' in the future [38].

Given this, it becomes essential to devise methods to determine whether eventual access exists in a given state. Unfortunately, this determination is undecidable, as it depends not only on the current object graph but also on the program code being executed.

In this work, we over-approximate lack of eventual access through a combination of two properties: one pertaining to the state, and the other to the internal code. The state-related property is that o is *protected* if, on any path from a locally reachable object to o, the penultimate object is internal. The program-related property is that it preserves the protection of object o.

It is straightforward to see that if o is protected and the internal code preserves its protection, then no external object can gain eventual access to o. We now define "protected":

Definition 5.4 (Satisfaction of Assertions – Protection)**.** – continuing definitions in 5.3:

(1) $M, \sigma \models \langle \alpha \rangle \nleftrightarrow \alpha_o \triangleq$
  (a) $\alpha \neq \alpha_0$, and
  (b) $\forall n \in \mathbb{N}. \forall f_1, \ldots f_n . [\, \lfloor \alpha_o.f_1 \ldots f_n \rfloor_\sigma = \alpha \implies M, \sigma \models \lfloor \alpha_o.f_1 \ldots f_{n-1} \rfloor_\sigma : C \wedge C \in M \,]$
(2) $M, \sigma \models \langle e \rangle \nleftrightarrow e_o \triangleq \exists \alpha, \alpha_o.[\, M, \sigma, e \hookrightarrow \alpha \wedge M, \sigma, e_0 \hookrightarrow \alpha_0 \wedge M, \sigma \models \langle \alpha \rangle \nleftrightarrow \alpha_o \,]$
(3) $M, \sigma \models \langle e \rangle \triangleq$
  (a) $\forall \alpha.[\, \alpha \in \mathit{LocRchbl}(\sigma) \wedge M, \sigma \models \alpha : \texttt{extl} \implies M, \sigma \models \langle e \rangle \nleftrightarrow \alpha \,]$, and
  (b) $M, \sigma \models \texttt{extl} \implies \forall x \in \sigma.\, M, \sigma \models x \neq e$

Figure 4 illustrates "protected from" and "protected". Pink and green indicate external and internal objects respectively. 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 expect 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 o_6 is not protected from o_2. Even through o_3 is internal, and is on the path from o_2 to o_6, it is not the penultimate object on that path. Therefore, o_2 can make a call to o_3, and then this call can return o_5. Once o_2 has access to o_5, it can also get access to o_6. The example justifies why we require that the *penultimate* object is internal.

In the third row of Figure 4 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

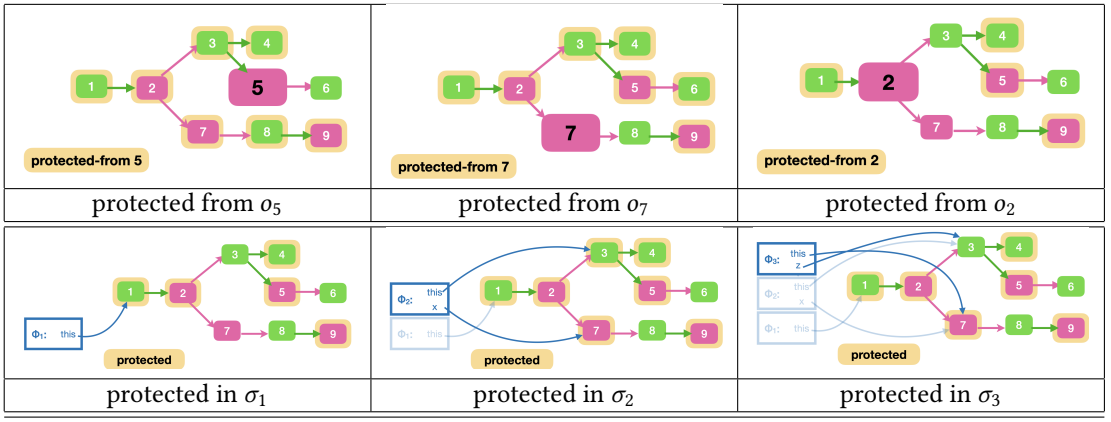


Fig. 4. Protection. Pink objects are external, and green objects are internal.

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 $\lfloor \texttt{this} \rfloor_{\sigma_3}$ is external, and o_3 is an argument to the call. As a result, during the call, o_7 may obtain unmitigated access to o_3.

Discussion. Lack of eventual 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 [88, 108] or borrowed ("2nd-hand") references [14, 22], textual modules [67], information flow [102], runtime checks [4], abstract data type exports [63], separation-based invariants Iris [41, 93], – more in § 11. 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".

One can see that protection together with protection preservation are sufficient for lack of eventual access. On the other hand, protection without protection preservation is not sufficient.

6 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.

We start with Lemma 6.1 which says that satisfaction of an assertion is not affected by replacing a variable by its value, nor by changing the continuation in a state.

Lemma 6.1. For all M, σ, α, x, e, $stmt$, and A:

(1) $M, \sigma \models A \iff M, \sigma \models A[\lfloor x \rfloor_\sigma / x]$
(2) $M, \sigma \models A \iff M, \sigma[\texttt{cont} \mapsto stmt] \models A$

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

6.1 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. This, however, is not the case for our assertions, where whether α is protected, *i.e.* whether $\langle\!\langle \alpha \rangle\!\rangle$ holds, depends on the heap as well as the set of objects reachable from the top frame; the latter changes when the frame changes. This is shown, *e.g.* in Fig. 4 where $\sigma_1, \chi \models \langle\!\langle o_3 \rangle\!\rangle$, then $.\sigma_2, \chi \not\models \langle\!\langle o_3 \rangle\!\rangle$, and then $\sigma_3, \chi \models \langle\!\langle o_3 \rangle\!\rangle$

Assertions which do not contain $\langle\_\rangle$, called $Stbl(\_)$, are preserved when pushing/popping frames. Less strictly, assertions which do not contain $\langle\_\rangle$ in *negative* positions, called $Stb^+(\_)$, are preserved when pushing internal frames provided that the range of the new frame contains locally reachable addresses – *c.f.* Defs in Appendix C and Lemma 6.2.

Lemma 6.2. For all states σ, frames ϕ, all assertions A with $Fv(A) = \emptyset$

- $Stbl(A) \implies [\ M,\sigma \models A \iff M,\sigma \triangledown \phi \models A\ ]$
- $Stb^+(A) \ \wedge\ Rng(\phi) \subseteq LocRchbl(\sigma) \ \wedge\ M,\sigma \triangledown \phi \models \texttt{intl} \ \wedge\ M,\sigma \models A \implies M,\sigma \triangledown \phi \models A$

While Stb^+ assertions *are* preserved when pushing internal frames, they are *not* necessarily preserved when pushing external frames (*c.f.* Ex. 6.3), *nor* when popping frames (*c.f.* Ex. 6.4).

Example 6.3 (Stb^+ not always preserved by External Push)**.** In Fig. 4, where σ_2 by pushing external frame onto σ_1, and $\sigma_1 \models \langle o_3\rangle$ but $\sigma_2 \not\models \langle o_3\rangle$.

Example 6.4 (Stb^+ not always preserved by Method Return)**.** Assume state σ_a, such that $\lfloor \texttt{this} \rfloor_{\sigma_a} = o_1$, $\lfloor \texttt{this}.f \rfloor_\sigma = o_2$, $\lfloor x \rfloor_\sigma = o_3$, $\lfloor x.f \rfloor_\sigma = o_2$, and $\lfloor x.g \rfloor_\sigma = o_4$, where o_2 is external and all other objects are internal. We then have $.., \sigma_a \models \langle o_4\rangle$. 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 $.., \sigma_b \models \langle o_4\rangle$. Assume the body of m is $\texttt{this}.f.m1(\texttt{this}.g); \texttt{this}.f := \texttt{this}; \texttt{this}.g := \texttt{this}$, 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 $.., \sigma_c \models \langle o_4\rangle$. However, after we pop the stack, we obtain σ_d, for which $.., \sigma_d \not\models \langle o_4\rangle$.

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* ($-\!\triangledown$ in Sect 6.2), and the latter through *scoped satisfaction* (§9.1).

6.2 Adaptation

As we discussed in §2.3 it is possible for an assertion not to be satisfied at the caller but to be satisfied at the called viewpoint (the callee). We define then operator $-\!\triangledown$ which translates an assertion from the viewpoint of the callee, to that of the caller.

Definition 6.5. [The $-\!\triangledown$ operator]

$$
\begin{array}{rcl@{\qquad}rcl}
(\langle e\rangle) -\!\triangledown \overline{y} & \triangleq & \langle e\rangle \leftarrow\!\!\!\times\ \overline{y} & (A_1 \wedge A_2) -\!\triangledown \overline{y} & \triangleq & (A_1 -\!\triangledown \overline{y}) \wedge (A_2 -\!\triangledown \overline{y}) \\
(\langle e\rangle \leftarrow\!\!\!\times\ \overline{u}) -\!\triangledown \overline{y} & \triangleq & \langle e\rangle \leftarrow\!\!\!\times\ \overline{u} & (\forall x : C.A) -\!\triangledown \overline{y} & \triangleq & \forall x : C.(A -\!\triangledown \overline{y}) \\
(e : \texttt{extl}) -\!\triangledown \overline{y} & \triangleq & e : \texttt{extl} & (\neg A) -\!\triangledown \overline{y} & \triangleq & \neg(A -\!\triangledown \overline{y}) \\
e -\!\triangledown \overline{y} & \triangleq & e & (e : C) -\!\triangledown \overline{y} & \triangleq & e : C
\end{array}
$$

Only the first equation in Def. 6.5 is interesting, $(\langle e\rangle) -\!\triangledown \overline{y}$. For e to be protected at the callee, it should be protected from all the call's arguments, *i.e.* $\langle e\rangle \leftarrow\!\!\!\times\ \overline{y}$. The notation $\langle e\rangle \leftarrow\!\!\!\times\ \overline{y}$ stands for $\langle e\rangle \leftarrow\!\!\!\times\ y_0 \ \wedge\ \ldots \wedge \langle e\rangle \leftarrow\!\!\!\times\ y_n$, assuming that $\overline{y}$=$y_0, \ldots y_n$.

Lemma 6.6 states that $-\!\triangledown$ indeed adapts assertions from the callee to the caller. It is the counterpart to the states' operator $\triangledown$: A caller σ satisfies $A -\!\triangledown \overline{y}$, if and only if the callee $(\sigma -\!\triangledown \phi)$ satisfies A.

Lemma 6.6. For state σ, assertion A with $Fv(A) = \emptyset$, variables $\overline{y}$, frame ϕ with $Range(\phi) = \overline{\lfloor y \rfloor_\sigma}$:

- $M,\sigma \models A -\!\triangledown \overline{y} \qquad \iff \qquad M,\sigma \triangledown \phi \models A$

Moreover, $-\!\triangledown$ turns an assertion into a stable assertion (Lemma 6.7), and in internal states, an assertion implies its adapted version (Lemma 6.8).

Lemma 6.7. For all assertions A : $\quad Stbl(A -\!\triangledown \overline{y})$

Lemma 6.8. For a state σ, variables $\overline{y} \subseteq dom(\sigma)$: $M, \sigma \models A \wedge \texttt{intl} \implies M, \sigma \models A \text{–}\triangledown \overline{y}$.

In general, original versions of assertions do not imply adapted versions, nor vice versa:

Example 6.9 (Adapted and Original versions are incomparable). • A does not imply $A \text{–}\triangledown \overline{y}$: *E.g.,* take a σ_1 where $\lfloor \texttt{this} \rfloor_{\sigma_1} = o_1$, and o_1 is external, and there is no other object. Then, we have $\_, \sigma_1 \models \langle \texttt{this} \rangle$ and $\_, \sigma_1 \not\models \langle \texttt{this} \rangle \leftrightarrow\!\!\!\!\times\ \texttt{this}$. • Nor does $A \text{–}\triangledown \overline{y}$ imply A. *E.g.,* take a σ_2 where $\lfloor \texttt{this} \rfloor_{\sigma_2} = o_1$, $\lfloor x \rfloor_{\sigma_2} = o_2$, and $\lfloor x.f \rfloor_{\sigma_2} = o_3$, and o_2 is external, and there are no other objects or fields. Then $\_, \sigma_2 \models \langle x.f \rangle \leftrightarrow\!\!\!\!\times\ \texttt{this}$ but $\_, \sigma_2 \not\models \langle x.f \rangle$.

6.3 Encapsulation

Proofs of adherence to specifications hinge on the expectation that some, specific, assertions are always satisfied unless some internal (and thus known) computation took place. We call such assertions *encapsulated.*

The judgment $M \vdash Enc(A)$ expresses that satisfaction of A involves looking into the state of internal objects only, *c.f.*. Def C.4. On the other hand, $M \models Enc(A)$ says that assertion A is *encapsulated* by a module M, *i.e.* in all possible states execution which involves M and any set of other modules $\overline{M}$, always satisfies A unless the execution included internal execution steps.

Definition 6.10 (An assertion A is *encapsulated* by module M).

$$M \models Enc(A) \triangleq \begin{cases} \forall \overline{M}, \sigma, \sigma', \overline{\alpha}, \overline{x} \text{ with } \overline{x} = Fv(A) \\ \quad [\ M, \sigma \models (A[\overline{\alpha/x}] \wedge \texttt{extl}) \wedge M \cdot \overline{M}; \sigma \rightsquigarrow \sigma' \implies M, \sigma' \models A[\overline{\alpha/x}]\ ] \end{cases}$$

Lemma 6.11 (Encapsulation Soundness). For all modules M, and assertions A:

$$M \vdash Enc(A) \implies M \models Enc(A).$$

7 SPECIFICATIONS

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

7.1 Specifications Syntax

Our specifications language supports scoped invariants, method specifications, and conjunctions.

Definition 7.1 (Specifications).

- The syntax of specifications, S, is given below

$$\begin{array}{rcl} S & ::= & \forall \overline{x : C}.\{A\} \mid \{A\}\, p\, C :: m(\overline{y : C})\, \{A\} \parallel \{A\} \mid S \wedge S \\ p & ::= & \texttt{private} \mid \texttt{public} \end{array}$$

- *Well-formedness*, $\vdash S$, is defined by cases on S:

$\vdash \forall \overline{x : C}.\{A\} \triangleq Fv(A) \subseteq \{\overline{x}\} \wedge M \vdash Enc(\overline{x : C} \wedge A)$;

$\vdash \{A\}\, p\, C :: m(\overline{y : C})\, \{A'\} \parallel \{A''\} \triangleq \exists \overline{x}, \overline{C'}.[$
$\texttt{res} \notin \overline{x}, \overline{y} \wedge Fv(A_0) \subseteq \overline{x}, \overline{y}, \texttt{this} \wedge Fv(A') \subseteq Fv(A), \texttt{res} \wedge Fv(A'') \subseteq \overline{x}$
$\wedge\ Stb^+(A) \wedge Stb^+(A') \wedge Stb^+(A'') \wedge M \vdash Enc(\overline{x : C'} \wedge A'')\ ]$

$\vdash S \wedge S' \triangleq \vdash S \wedge \vdash S'$.

Example 7.2 (Scoped Invariants). S_5 guarantees that non-null keys do not change:

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

Example 7.3 (Method Specifications). A method specification for method `buy` appear in §2.3. S_9 guarantees that `set` preserves the protectedness of any account, as well as of any key. Appendix D contains further examples.

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

Note that in S_9 the variables 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.

7.2 Specifications Semantics

We now move to the semantics of specifications: $M \models S$ expresses that module M satisfies a specification S. For this, we first define what it means for a state σ to satisfy a triple of assertions:

Definition 7.4. For modules $\overline{M}, M$, state σ, and assertions A, A' and A'', we define:

- $\overline{M}; M \models \{A\}\,\sigma\,\{A'\} \parallel \{A''\} \triangleq \forall \overline{z}, \overline{w}, \sigma', \sigma''.[$
  $M, \sigma \models A \implies$
  $[\ \overline{M} \cdot M;\ \sigma \rightsquigarrow^*_{fin} \sigma' \implies M, \sigma' \models A'\ ] \wedge$
  $[\ \overline{M} \cdot M;\ \sigma \rightsquigarrow^* \sigma'' \implies M, \sigma'' \models (\texttt{extl} \rightarrow A''[\overline{\lfloor z \rfloor_\sigma / z}])\ ]$
  where $\overline{z} = Fv(A)$]

Definition 7.5 (Semantics of Specifications)**.** We define $M \models S$ by cases over S:

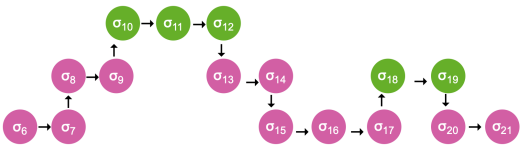
(1) $M \models \forall \overline{x : C}.\{A\} \triangleq \forall \overline{M}, \sigma.[\ \overline{M}; M \models \{\texttt{extl} \wedge \overline{x : C} \wedge A\}\,\sigma\,\{A\} \parallel \{A\}\ ]$.

(2) $M \models \{A_1\}\, p\ D :: m(\overline{y : D})\, \{A_2\} \parallel \{A_3\} \triangleq \forall \overline{M}, \sigma.[$
$\forall y_0, \overline{y}, \sigma[\ \sigma.\texttt{cont} \overset{\text{txt}}{=} u := y_0.m(y_1, ..y_n) \implies M \models \{A_1'\}\,\sigma\{A_2'\} \parallel \{A_3'\}\ ]$
where
$A_1' \triangleq y_0 : D, \overline{y : D} \wedge A[y_0/\texttt{this}],\ A_2' \triangleq A_2[u/res, y_0/\texttt{this}],\ A_3' \triangleq A_3$]

(3) $M \models S \wedge S' \triangleq M \models S \wedge M \models S'$

We demonstrate the meaning of $\forall \overline{x : C}.\{A\}$ in Fig. 5 where we refine the execution shown in Fig. 2, and take it that the pink states, *i.e.* σ_6-σ_9 and σ_{13}-σ_{17}, and σ_{20}, σ_{21} are external, and the green states, *i.e.* σ_{10}, σ_{11}, σ_{12}, σ_{18}, and σ_{19}, are internal.

Appendix D contains examples of the semantics of some of our specifications.



...; $\sigma_6 \rightsquigarrow^* \sigma_{13}$	A_0 guaranteed to be preserved from σ_6 to σ_{13}.
...; $\sigma_6 \rightsquigarrow^* \sigma_{19}$, .., $\sigma_{19} \not\models \texttt{extl}$	A_0 *not* guaranteed to be preserved from σ_6 to σ_{19}.
...; $\sigma_6 \rightsquigarrow^* \sigma_{20}$	A_0 guaranteed to be preserved from σ_6 to σ_{20}.
...; $\sigma_8 \not\rightsquigarrow^* \sigma_{20}$	A_0 *not* guaranteed to be preserved from σ_8 to σ_{20}.

Fig. 5. Illustrating the meaning on $\forall \overline{x : C}.\{A_0\}$ – refining Fig. 2.

8 HOARE LOGIC

We will now develop an inference system to prove that a module satisfies its specification. This is done in three phases.

In the first phase, we develop a logic of triples $M \vdash \{ A \}\ stmt\ \{ A' \}$, with the expected meaning, *i.e.* (*) execution of statement *stmt* in a state satisfying the *precondition* A will lead to a state satisfying the *postcondition* A'. These triples only apply to *stmt*'s that do not contain method calls (even internal) – this is so, because method calls may contain calls to external methods, and therefore can only be described through quadruples. Our triples extend an underlying Hoare logic ($M \vdash_{ul} \{A\}\ stmt\ \{A'\}$) and introduce new judgements to talk about protection.

In the second phase, we develop a logic of quadruples $M \vdash \{ A \}\ stmt\ \{ A' \} \parallel \{ A'' \}$. These promise, that (*) and in addition, that (**) any intermediate external states reachable during execution of that statement satisfy the invariant A''. We incorporate all triples from the first phase, introduce invariants, 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.

In the third phase, we prove modules' adherences to specifications. For method specifications we prove that the body maps the precondition to the postcondition and preserves the method's invariant. For module invariants we prove that they are preserved by the public methods of the module.

8.1 Preliminaries: Specification Lookup, Renamings, Underlying Hoare Logic

First some 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. 8.4, Def. F.1, and Ex. F.2. We also require an underlying Hoare logic with judgements $M \vdash_{ul} \{A\}stmt\{A'\}$, – *c.f.* Ax. F.3.

8.2 First Phase: Triples

In Fig. 6 we introduce our triples, of the form $M \vdash \{ A \}stmt\{ A' \}$. These promise, as expected, that any execution of *stmt* in a state satisfying A leads to a state satisfying A'.

$$\frac{Stbl(A)\ \ Stbl(A') \qquad M \vdash_{ul} \{ A \}\ stmt\ \{ A' \} \qquad stmt \text{ contains no method call}}{M \vdash \{ A \}\ stmt\ \{ A' \}}\ [\textsc{embed\_ul}]$$

$$\frac{}{M \vdash \{ true \}\ u = \texttt{new}\ C\ \{ \langle u \rangle \wedge \langle u \rangle \not\leftrightarrow x \}}\ [\textsc{prot-new}] \quad u \overset{\text{txt}}{\neq} x$$

$$\frac{stmt \text{ is free of method cals, or assignment to } z \qquad M \vdash \{ e = z \}\ stmt\ \{ e = z \}}{M \vdash \{ \langle e \rangle \}\ stmt\ \{ \langle e \rangle \}}\ [\textsc{prot-1}]$$

$$\frac{stmt \text{ is either } x := y \text{ or } x := y.f, \text{ and } z, z' \overset{\text{txt}}{\neq} x \qquad M \vdash \{ z = e \wedge z' = e' \}\ stmt\{ z = e \wedge z' = e' \}}{M \vdash \{ \langle e \rangle \not\leftrightarrow e' \}\ stmt\ \{ \langle e \rangle \not\leftrightarrow e' \}}\ [\textsc{prot-2}]$$

$$\frac{}{M \vdash \{ \langle y.f \rangle \not\leftrightarrow z \}\ x = y.f\ \{ \langle x \rangle \not\leftrightarrow z \}}\ [\textsc{prot-3}] \quad x \overset{\text{txt}}{\neq} z$$

$$\frac{}{M \vdash \{ \langle x \rangle \not\leftrightarrow z \wedge \langle x \rangle \not\leftrightarrow y' \}\ y.f = y'\ \{ \langle x \rangle \not\leftrightarrow z \}}\ [\textsc{prot-4}]$$

Fig. 6. Embedding the Underlying Hoare Logic, and Protection

With rule EMBED_U in Fig. 6, any assertion $M \vdash_{ul} \{A\}stmt\{A'\}$ 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 ony 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 these rules can be found in App. G.5.1. Note that "protection" of an object can decrease is if we call an external method, and pass it as argument. This will be covered by the rule in Fig. 7.

Lemma 8.1. If $M \vdash \{ A \}\ stmt\ \{ A' \}$, then $stmt$ contains no method calls.

8.3 Second Phase: Quadruples

8.3.1 Introducing invariants, 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 s is guaranteed not to contain method calls (by lemma 8.1)

$$\frac{M \vdash \{ A \}\ s\ \{ A' \}}{M \vdash \{ A \}\ s\ \{ A' \} \parallel \{ A'' \}} \text{[Mid]}$$

The remaining substructural quadruple rules appear in Fig. 14, 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. The last three rules apply to *any* statements – even those containing method calls.

8.3.2 Reasoning about calls. is described in Fig. 7. Call_Int and Call_Int_Adapt for internal methods, whether public or private; and Call_Ext_Adapt for external methods.

$$\frac{\vdash M : \{ A_1 \}\ p\ C::m(\overline{x:C})\ \{ A_2 \} \parallel \{A_3\} \qquad A_1' = A_1[y_0, \overline{y}/\texttt{this}, \overline{x}] \qquad A_2' = A_2[y_0, \overline{y}, u/\texttt{this}, \overline{x}, \texttt{res}]}{M \vdash \{\ y_0 : C, \overline{y:C} \wedge A_1'\ \}\ u := y_0.m(y_1, ..y_n)\ \{\ A_2'\ \} \parallel \{\ A_3\ \}} \text{[Call_Int]}$$

$$\frac{\vdash M : \{ A_1 \}\ p\ C::m(\overline{x:C})\ \{ A_2 \} \parallel \{A_3\} \qquad A_1' = A_1[y_0, \overline{y}/\texttt{this}, \overline{x}] \qquad A_2' = A_2[y_0, \overline{y}, u/\texttt{this}, \overline{x}, \texttt{res}]}{M \vdash \{\ y_0 : C, \overline{y:C} \wedge A_1' \multimap \triangledown(y_0, \overline{y})\ \}\ u := y_0.m(y_1, ..y_n)\ \{\ A_2' \multimap \triangledown(y_0, \overline{y})\ \} \parallel \{\ A_3\ \}} \text{[Call_Int_Adapt]}$$

$$\frac{\vdash M : \forall \overline{x:C}.\{A\}}{M \vdash \{\ y_0 : \texttt{extl} \wedge \overline{x:C} \wedge A \multimap \triangledown(y_0, \overline{y})\ \}\ u := y_0.m(y_1, ..y_n)\ \{\ A \multimap \triangledown(y_0, \overline{y})\ \} \parallel \{\ A\ \}} \text{[Call_Ext_Adapt]}$$

$$\frac{\vdash M : \forall \overline{x:C}.\{A\}}{M \vdash \{\ y_0 : \texttt{extl} \wedge \overline{x:C} \wedge A \wedge A \multimap \triangledown(y_0, \overline{y})\ \}\ u := y_0.m(y_1, ..y_n)\ \{\ A \wedge A \multimap \triangledown(y_0, \overline{y})\ \} \parallel \{\ A\ \}} \text{[Call_Ext_Adapt_Strong]}$$

Fig. 7. Hoare Quadruples for Internal and External Calls – here $\overline{y}$ stands for $y_1, ...y_n$

For the internal calls, we start as usual by looking up the method's specification, and naming the formal parameters in the method's pre- and post-condition. Call_Int is as expected: we require the precondition, and guarantee the postcondition and invariant. For Call_Int_Adapt we require the adapted pre-condition $(A_1' \multimap \triangledown(y_0, \overline{y})$ rather than $A_1')$ and also ensure the adapted post-condition $(A_2' \multimap \triangledown(y_0, \overline{y})$ rather than $A_2')$. Remember that $A_1 \multimap \triangledown(y_0, \overline{y})$ at the caller's side guarantees that A_1 holds at the start of the call (after pushing the frame with $y_0, \overline{y}$), while A_2 at the end of the call guarantees that $A_2 \multimap \triangledown(y_0, \overline{y})$ holds when returning to the caller's side (after popping the callee's frame) – cf. lemma 6.6. Call_Int and Call_Int_Adapt are applicable whether the method is public or private.

For external methods, Call_Ext_Adapt, we consider the module's invariants. If the module promises to preserve A, *i.e.* if $\vdash M : \forall \overline{x : D}.\{A\}$, and $A \text{–}\triangledown (y_0, \overline{y})$ holds before the call, then it also holds after the call. In Call_Ext_adapt, we require that the adapted version, *i.e.* that $A \text{–}\triangledown (y_0, \overline{y})$ holds before the call. Then, the adapted version also holds after the call.

Notice that for internal calls, in Call_Int we require the *un-adapted* method precondition (*i.e.* A'_1), while for external calls, both Call_Ext_Adapt and Call_Ext_Adapt_Strong, we require the *adapted* invariant (*i.e.* $A \text{–}\triangledown (y_0, \overline{y})$). This is so, because when the callee is internal, then $Stb^+(\_)$-assertions are preserved against pushing of frames – *c.f.* Lemma 6.2. On the other hand, when the callee is external, then $Stb^+(\_)$-assertions are not necessarily preserved against pushing of frames – *c.f.* Ex. 6.3 Therefore, in order to guarantee that A holds upon entry to the callee, we need to know that $A \text{–}\triangledown (y_0, \overline{y})$ held at the caller site – *c.f.* Lemma 6.6.

Remember also, that A does not imply $A \text{–}\triangledown (y_0, \overline{y})$, nor does $A \text{–}\triangledown (y_0, \overline{y})$ imply A – *c.f.* example 6.9.

Finally notice, that while $Stb^+(\_)$,-assertions are preserved against pushing of internal frames, they are not necessarily preserved against popping of such frames *c.f.* Ex. 6.4. Nevertheless, Call_Int guarantees the unadapted version, A, upon return from the method call. This is sound, because of our *scoped satisfaction* of assertions – more in Sect. 9.1.

8.4 Third phase: Proving adherence to Module Specifications

In Fig. 8 we define the judgment $\vdash M$, which says that M has been proven to be well formed.

$$\frac{\vdash \mathscr{S}pec(M) \qquad M \vdash \mathscr{S}pec(M)}{\vdash M}\ \textsc{WellFrm\_Mod} \qquad \frac{M \vdash S_1 \qquad M \vdash S_2}{M \vdash S_1 \wedge S_2}\ \textsc{Comb\_Spec}$$

$$\frac{\mathtt{mBody}(m, D, M) = p\ (\overline{y : D})\{\ stmt\ \} \qquad M \vdash \{\ \mathtt{this} : \mathtt{D}, \overline{y : D} \wedge A_1 \wedge A_1 \text{–}\triangledown (\mathtt{this}, \overline{y})\ \}\ stmt\ \{\ A_2 \wedge A_2 \text{–}\triangledown \mathtt{res}\ \} \parallel \{A_3\}}{M \vdash \{A_1\}\ p\ D :: m(\overline{y : D})\ \{A_2\} \parallel \{A_3\}}\ \textsc{method}$$

$$\frac{\forall D, m : \quad \mathtt{mBody}(m, D, M) = \mathtt{public}\ (\overline{y : D})\{\ stmt\ \} \implies M \vdash \{\ \mathtt{this} : \mathtt{D}, \overline{y : D}, \overline{x : C} \wedge A \wedge A \text{–}\triangledown (\mathtt{this}, \overline{y})\ \}\ stmt\ \{\ A \wedge A \text{–}\triangledown \mathtt{res}\ \} \parallel \{\ A\ \}}{M \vdash \forall \overline{x : C}.\{A\}}\ \textsc{invariant}$$

Fig. 8. Methods' and Modules' Adherence to Specification

Method says that a module satisfies a method specification if the body satisfies the corresponding pre-, post- and midcondition. Moreover, the precondition is strengthened by $A \text{–}\triangledown (\mathtt{this}, \overline{y})$ – this is sound because the state is internal, and by Lemma 6.8. In the postcondition we also ask that $A \text{–}\triangledown \mathtt{res}$, so that `res` does not leak any of the values that A promises will be protected. Invariant says that a module satisfies an invariant specification $\forall \overline{x : C}.\{A\}$, if the method body of each public method has A as its pre-, post- and midcondition. The pre- and post- conditions are strengthened in similar ways to Method.

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 ($Vs(stmt) \cap Fv(A_1) \subseteq \{\mathtt{this}, y_1, \ldots y_n\}$). And in invariant we require the free variables in A (which are a subset of $\overline{x}$) not to overlap with the variable in $stmt$ ($Vs(stmt) \cap \overline{x} = \emptyset$). This can easily be achieved through renamings, *c.f.* Def. F.1.

9 SOUNDNESS

In this section we demonstrate that the proof system presented in section 8 is sound. For this, we give a stronger meaning to our Hoare tuples (Sect 9.1), we require that an assertion hold from the perspective of *several frames*, rather than just the top frame.

We prove soundness of Hoare triples (Sect 9.2). We then show how execution starting from some external state can be summarised into purely external execution and terminating execution of public methods (Sect 9.3). We then use these decompositions and a well-founded ordering to prove soundness of our quadruples and of the overall system (Sect 9.4).

9.1 Scoped Satisfaction

As shown in section 6.2, an assertion which held at the end of a method execution, need not hold upon return from it – *c.f.* Ex. 6.4.

To address this problem, we introduce *scoped satisfaction*: $M, \sigma, k \models A$ says that A is satisfied in σ in all frames of σ from k onwards, *i.e.* $M, \sigma, k \models A$ iff $\sigma = ((\phi_1 \cdot ... \phi_n), \chi)$ and $k \leq n$ and $\forall j.[\ k \leq j \leq n \implies M, ((\phi_1 \cdot ... \phi_j), \chi) \models A'\ ]$ where A' is A whose free variables have been substituted according to ϕ_n – *c.f.* Def. G.5. We allso introduce "scoped" quadruples, $M \models_\circ \{\ A\ \}\ \sigma \{\ A'\ \} \ \|\ \{\ A''\ \}$, which promise that if σ satisfies A from k onwards, and executes its continuation to termination, then the final state will satisfy A' from k onwards, and also, all intermediate external states will satisfy A'' from k onwards - *c.f.* Def G.5.

Thus. continuing with example 6.4, we have that $M \models \{\ \langle o_4 \rangle\ \}\ \sigma_b \{\ \langle o_4 \rangle\ \} \ \|\ \{\ true\ \}$, but $M \not\models_\circ \{\ \langle o_4 \rangle\ \}\ \sigma_b \{\ \langle o_4 \rangle\ \} \ \|\ \{\ true\ \}$. In general, scoped satisfaction is stronger than shallow:

Lemma 9.1 (Scoped vs Shallow Satisfaction)**.** For all M, A, A', A'', σ, $stmt$, and S

- $M \models_\circ \{\ A\ \}\ \sigma \{\ A'\ \} \ \|\ \{\ A''\ \} \implies M \models \{\ A\ \}\ \sigma \{\ A'\ \} \ \|\ \{\ A''\ \}$

9.2 Soundness of the Hoare Triples Logic

We require a proof system for assertions, $M \vdash A$, and expect it to be sound. We also expect the underlying Hoare logic, $M \vdash_{ul} \{\ A\ \}\ stmt \{\ A'\ \}$, to be be sound, *c.f.* Axiom G.1. We then prove various properties about protection – *c.f.* section G.5.1 – and finally prove soundness of the inference system for triples $M \vdash \{\ A\ \}\ stmt \{\ A'\ \}$ – *c.f.* Appendix, G.5.

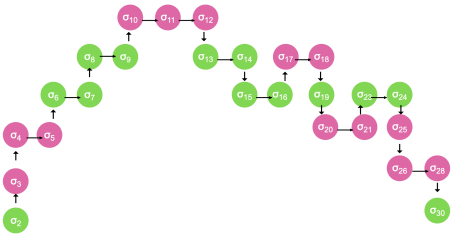
Theorem 9.2. For module M such that $\vdash M$, and for any assertions A, A', A'' and statement $stmt$:

$$M \vdash \{\ A\ \}\ stmt \{\ A'\ \} \implies M \models_\circ \{\ A\ \}\ stmt \{\ A'\ \} \ \|\ \{\ A''\ \}$$

9.3 Summarised Execution

When proving soundness of the external call rule, we are faced with the challenge that execution of an external call may consist of any number of external steps, interleaved with calls to public internal methods, which in turn may make any number of further internal call (whether public or private), and these, again may call external methods.

The diagram opposite shows such an execution: $\overline{M} \cdot M;\ \sigma_2 \rightsquigarrow^*_{fin} \sigma_{30}$ consists of **three** public internal calls, and four external calls,.The internal calls are from σ_5 to σ_6, from σ_7 to σ_8, and from σ_{21} to σ_{23}.



When proving soundness of rule CALL_EXT, we use that $M \vdash Enc(A)$ (and also scoped satisfaction) to argue that external transitions (from one external state to another external state) preserve A. For calls from external states to internal methods (as in σ_5 to σ_6), we want to apply the fact that the method body has been proven to preserve A (by METHOD). That is, for the external states we consider small steps, while for each of the public method calls we want to consider large steps – in other words, we notionally *summarize* the calls of the public methods into one large step. Moreover, for the external states we apply different arguments than for the internal states.

In terms of our example, we summarise the execution into **two** public internal calls. the "large" steps σ_6 to σ_{19} and σ_{23} to σ_{24}.

In order to express such summaries, Def. G.25 introduces *summarized* executions, whereby $(\overline{M} \cdot M,\ \sigma_{sc}); \sigma \rightsquigarrow^{*}_{e,p} \sigma'\ \mathbf{pb}\ \sigma_1 ... \sigma_n$ says that σ reaches σ' through external states, interleaved with summarised public internal method calls, starting at $\sigma_1 ... \sigma_n$, respectively. In our example, we have $(\overline{M} \cdot M,\ \sigma_2); \sigma_2 \rightsquigarrow^{*}_{e,p} \sigma_{20}\ \mathbf{pb}\ \sigma_5, \sigma_{21}$.

Lemma G.26 from the App. 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.27 says that an encapsulated assertion A is preserved by an execution starting at an external state is provided that all finalising internal executions also preserve A: that is, if A is encapsulated, and $(\overline{M} \cdot M,\ \sigma_{sc}); \sigma \rightsquigarrow^{*}_{e,p} \sigma'\ \mathbf{pb}\ \sigma_1 ... \sigma_n$, and the calls at $\sigma_1, ... \sigma_n$ preserve A, then A is preserved from σ to σ'.

9.4 Soundness of the Hoare Quadruples Logic

Another challenge when proving soundness of our quadruples, is that the proof of some cases requires induction on the execution while the proof of other cases requires induction on the quadruples proof. We address this challenge through a well-founded ordering that combines both:

In Def. G.20 we define that $(A_1, \sigma_1, A_2, A_3) \ll_{M,\overline{M}} (A_4, \sigma_2, A_5, A_6)$ if σ_1 executes to termination in fewer steps than σ_2 (considering the shortest scoped executions), or the proof of $M \vdash \{ A_1 \}\ \sigma.\texttt{cont}\{ A_2 \} \parallel \{ A_3 \}$ is shallower than that of $M \vdash \{ A_4 \}\ \sigma.\texttt{cont}\{ A_5 \} \parallel \{ A_6 \}$ (considering the shallowest proofs). The relation $\_ \ll_{M,\overline{M}} \_$ is well-founded – *c.f.* lemma G.21

THEOREM 9.3. *For module M, assertions A, A', A'', state σ, and specification S:*

$$(A):\ \vdash M \ \wedge\ M \vdash \{ A \}\ stmt\{ A' \} \parallel \{ A'' \} \implies M \models_{\circ} \{ A \}\ stmt\{ A' \} \parallel \{ A'' \}$$
$$(B):\ M \vdash S \implies M \models_{\circ} S$$

The proofs make use of summarized executions, well-founded orderings, and various assertion preservation properties. They can be found in App. G.16

10 OUR EXAMPLE PROVEN

Using our Hoare logic, we have developed a mechanised proof that, indeed, $M_{good} \vdash S_2 \wedge S_3$. In appendix H, included in the auxilliary material, we outline the main ingredients of that proof. We expand our semantics and logic to deal with scalars and conditionals, and then highlight the most interesting proof steps of that proof. The source code of the mechanised proof is included in the auxilliary material and will be submitted as an artefact.

11 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 [76, 78] 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 [84].

Our Work aims to elucidate such guarantees. We want to formalize and prove that [38]:

> *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 the concept of protection, which is 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 scoped 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 have developed a specification language for modules taming 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, 25]. These types enforce encapsulation boundaries to safeguard internal implementations, thereby ensuring representation independence and defensive consistency [6, 24, 86]. Ownership is fundamental to key systems like Rust's memory safety [53, 56], Scala's Concurrency [44, 45], Java heap analyses [48, 80, 91], and plays a critical role in program verification [13, 59] including Spec# [8, 9] and universes [31, 32, 65], Borrowable Fractional Ownership [85], and recently integrated into languages like OCAML [64, 69].

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 for capabilities accessible to some external objects while still inaccessible to others. Moreover, tightening some rules in our current Hoare logic (e.g. Def. 5.4) may lead to a native Hoare logic of ownership. Also, recent approaches like the Alias Calculus [57, 96], Reachability Types [7, 106] and Capturing Types [12, 17, 109] 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 [66] directly prohibits nominated effects, but within a closed, fully-typed world.

Temporal scope of the guarantee Starting with loop invariants[40, 49], property preservation at various granularities and duration has been widely and successfully adapted and adopted [8, 26, 36, 50, 60–62, 74, 75, 83]. 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 §4.2.

Such guarantees are maintained by the module as a whole. Drossopoulou et al. [37] proposed "holistic specifications" which take an external perspective across the interface of a module. Mackay et al. [67] 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 [37, 67] 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 [49], and have inspired a plethora of influential further developments and tools. We shall discuss a few only.

Separation logics [51, 94] reason about disjoint memory regions. Incorporating with Separation Logic's powerful framing mechanisms will pose several challenges: We have no specification 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 [28, 35] 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 [87] 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 [37, 67] 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 [46, 104] and Deny-Guarantee [34] 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 [46], rather than just at method boundaries as in visible states semantics [36, 83, 100]. In concurrent reasoning, this is because shared state may be accessed by another coöperating 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 [84] 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 that interface. Rao et al. [93]'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. [102] 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, a and prove object-capability patterns, such as sealer/unsealer, caretaker, and membrane. RustBelt [53] developed this approach to prove Rust memory safety using Iris [54], and combined with RustHorn [71] for the safe subset, produced RustHornBelt [70] that verifies both safe and unsafe

Rust programs. Similar techniques were extended to C [97]. 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. [30] 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 [63] and Cerise, which uses Iris invariants to support proofs of programs with outgoing calls and callbacks, on capability-safe CPUs [41], 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 [99] 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 [90]. VerX's specification language is based on temporal logic. It is restricted to "effectively call-back free" programs [2, 43], delaying any callbacks until the incoming call to the internal object has finished.

ConSol [107] provides a specification langauge for smart contracts, checked at runtime [39]. SCIO* [4], implemented in F*, supports both verified and unverified code. Both Consol and SCIO* are similar to gradual verification techniques [27, 110] that insert dynamic checks between verified and unverified code, and contracts for general access control [33, 55, 81].

Programming Languages incorporating object capabilities Google's Caja [79] applies (object-)capabilities [29, 76, 82], sandboxes, proxies, and wrappers to limit components' access to *ambient* authority. Sandboxing has been validated formally [68]; Many recent languages [19, 47, 95] including Newspeak [16], Dart [15], Grace [11, 52] and Wyvern [72] have adopted object capabilities. Schaefer et al. [98] has also adopted an information-flow approach to ensure confidentially 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 confidentially of both caller and callee are ensured, by assigning objects to security classes. Schaefer et al. [98] has also adopted an information-flow approach to ensure confidentially by construction.

Future work. We are interested in looking at the application of our techniques to languages that rely on lexical nesting for access control such as Javascript [77], rather than public/private annotations, languages that support ownership types such as Rust, that can be leveraged for verification [5, 58, 70], and languages from the functional tradition such as OCAML, which are gaining imperative features such as ownership and uniqueness[64, 69]. These different language paradigms may lead us to refine our ideas for eventual access, footprints and framing operators.

We expect our techniques can be incorporated into existing program verification tools [27], especially those attempting gradual verification [110], thus paving the way towards practical verification for the open world.

DATA AVAILABILITY STATEMENT

An extended version of the paper including extensive appendices of full definitions and manual proofs have been uploaded as anonymised auxilliary 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] Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, and Jérémy Thibault. 2019. Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation. arXiv:1807.04603 [cs.PL]

[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.

[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, Ştefan Ciobâcă, 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.

[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.

[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.

[8] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. 2004. Verification of object-oriented programs with invariants. *JOT* 3, 6 (2004), 27–56.

[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).

[11] Andrew Black, Kim Bruce, Michael Homer, and James Noble. 2012. Grace: the Absence of (Inessential) Difficulty. In *Onwards*.

[12] Aleksander Boruch-Gruszecki, Martin Odersky, Edward Lee, Ondrej Lhoták, and Jonathan Immanuel Brachthäusera. 2023. Capturing Types. *TOPLAS* 45, 4 (2023), 21:1–21:52.

[13] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. 2003. Ownership types for object encapsulation. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New Orleans, Louisiana, USA). ACM Press, New York, NY, USA, 213–223. https://doi.org/10.1145/604131.604156

[14] John Boyland. 2001. Alias burying: Unique variables without destructive reads. *S:P&E* 31, 6 (2001), 533–553.

[15] Gilad Bracha. 2015. *The Dart Programming Language*.

[16] Gilad Bracha. 2017. The Newspeak Language Specification Version 0.1. (Feb. 2017). `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.

[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.

[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 *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24 - 27, 2017*. 128–141. https://doi.org/10.1145/3127479.3131209

[20] Nicholas Cameron, Sophia Drossopoulou, and James Noble. 2012. Ownership Types are Existential Types. *Aliasing in Object-Oriented Programming* (2012).

[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.

[22] Edwin C. Chan, John Boyland, and William L. Scherlis. 1998. Promises: Limited Specifications for Analysis and Manipulation. In *ICSE*. 167–176.

[23] Christoph Jentsch. 2016. Decentralized Autonomous Organization to automate governance. (March 2016). https://download.slock.it/public/DAO/WhitePaper.pdf

[24] David G. Clarke, John M. Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *OOPSLA*.

[25] David G. Clarke, John M. Potter, and James Noble. 2001. Simple Ownership Types for Object Containment. In *ECOOP*.

[26] Ernie Cohen, Michal Moskal, and Wolfram Schulte andStephan Tobies. 2010. Local Verification of Global Invariants in Concurrent Programs. In *CAV*. 480–494.

[27] 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

[28] Thibault Dardinier and Peter Müller. 2024. Hyper Hoare Logic: (Dis-)Proving Program Hyperproperties. In *PLDI*, Vol. 8. https://doi.org/10.1145/3656437

[29] Jack B. Dennis and Earl C. Van Horn. 1966. Programming Semantics for Multiprogrammed Computations. *Comm. ACM* 9, 3 (1966).

[30] 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

[31] W. Dietl and P. Müller. 2005. Universes: Lightweight Ownership for JML. *JOT* 4, 8 (October 2005), 5–32.

[32] W. Dietl, Drossopoulou S., and P. Müller. 2007. Generic Universe Types. (Jan 2007). FOOL/WOOD'07.

[33] Christos Dimoulas, Scott Moore, Aslan Askarov, and Stephen Chong. 2014. Declarative Policies for Capability Control. In *Computer Security Foundations Symposium (CSF)*.

[34] Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. 2009. Deny-guarantee reasoning. In *ESOP*. Springer.

[35] Emanuele D'Osualdo, Azadeh Farzan, and Derek Dreyer. 2022. Proving hypersafety compositionally. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 289–314.

[36] S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers. 2008. A Unified Framework for Verification Techniques for Object Invariants. In *ECOOP (LNCS)*. Springer.

[37] Sophia Drossopoulou, James Noble, Julian Mackay, and Susan Eisenbach. 2020. Holistic Specifications for Robust Programs. In *FASE*. Cham, 420–440. https://doi.org/10.1007/978-3-030-45234-6_21

[38] Sophia Drossopoulou, James Noble, Mark Miller, and Toby Murray. 2016. Permission and Authority revisited – towards a formalization. In *(FTfJP)*.

[39] Robert Bruce Findler and Matthias Felleisen. 2001. Contract Soundness for object-oriented languages. In *Object-oriented programming, systems, languages, and applications (OOPSLA)* (Tampa Bay, FL, USA). ACM Press, 1–15. https://doi.org/10.1145/504282.504283

[40] Robert W. Floyd. [n. d.]. ([n. d.]).

[41] Aïna 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.

[42] A.D. Gordon and A. Jeffrey. 2001. Authenticity by typing for security protocols. In *Proceedings. 14th IEEE Computer Security Foundations Workshop, 2001.* 145–159. https://doi.org/10.1109/CSFW.2001.930143

[43] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, 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

[44] Philipp Haller. 2024. Lightweight Affine Types for Safe Concurrency in Scala (Keynote). In *Programming*.

[45] 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

[46] Ian J. Hayes and Cliff B. Jones. 2018. A Guide to Rely/Guarantee Thinking. In *SETSS 2017 (LNCS11174)*. 1–38.

[47] 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

[48] 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.

[49] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Comm. ACM* 12 (1969), 576–580.

[50] C. A. R. Hoare. 1974. Monitors: an operating system structuring concept. *Commun. ACM* 17, 10 (1974), 549–557.

[51] S. S. Ishtiaq and P. W. O'Hearn. 2001. BI as an assertion language for mutable data structures. In *POPL*. 14–26.

[52] 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

[53] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *PACMPL* 2, POPL, Article 66 (Jan. 2017), 66:1–66:34 pages.

[54] 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.

[55] John Kastner, Aaron Eline, Joseph W. Cutler, Shaobo He, Emina Torlak, Anwar Mamat, Lef Ioannidis, Darin McAdams, Matt McCutchen, Andrew Wells, Michael Hicks, Neha Rungta, Kyle Headley, Kesha Hietala, and Craig Disselkoen.

2024. Cedar: A New Language for Expressive, Fast, Safe, and Analyzable Authorization. In *oopsla*.
[56] Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language* (2nd ed.).
[57] Alexander Kogtenkov, Bertrand Meyer, and Sergey Velder. 2015. Alias calculus, change calculus and frame inference. *Sci. Comp. Prog.* 97 (2015), 163–172.
[58] 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. https://doi.org/10.1145/3586037
[59] Dirk Leinenbach and Thomas Santen. 2009. Verifying the Microsoft Hyper-V Hypervisor with VCC. In *Formal Methods*.
[60] K. Rustan M. Leino and Peter Müller. 2004. Object Invariants in Dynamic Contexts. In *ECOOP*.
[61] K. Rustan M. Leino and Wolfram Schulte. 2007. Using History Invariants to Verify Observers. In *ESOP*.
[62] B. Liskov and J. Wing. 1994. A Behavioral Notion of Subtyping. *ACM ToPLAS* 16, 6 (1994), 1811–1841.
[63] 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. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1438–1462.
[64] Anton Lorenzen, Stephen Dolan, Richard A. Eisenberg, and Sam Lindley. 2024. Oxidizing OCaml with Modal Memory Management. In *ICFP*.
[65] Y. Lu and J. Potter. 2006. Protecting Representation with Effect Encapsulation.. In *POPL*. 359–371.
[66] Matthew Lutze, Magnus Madsen, Philipp Schuster, and Jonathan Immanuel Brachthäuser. 2023. With or Without You: Programming with Effect Exclusion. *ICFP*, Article 204 (aug 2023), 28 pages. https://doi.org/10.1145/3607846
[67] Julian Mackay, Susan Eisenbach, James Noble, and Sophia Drossopoulou. 2022. Necessity Specifications are Necessary. In *OOPSLA*. ACM.
[68] S. Maffeis, J.C. Mitchell, and A. Taly. 2010. Object Capabilities and Isolation of Untrusted Web Applications. In *Proc of IEEE Security and Privacy*.
[69] Daniel Marshall and Dominic Orchard. 2024. Functional Ownership through Fractional Uniqueness. In *OOPSLA*. https://doi.org/10.1145/3649848
[70] 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.
[71] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2021. RustHorn: CHC-based Verification for Rust Programs. *TOPLAS* (2021).
[72] 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
[73] Adrian Mettler, David Wagner, and Tyler Close. 2010. Joe-E a Security-Oriented Subset of Java. In *NDSS*.
[74] Bertrand Meyer. 1992. Applying "Design by Contract". *Computer* 25, 10 (1992), 40–51.
[75] B. Meyer. 1992. *Eiffel: The Language*. Prentice Hall.
[76] Mark Samuel Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Ph. D. Dissertation. Baltimore, Maryland.
[77] Mark Samuel Miller. 2011. Secure Distributed Programming with Object-capabilities in JavaScript. (Oct. 2011). Talk at Vrije Universiteit Brussel, mobicrant-talks.eventbrite.com.
[78] Mark Samuel Miller, Tom Van Cutsem, and Bill Tulloh. 2013. Distributed Electronic Rights in JavaScript. In *ESOP*.
[79] Mark Samuel Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. 2008. Safe active content in sanitized JavaScript. code.google.com/p/google-caja/.
[80] Nick Mitchell. 2006. The Runtime Structure of Object Ownership. In *ECOOP*. 74–98.
[81] Scott Moore, Christos Dimoulas, Robert Bruce Findler, Matthew Flatt, and Stephen Chong. 2016. Extensible access control with authorization contracts. In *OOPSLA*, Eelco Visser and Yannis Smaragdakis (Eds.). 214–233.
[82] James H. Morris Jr. 1973. Protection in Programming Languages. *CACM* 16, 1 (1973).
[83] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. 2006. Modular Invariants for Layered Object Structures. *Science of Computer Programming* 62 (2006), 253–286.
[84] Toby Murray. 2010. *Analysing the Security Properties of Object-Capability Patterns*. Ph. D. Dissertation. University of Oxford.
[85] Takashi Nakayama, Yusuke Matsushita, Ken Sakayori, Ryosuke Sato, and Naoki Kobayashi. 2024. Borrowable Fractional Ownership Types for Verification. In *VMCAI*, Vol. LNCS14500. 224–246.
[86] James Noble, John Potter, and Jan Vitek. 1998. Flexible Alias Protection. In *ECOOP*.
[87] Peter W. O'Hearn. 2019. Incorrectness Logic. *Proc. ACM Program. Lang.* 4, POPL, Article 10 (Dec. 2019), 32 pages. https://doi.org/10.1145/3371078
[88] Leo Osvald, 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.

[89] Marco Patrignani and Deepak Garg. 2021. Robustly Safe Compilation, an Efficient Form of Secure Compilation. *ACM Trans. Program. Lang. Syst.* 43, 1, Article 1 (Feb. 2021), 41 pages. https://doi.org/10.1145/3436809

[90] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2020. VerX: Safety Verification of Smart Contracts. In *IEEE Symp. on Security and Privacy*.

[91] John Potter, James Noble, and David G. Clarke. 1998. The Ins and Outs of Objects. In *Australian Software Engineering Conference*. 80–89.

[92] Xiaojia Rao, Aïna Linn Georges, Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner, and Lars Birkedal. 2023. Iris-Wasm: Robust and Modular Verification of WebAssembly Programs. *Proc. ACM Program. Lang.* 7, PLDI, Article 151 (jun 2023), 25 pages. https://doi.org/10.1145/3591265

[93] Xiaojia Rao, Aïna 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.

[94] J. C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. IEEE Computer Society, 55–74.

[95] 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

[96] 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.

[97] 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*. ACM, 158–174.

[98] Ina Schaefer, Tobias Runge, Alexander Knüppel, Loek Cleophas, Derrick G. Kourie, and Bruce W. Watson. 2018. Towards Confidentiality-by-Construction. In *Leveraging Applications of Formal Methods, Verification and Validation. Modeling - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part I*. 502–515. https://doi.org/10.1007/978-3-030-03418-4_30

[99] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan. 2019. Safer Smart Contract Programming with Scilla. In *OOPSLA*.

[100] Alexander J. Summers and Sophia Drossopoulou. 2010. Considerate Reasoning and the Composite Pattern. In *VMCAI*.

[101] A. J. Summers, S. Drossopoulou, and P. Müller. 2009. A Universe-Type-Based Verification Technique for Mutable Static Fields and Methods. *JOT* (2009).

[102] David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and Compositional Verification of Object Capability Patterns. In *OOPSLA*.

[103] The Ethereum Wiki. 2018. ERC20 Token Standard. (Dec. 2018). https://theethereum.wiki/w/index.php/ERC20_Token_Standard

[104] Stephan van Staden. 2015. On Rely-Guarantee Reasoning. In *MPC*, Vol. LNCS9129. 30–49.

[105] Thomas Van Strydonck, Aina Linn Georges, Armaël Guéneau, Alix Trieu, Amin Timany, Frank Piessens, Lars Birkedal, and Dominique Devriese. 2022. Proving full-system security properties under multiple attacker models on capability machines. In *CSF*.

[106] Guannan Wei, Oliver Bracevac, Songlin Jia, Yuyan Bao, and Tiark Rompf. 2024. Polymorphic Reachability Types: Tracking Freshness, Aliasing, and Separation in Higher-Order Generic Programs. *POPL* 8 (2024), 393–424.

[107] Guannan Wei, Danning Xie, Wuqi Zhang, Yongwei Yuan, and Zhuo Zhang. 2024. Consolidating Smart Contracts with Behavioral Contracts. In *PLDI*. https://doi.org/10.1145/3656416

[108] 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.

[109] Yichen Xu and Martin Odersky. 2024. A Formal Foundation of Reach Capabilities. In *Programming*.

[110] Conrad Zimmerman, Jenna DiVincenzo, and Jonathan Aldrich. 2024. Sound Gradual Verification with Symbolic Execution. *Proc. ACM Program. Lang.* 8, POPL, Article 85 (jan 2024), 30 pages. https://doi.org/10.1145/3632927

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

We introduce $\mathscr{L}_{ul}$, a simple, typed, class-based, object-oriented language. To reduce the complexity of our formal models, $\mathscr{L}_{ul}$ lacks many common languages features, omitting static fields and methods, interfaces, inheritance, subsumption, exceptions, and control flow. $\mathscr{L}_{ul}$ includes ghost fields, that may only be used in the specification language. and which may be defined recursively.

A.1 Syntax

The syntax of $\mathscr{L}_{ul}$ is given in Fig. 9[10]. $\mathscr{L}_{ul}$ 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 $\mathscr{L}_{ul}$. A program state (σ) is a pair of of a stack and a heap. The stack is a a stack is a non-empty list of frames (ϕ), and the heal (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 ($u := y_0.m(\overline{y})$), a constructor call (`new` C), a sequence of statements ($s;\ s$), or empty (ϵ).

$\mathscr{L}_{ul}$ also includes syntax for ghost terms gt that may be used in writing specifications or the definition of ghost fields.

A.2 Semantics

$\mathscr{L}_{ul}$ is a simple object oriented language, and the operational semantics (given in Fig. 10 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 $\phi = (\overline{x \mapsto v}, s)$, and a program state $\sigma = (\overline{\phi} \cdot \phi, , \chi)$, we define:

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

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.5 defines the class lookup function an object identified by variable x.

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

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

[10]Our motivating example is provided in a slightly richer syntax for greater readability.

x, y, z			**Variable**
C, D			**Class Id.**
f			**Field Id.**
g			**Ghost Field Id.**
m			**Method Id.**
α			**Address Id.**
i	$\in$	$\mathbb{Z}$	**Integer**
v	::=	$\alpha \mid i \mid \texttt{true} \mid \texttt{false} \mid \texttt{null}$	**Value**
Mdl	::=	$\overline{C \mapsto CDef}$	**Module Def.**
$CDef$	::=	$\texttt{class}\ C\ \{\ \overline{fld}; \overline{mth}; \overline{gfld};\ \}$	**Class Def.**
T	::=	C	**Type**
s	::=	$x := y \mid x := v \mid x := y.f \mid x.f := y \mid x := y_0.m(\overline{y}) \mid \texttt{new}\ C \mid s; s \mid \epsilon$	**Statement**
mth	::=	$pr\ \texttt{method}\ m\ (\overline{x : T})\texttt{:}\ T\{\ s\ \}$	**Method Def.**
pr	::=	$\texttt{private} \mid \texttt{protected}$	**Privacy**
fld	::=	$\texttt{field}\ f\ \texttt{:}\ T$	**Field Def.**
$gfld$	::=	$\texttt{ghost}\ g\ (\overline{x : T})\{\ gt\ \} : T \mid \texttt{ghost intrnl}\ g\ (\overline{x : T})\{\ gt\ \} : T$	**Ghost Field Def.**
gt	::=	$x \mid v \mid gt{+}gt \mid gt{=}gt \mid gt{<}gt \mid \texttt{if}\ gt\ \texttt{then}\ gt\ \texttt{else}\ gt \mid gt.f \mid gt.g(\overline{gt})$	**Ghost Term**
σ	::=	$(\overline{\phi}, \chi)$	**Program State**
ϕ	::=	$(\ \overline{x \mapsto v};\ s\ )$	**Frame**
χ	::=	$(\ \overline{\alpha \mapsto o}\ )$	**Heap**
o	::=	$(\ C;\ \overline{f \mapsto v}\ )$	**Object**

Fig. 9. $\mathscr{L}_{ul}$ Syntax

Module linking is defined for modules with disjoint definitions:

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

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 $\overline{M}$, class C, and method name m, method lookup is defined as

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

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

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

Definition A.5 (Same Module)**.** For program state σ, modules $\overline{M}$, and variables x and y, we defone

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

Finally, we define well-formed states, $\models \sigma$, which guarantee that for all frames on the stack of σ, any the actual parameter of any caller frame have the same values as the formal parameters in the callee frame

$$\frac{\phi_n(\texttt{cont}) \overset{\text{txt}}{=} u := y_0.m(\overline{y}); s \qquad Meth(\overline{M}, classOf((\phi, \chi), y), m) = p\ C :: m(\overline{x : T}) \{ s' \} \\ p = \texttt{public} \vee SameModule(\texttt{this}, y_0, \sigma_1, \overline{M}) \qquad \phi'_{n+1} = (\ \texttt{this} \mapsto \lfloor y_0 \rfloor_\phi, \overline{x \mapsto \lfloor y \rfloor_\phi};\ s'\ )}{\overline{M}, (\overline{\phi} \cdot \phi_n, \chi) \dashrightarrow (\overline{\phi} \cdot \phi_n \cdot \phi_{n+1}, \chi)} \quad \text{(Call)}$$

$$\frac{\phi_{n+1}(\texttt{cont}) \overset{\text{txt}}{=} \epsilon \qquad \phi_n(\texttt{cont}) \overset{\text{txt}}{=} x := y_0.m(\overline{y}); s \qquad \phi'_n = \phi[x \mapsto \lfloor \texttt{res} \rfloor_{\phi_{n+1}}][\texttt{cont} \mapsto s] \qquad \models (\overline{\phi} \cdot \phi'_n, \chi)}{\overline{M}, (\overline{\phi} \cdot \phi_n \cdot \phi_{n+1}, \chi) \dashrightarrow (\overline{\phi} \cdot \phi'_n, \chi)} \quad \text{(Return)}$$

$$\frac{\sigma_1(\texttt{cont}) \overset{\text{txt}}{=} x := y.f; s \qquad SameModule(\texttt{this}, y, \sigma_1, \overline{M}) \qquad \sigma_2 = \sigma_1[x \mapsto \lfloor y.f \rfloor_{\sigma_1}\}][\texttt{cont} \mapsto s] \qquad \models \sigma_2}{\overline{M}, \sigma_1 \dashrightarrow \sigma_2} \quad \text{(Read)}$$

$$\frac{\sigma_1(\texttt{cont}) \overset{\text{txt}}{=} x.f := y; s \qquad SameModule(\texttt{this}, x, \sigma_1, \overline{M}) \qquad \sigma_2 = \sigma[\lfloor x \rfloor_{\sigma_1}.f \mapsto \lfloor y \rfloor_{\sigma_1}][\texttt{cont} \mapsto s]}{\overline{M}, \sigma_1 \dashrightarrow \sigma_2} \quad \text{(Write)}$$

$$\frac{\sigma_1(\texttt{cont}) \overset{\text{txt}}{=} x := \texttt{new}\ C; s \qquad fields(C) = \overline{f} \\ \overline{v} \text{ initial values for } \overline{f} \qquad \alpha \text{ fresh in } \sigma_1 \qquad \sigma_2 = \sigma_1[x \mapsto \alpha][\alpha \mapsto (\ C; \overline{f \mapsto v}\ )][\texttt{cont} \mapsto s] \qquad \models \sigma_2}{\overline{M}, \sigma_1 \dashrightarrow \sigma_2} \quad \text{(New)}$$

Fig. 10. $\mathscr{L}_{ul}$ operational Semantics

Notice that the operational semantics preserves wellformedness of the state. This is expressed through an explicit condition which overwrites a variable in the top frame, but can, if course, be done more elegantly, eg by looking up the names of the formal parameters. However, since this can easily be done, and is not the main subject of our paper, we decided not to incorporate it in the operational semantics. From now on we require implicitly that $\models \sigma$.

$$M, \sigma, v \hookrightarrow v \quad \text{(E-Val)} \qquad M, \sigma, x \hookrightarrow \lfloor x \rfloor_\sigma \quad \text{(E-Var)}$$

$$\frac{M, \sigma, e_1 \hookrightarrow i_1 \qquad M, \sigma, e_2 \hookrightarrow i_2 \qquad i_1 + i_2 = i}{M, \sigma, e_1 + e_2 \hookrightarrow i} \quad \text{(E-Add)} \qquad \frac{M, \sigma, e_1 \hookrightarrow v \qquad M, \sigma, e_2 \hookrightarrow v}{M, \sigma, e_1 = e_2 \hookrightarrow \texttt{true}} \quad \text{(E-Eq}_1\text{)}$$

$$\frac{M, \sigma, e_1 \hookrightarrow v_1 \qquad M, \sigma, e_2 \hookrightarrow v_2 \qquad v_1 \neq v_2}{M, \sigma, e_1 = e_2 \hookrightarrow \texttt{false}} \quad \text{(E-Eq}_2\text{)} \qquad \frac{M, \sigma, e \hookrightarrow \texttt{true} \qquad M, \sigma, e_1 \hookrightarrow v}{M, \sigma, e \hookrightarrow v} \quad \text{(E-If}_1\text{)}$$

$$\frac{M, \sigma, e \hookrightarrow \texttt{false} \qquad M, \sigma, e_2 \hookrightarrow v}{M, \sigma, e \hookrightarrow v} \quad \text{(E-If}_2\text{)} \qquad \frac{M, \sigma, e \hookrightarrow \alpha}{M, \sigma, e.f \hookrightarrow \lfloor \alpha.f \rfloor_\sigma} \quad \text{(E-Field)}$$

$$\frac{M, \sigma, e_1 \hookrightarrow \alpha \qquad M, \sigma, e_2 \hookrightarrow v' \qquad \texttt{ghost}\ g(x : T)\{e\} : T' \in M(classOf(\sigma, \alpha))(\texttt{gflds}) \qquad M, \sigma, [v'/x]e \hookrightarrow v}{M, \sigma, e_1.g(e_2) \hookrightarrow v} \quad \text{(E-Ghost)}$$

Fig. 11. $\mathscr{L}_{ul}$ expression evaluation

Definition A.6 (Well-formed state)**.** For program state σ:

$$\begin{aligned} \models \sigma \ \triangleq\ & \forall \overline{\phi}, \overline{\phi'}, \phi_k, \phi_{k+1}, stmt \\ & [\ \sigma = ((\overline{\phi} \cdot \phi_k \cdot \phi_{k+1} \cdot \overline{\phi'}), \chi) \wedge \phi_k.\texttt{cont} = z := y_0.m(y_1, ...y_n); stmt \implies \\ & \quad \lfloor y_0 \rfloor_{\phi_k} = \lfloor \texttt{this} \rfloor_{\phi_{k+1}} \ \wedge\ \exists y'_1, ...y'_n. \forall j \in [1..n].\ [\lfloor y_j \rfloor_{\phi_k} = \lfloor y'_j \rfloor_{\phi_{k+1}}\ ] \end{aligned}$$

Fig. 10 gives the operational semantics of $\mathscr{L}_{ul}$. Program state σ_1 reduces to σ_2 in the context of modules$\overline{M}$ if $\overline{M}, \sigma_1 \dashrightarrow \sigma_2$. The semantics in Fig. 10 are unsurprising, but it is notable that reads (Read) and writes (Write) are restricted to the class that the field belongs to, and methods may only be called if public, or from same module as current receiver.

While the small-step operational semantics of $\mathscr{L}_{ul}$ is given in Fig. 10, 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.7 (Initial Program State)**.** A program state σ is said to be an initial state ($\mathit{Initial}(\sigma)$) if and only if

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

for some address α and some statement s.

We provide a semantics for expression evaluation is given in Fig. 11. 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 $\mathscr{L}_{ul}$, and thus there is no restriction on field access.

Lemmas and Proofs. We prove lemma 4.4, using the following lemma:

Lemma A.8. For any states σ, σ', modules $\overline{M}$, number k, and variable y:

(1) $|\sigma| \leq |\sigma'| \implies |\sigma| \leq |\sigma|$.
(2) $(\overline{M}, \sigma);\ \sigma_1 \rightsquigarrow \sigma_2 \implies |\sigma| \leq |\sigma_1| \ \wedge\ |\sigma| \leq |\sigma_2|$.
(3) $(\overline{M}, \sigma);\ \sigma_1 \rightsquigarrow \sigma_2 \ \wedge\ k = |\sigma| \ \wedge\ (k < |\sigma_1| \vee k < |\sigma_2|) \implies \lfloor y \rfloor_{\sigma} = \lfloor y \rfloor_{\sigma_1[k]} = \lfloor y \rfloor_{\sigma_2[k]}$.
(4) $\overline{M}; \sigma \dashrightarrow \sigma' \ \wedge\ |\sigma| = |\sigma'| \ \wedge\ y \notin \mathit{Vs}(\sigma.\texttt{cont}) \implies \lfloor y \rfloor_{\sigma} = \lfloor y \rfloor_{\sigma'}$

Proof of lemma A.8

(1) Follows from the definition of $|\_|$, and $|\_| \leq |\_|$.
(2) Follows from the definition $(\_, \_);\ \_ \rightsquigarrow \_$ and (1).
(3) From. $(\overline{M}, \sigma);\ \sigma_1 \rightsquigarrow \sigma_2 \ \wedge\ k = |\sigma| \ \wedge\ (k < |\sigma_1| \vee k < |\sigma_2|)$ we can deduce that the step from σ_1 to σ_2 is either a method call from σ, ????
(4) Follows from the operational semantics

End Proof

We now prove lemma 4.4:

Proof of lemma 4.4

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

End Proof

B APPENDIX TO SECTION 4 – FUNDAMENTAL CONCEPTS

Proof of lemma 4.3

- 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 4.4

- We unfolding the definition of $\overline{M}$; $\sigma \rightsquigarrow \sigma'$ $\overline{M}$; $\sigma \rightsquigarrow \sigma'$ and the rules of the operational semantics.
- Take $k = |\sigma|$. We unfold the definition from 4.2, and obtain that $\sigma = \sigma'$ or, $\exists \sigma_1, ...\sigma_{n1}.\forall i \in [1..n)[\ \overline{M}; \sigma_i \dashrightarrow \sigma_{i+1} \ \wedge \ |\sigma_1| \leq |\sigma_{i+1}| \ \wedge \sigma = \sigma_1 \ \wedge \ \sigma' = \sigma_n\ ]$
  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 $y \notin Vs(\sigma.\texttt{cont})$, 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 $\overline{M}$; $\sigma \rightsquigarrow^*_{fin} \sigma'$ we obtain that $|\sigma| = |\sigma'| = k$. Therefore, we have that $\lfloor y \rfloor_{\sigma} = \lfloor y \rfloor_{\sigma'}$.

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.1. $\models \sigma \ \wedge k < |\sigma| \quad \Longrightarrow \quad LocRchbl(\sigma[k+1]) \subseteq LocRchbl(\sigma[k])$

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 4.6

(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.1), 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 6 – PRESERVATION OF SATISFACTION

Proof of lemma 6.1

Take 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. 5.3, 5.4, and 5.4, and auxiliary lemma **??**.

(2) To show that $M, \sigma \models A \iff M, \sigma[\texttt{cont} \mapsto stmt] \models A$
The proof goes by induction on the structure of A, application of Defs. 5.3, 5.4, and 5.4.

End Proof

In addition to what is claimed in Lemma 6.1, it also holds that

Lemma C.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. 5.3, 5.4, and 5.4. □

C.1 Stability

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

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

$Stbl(\langle e \rangle) \triangleq false \qquad Stbl(\langle e \rangle \leftarrowtail\!\!\!\!\times\, \overline{u}) = Stbl(e : \texttt{intl}) = Stbl(e) = Stbl(e : C) \triangleq true$

$Stbl(A_1 \wedge A_2) \triangleq Stbl(A_1) \wedge Stbl(A_2) \qquad Stbl(\forall x : C.A) = Stbl(\neg A) \triangleq Stbl(A)$

Definition C.3 ($Stb^+(\_)$)**.** assertions:

$Stb^+(\langle e \rangle) = Stb^+(\langle e \rangle \leftarrowtail\!\!\!\!\times\, \overline{u}) = Stb^+(e : \texttt{intl}) = Stb^+(e) = Stb^+(e : C) \triangleq true$

$Stb^+(A_1 \wedge A_2) \triangleq Stb^+(A_1) \wedge Stb^+(A_2) \quad Stb^+(\forall x : C.A) \triangleq Stb^+(A) \quad Stb^+(\neg A) \triangleq Stbl(A)$

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

Proof of lemma 6.2 Take any state σ, frame ϕ, assertion A,

- To show $Stbl(A) \wedge Fv(A) = \emptyset \implies [\ M, \sigma \models A \iff M, \sigma \triangledown \phi \models A\ ]$
  By induction on the structure of the definition of $Stbl(A)$.
- To show $Stb^+(A) \wedge Fv(A) = \emptyset \wedge Rng(\phi) \subseteq LocRchbl(\sigma)\ M, \sigma \models A \wedge M, \sigma \triangledown \phi \models \texttt{intl} \implies M, \sigma \triangledown \phi \models A\ ]$
  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 $\lfloor e \rfloor_\sigma = \lfloor e \rfloor_{\sigma \triangledown \phi}$. 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 $M, \sigma \triangledown \phi \models \langle \alpha \rangle$. Because $Rng(\phi) \subseteq LocRchbl(\sigma)$, we also obtain that $\sigma \triangledown \phi \Rightarrow\!\!\subseteq LocRchbl(\sigma)$. The rest follows by unfolding and folding Def. 5.4.

End Proof

C.2 Adaptation

Proof of lemma 6.6 The $\implies$ direction follows from Lemma G.38, part .b.. The $\impliedby$ direction follows from Lemma G.40, part .(ii).

End Proof

Proof of lemma 6.7 By induction on structure of A, and application of the definition of $\multimap\!\!\triangledown$, and the definition of $Stbl$. **End Proof**

Proof of lemma 6.8 By unfolding the definitions. It is crucial that σ is an internal state – an example why, is given in Ex 6.9 **End Proof**

C.3 Encapsulation

Proofs of adherence to $\mathscr{L}^{spec}$ 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 $M;\Gamma \vdash Enc(A);\Gamma'$ which checks that any objets read in the validation of A are internaml. We assume a judgment $M;\Gamma \vdash e : \texttt{intl}$ which says that in the context of Γ, the expression e belongs to a class from M. We also assume that the judgement $M;\Gamma \vdash e : \texttt{intl}$ can deal with ghistfields – eg through appropriate annotations of the ghost methods. Note that it is possible for $M;\Gamma \vdash Enc(e)$ to hold and $M;\Gamma \vdash e : \texttt{intl}$ not to hold.

$$\textsc{Enc\_1}\ \frac{M;\Gamma \vdash e : \texttt{intl} \quad M;\Gamma \vdash Enc(e);\Gamma}{M;\Gamma \vdash Enc(e.f);\Gamma} \qquad \textsc{Enc\_2}\ \frac{M;\Gamma \vdash Enc(e);\Gamma}{M;\Gamma \vdash Enc(\langle e \rangle);\Gamma} \qquad \textsc{Enc\_3}\ \frac{M;\Gamma \vdash Enc(e);\Gamma}{M;\Gamma \vdash Enc(e : C);\Gamma}$$

$$\textsc{Enc\_4}\ \frac{M;\Gamma, x : C \vdash Enc(A);\Gamma'}{M;\Gamma \vdash Enc(\forall x : C.A);\Gamma} \qquad \textsc{Enc\_5}\ \frac{M;\Gamma \vdash Enc(A);\Gamma' \quad Stbl(A)}{M;\Gamma \vdash Enc(\neg A);\Gamma'} \qquad \textsc{Enc\_6}\ \frac{M;\Gamma \vdash Enc(A_1);\Gamma'' \quad M;\Gamma'' \vdash Enc(A_2);\Gamma'}{M;\Gamma \vdash Enc(A_1 \wedge A_2);\Gamma'}$$

Fig. 12. The judgment $M;\Gamma \vdash Enc(A);\Gamma'$

An assertion A is encapsulated by a module M if in all possible states which arise from execution of module M with any other module $\overline{M}$, the validity of A can only be changed via computations internal to that module.

Definition C.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. 12.

More on Def. 6.10 If the definition 6.10 used the more general execution, $M{\cdot}\overline{M};\sigma \dashrightarrow \sigma'$, rather than the scoped execution, $M{\cdot}\overline{M};\ \sigma \rightsquigarrow \sigma'$, 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 $\sigma{=}(\phi_1{\cdot}\phi_2, \chi)$, where ϕ_1's receiver is 1, ϕ_2's receiver is 2, and there are no local variables. We have $...\sigma \models \texttt{extl} \wedge \langle 3.f \rangle$. We return from the most recent all, getting $...;\sigma \dashrightarrow \sigma'$ where $\sigma' = (\phi_1, \chi)$; and have $..., \sigma' \not\models \langle 3.f \rangle$.

Example C.5. For an assertion $A_{bal} \triangleq a : \texttt{Account} \wedge a.\texttt{balance} = b$, and modules $\texttt{M}_{bad}$ and $\texttt{M}_{fine}$ from § 2, we have $\texttt{M}_{bad} \models Enc(A_{bal})$, and $\texttt{M}_{bad} \models Enc(A_{bal})$.

Example C.6. Assume further modules, $\texttt{M}_{unp}$ and $\texttt{M}_{prt}$, which use ledgers mapping accounts to their balances, and export functions that update this map. In $\texttt{M}_{unp}$ the ledger is part of the internal module, while in $\texttt{M}_{prt}$ it is part of the external module. Then $\texttt{M}_{unp} \not\models Enc(A_{bal})$, and $\texttt{M}_{prt} \models Enc(A_{bal})$. Note that in both $\texttt{M}_{unp}$ and $\texttt{M}_{prt}$, the term $\texttt{a.balance}$ is a ghost field.

Note C.7. Relative protection is not encapsulated, (*e.g.* $M \not\models Enc(\langle x \rangle \leftrightarrow\!\!\!\!\!\times\ y)$), even though absolute protection is (*e.g.* $M \models Enc(\langle x \rangle)$). Encapsulation of an assertion does not imply encapsulation of its negation; for example, $M \not\models Enc(\neg\langle x \rangle)$.

Proof of lemma 6.11 By induction on the definition of the judgment $\_ \vdash Enc(\_)$, and then case analysis on program execution **End Proof**

D APPENDIX TO SECTION 7 – SPECIFICATIONS

Example D.1 (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.

$$
\begin{array}{lcl}
S_7 & \triangleq & \{\, a : \texttt{Account} \wedge a.\texttt{blnce} = b \wedge (\texttt{dst} \neq a \neq \texttt{this} \vee \texttt{key'} \neq a.\texttt{key}) \,\} \\
 & & \quad \texttt{public Account} :: \texttt{transfer}(\texttt{dst} : \texttt{Account}, \texttt{key'} : \texttt{Key}, \texttt{amt} : \texttt{int}) \\
 & & \{\, a.\texttt{blnce} = b \,\} \\
S_8 & \triangleq & \{\, \texttt{this} \neq \texttt{dst} \wedge \texttt{this.blnce} = b \wedge \texttt{dst.blnce} = b' \,\} \\
 & & \quad \texttt{public Account} :: \texttt{transfer}(\texttt{dst} : \texttt{Account}, \texttt{key'} : \texttt{Key}, \texttt{amt} : \texttt{int}) \\
 & & \{\, \texttt{this.blnce} = b - \texttt{amt} \wedge \texttt{dst.blnce} = b' + \texttt{amt} \,\} \\
S_9 & \triangleq & \{\, a : \texttt{Account} \wedge \langle a.\texttt{key} \rangle \,\} \\
 & & \quad \texttt{public Account} :: \texttt{set}(\texttt{key'} : \texttt{Key}) \\
 & & \{\, \langle a.\texttt{key} \rangle \,\}
\end{array}
$$

D.1 Examples of Semantics of our Specifications

Example D.2. We revisit the specifications given in Sect. 2.1, the three modules from Sect. 2.2, and Example D.1

$$
\begin{array}{llll}
\mathtt{M}_{good} \models S_1 & \mathtt{M}_{good} \models S_2 & \mathtt{M}_{good} \models S_3 & \mathtt{M}_{good} \models S_5 \\
\mathtt{M}_{bad} \models S_1 & \mathtt{M}_{bad} \not\models S_2 & \mathtt{M}_{bad} \not\models S_3 & \mathtt{M}_{bad} \not\models S_5 \\
\mathtt{M}_{fine} \models S_1 & \mathtt{M}_{fine} \models S_2 & \mathtt{M}_{fine} \not\models S_3 & \mathtt{M}_{fine} \not\models S_5
\end{array}
$$

Example D.3. For Example 7.3, we have $\mathtt{M}_{good} \models S_7$ and $\mathtt{M}_{bad} \models S_7$ and $\mathtt{M}_{fine} \models S_7$. Also, $\mathtt{M}_{good} \models S_8$ and $\mathtt{M}_{bad} \models S_8$ and $\mathtt{M}_{fine} \models S_8$. However, $\mathtt{M}_{good} \models S_9$, while $\mathtt{M}_{bad} \not\models S_9$.

Example D.4. For any specification $S \triangleq \{\, A \,\}\, p\ C :: m(\overline{x : C})\ \{\, A' \,\}$ 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. 7.5(3) would be trivially satisfied. Thus, $\mathtt{M}_{fine} \models S_8$.

E EXPRESSIVENESS

We argue the expressiveness of our approach by comparing with example specifications proposed in [67].

E.1 The DOM

This is the motivating example in [30], 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 `Node`s 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:

$$S_{dom\_1} \triangleq \forall nd : \texttt{DomNode}.\{\ \forall pr : \texttt{Proxy}.[\ may\_modify(pr, nd) \rightarrow \langle pr \rangle\ ]\ \}$$

$$S_{dom\_2} \triangleq \forall nd : \texttt{DomNode}, val : \texttt{PropertyValue}.\ \{\ \forall pr : \texttt{Proxy}.[\ may\_modify(pr, nd) \rightarrow \langle pr \rangle\ ] \wedge nd.property = val\ \}$$

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

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

In [67] this was specified as follows:

```
DOMSpec ≜ from nd : Node ∧ nd.property = p  to nd.property != p
          onlyIf ∃ o.[ o:extl ∧
                       ( ∃ nd':Node.[ o access nd'⟩ ]  ∨
                         ∃ pr:Proxy,k:ℕ.[o access pr⟩ ∧ 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-capabilties to the node that was modified. The assertion $\exists o.[\ \texttt{o} : \texttt{extl} \wedge \texttt{o access pr}\rangle\ ]$ is the contrapositive of our $\langle pr \rangle$, but is 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 DAO

The Decentralized Autonomous Organization (DAO) [23] 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.

$$S_{dao\_1} \triangleq \forall d : \texttt{DAO}.\{\ \forall p : \texttt{Participant}.[\ d.ether \geq d.balance(p)\ ]\ \}$$

$$S_{dao\_2} \triangleq \forall d : \texttt{DAO}.\{\ d.ether \geq \sum_{p \in d.particiants} d.balance(p)\ \}$$

The specifications above say the following:

S_{edao_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 [74]. 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. [11]

We now compare with the specification given in [67]. `DAOSpec1` in similar to S_{dao_1}: iy says that no participant's balance may ever exceed the ether remaining in DAO. It is, essentially, a one-state invariant.

```
DAOSpec1 ≜ from d : DAO ∧ 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.

[67] gives one more specification:

```
DAOSpec2 ≜ from d : DAO ∧ p : Object
           next d.balance(p) = m
           onlyIf ⟨p calls d.repay(_)⟩ ∧ m = 0 ∨ ⟨p calls d.join(m)⟩ ∨ 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.3 ERC20

The ERC20 [103] 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 : `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 $\forall \overline{x : C}.\{A\}.\{A'\}$ which expresses that any execution which satisfies A, will preserve A'.

We specify the guarantees of ERC20 as follows:

$$S_{erc\_1} \triangleq \forall e : \texttt{ERC20}, p : \texttt{Participant}.\{\ e.allowed(p, p)\ \}$$

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

[11] This should have been explained somewhere earlier.

$$S_{erc\_3} \triangleq \forall e : \texttt{ERC20}, p, p' : \texttt{Participant}. \{ \forall p'.[\, (e.allowed(p', p) \rightarrow \langle\!\langle p' \rangle\!\rangle \,] \,\}.\{ \neg(e.allowed(p'', p)) \,\}$$

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 the balance of e's account. That is, f 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 [67]: 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 ≜ from e : ERC20 ∧ e.balance(p) = m + m' ∧ m > 0
             next e.balance(p) = m'
             onlyIf ∃ p' p''.[⟨p' calls e.transfer(p, m)⟩ ∨
                    e.allowed(p, p'') ≥ m ∧ ⟨p" calls e.transferFrom(p', m)⟩]
```

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 ≜ from e : ERC20 ∧ p : Object ∧ p' : Object ∧ m : Nat
             next e.allowed(p, p') = m
             onlyIf ⟨p calls e.approve(p', m)⟩ ∨
                    (e.allowed(p, p') = m ∧
                     ¬ (⟨p' calls e.transferFrom(p, _)⟩ ∨
                            ⟨p calls e.allowed(p, _)⟩)) ∨
                    ∃ p''. [e.allowed(p, p') = m + m' ∧ ⟨p' calls e.transferFrom(p", m')⟩]
```

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

F APPENDIX TO SECTION 8 – PROVING OPEN CALLS AND ADHERENCE TO $\mathscr{L}^{spec}$ SPECIFICATIONS

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

Definition F.1 is broken down as follows: $S_1 \overset{\text{txt}}{\leq} S_2$ 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 $\forall \overline{x : C}.\{A\}$ can replace any of the variables $\overline{x}$. A safe renaming of $\{ A_1 \}\, p\ D :: m(\overline{y : D})\, \{ A_2 \} \ \|\ \{A_3\}$ can replace the formal parameters ($\overline{y}$) by actual parameters ($\overline{y'}$) but requires the actual parameters not to include `this`, or `res`, (*i.e.* `this`, `res` $\notin \overline{y'}$). – Moreover, it can replace the free variables which do not overlap with the formal parameters or the receiver ($\overline{x} = Fv(A_1) \setminus \{\overline{y}, \texttt{this}\}$).

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

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

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, `this'`:=`this`, and `res'`:=`res`, and using `this'` and `res'` in the call.

Example F.2. The specification from Example 7.3 can be renamed as

$$S_{9r} \quad \triangleq \quad \{\, a1 : \texttt{Account}, a2 : \texttt{Account} \wedge \langle a1 \rangle \wedge \langle a2.\texttt{key} \rangle \,\}$$
$$\texttt{public Account :: set(nKey : Key)}$$
$$\{\, \langle a1 \rangle \wedge \langle a2.\texttt{key} \rangle \,\}$$

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

F.2 Types

The rules in Fig. 13 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.

In TYPES-1 we restricted to statements which do not contain method calls in order to make lemma 8.1 valid.

F.3 Second Phase - more

We present the remaining rules of the second phase:

Finally, we discuss the proof

Proof of lemma 8.1 By induction on the rules in Fig. 6.

$$\textsc{types-1}\quad \frac{\textit{stmt} \text{ contains no method call} \qquad \textit{stmt} \text{ contains no assignment to } x}{M \vdash \{x : C\}\ \textit{stmt}\ \{x : C\}}$$

$$\textsc{types-2}\quad \frac{M \vdash \{A\}\ s\ \{A'\} \parallel \{A''\}}{M \vdash \{x : C \wedge A\}\ s\ \{x : C \wedge A'\} \parallel \{A''\}}$$

Fig. 13. Types

$$\frac{M \vdash \{A_1\}\ s\ \{A_2\} \parallel \{A\} \quad M \vdash \{A_3\}\ s\ \{A_4\} \parallel \{A\}}{M \vdash \{A_1 \wedge A_3\}\ s\ \{A_2 \wedge A_4\} \parallel \{A\}}\ [\textsc{combine}] \qquad \frac{M \vdash \{A_1\}\ s_1\ \{A_2\} \parallel \{A\} \quad M \vdash \{A_2\}\ s_2\ \{A_3\} \parallel \{A\}}{M \vdash \{A_1\}\ s_1; s_2\ \{A_3\} \parallel \{A\}}\ [\textsc{sequ}]$$

$$\frac{M \vdash \{A_4\}\ s\ \{A_5\} \parallel \{A_6\} \qquad M \vdash A_1 \to A_4 \qquad M \vdash A_5 \to A_2 \qquad M \vdash A_6 \to A_3}{M \vdash \{A_1\}\ s\ \{A_2\} \parallel \{A_3\}}\ [\textsc{consequ}]$$

Fig. 14. Hoare Quadruples - substructural rules

End Proof

G APPENDIX TO SECTION 9 – 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', $stmt$:

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

G.2 Scoped satisfaction of assertions

Definition G.2. For a state σ, and a number $i \in \mathbb{N}$ with $i \leq |\sigma|$, module M, and assertions A, A' we define:

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

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

Lemma G.3. For a states σ, σ', numbers $k, k' \in \mathbb{N}$, assertions A, A', frame ϕ and variables $\overline{z}$, $\overline{u}$:

(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) $M \models A \to A' \implies \forall \sigma. \forall k \leq |\sigma|.[\ M, \sigma, k \models A \implies M, \sigma, k \models A'\ ]$

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: Find a σ, a k and such that $\forall i \geq k.[M, \sigma[i] \models A[\overline{\lfloor z \rfloor_\sigma / z}]$, and $\exists j \geq k.[M, \sigma[j] \not\models A'[[\overline{\lfloor z \rfloor_\sigma / z}]$ such that $\overline{z} = Fv(A)$. Take $\sigma'' \triangleq \sigma[j]$, and then we have that $M, \sigma'' \models A[[\overline{\lfloor z \rfloor_\sigma / z}]$ and $M, \sigma'' \not\models A'[[\overline{\lfloor z \rfloor_\sigma / z}]$. This contradicts $M \models A \to A'$. Here we are also using the property that $M \models A$ and $u \notin Fv(A)$ implies $M \models A[u/z]$ – this is needed because we have free variables in A which are not free in $A[...]$

End Proof Sketch

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

Lemma G.4. For states σ, σ', frame ϕ such that $\sigma' = \sigma \triangledown \phi$, 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. □

G.3 Shallow and Scoped Semantics of Hoare tuples

Definition G.5 (Scoped Satisfaction of Quadruples by States)**.** For modules $\overline{M}$, M, state σ, and assertions A, A' and A''

- $\overline{M}; M \models_{\circ} \{\ A\ \}\ \sigma\ \{\ A'\ \} \parallel \{\ A''\ \} \quad \triangleq$
  $\forall k, \overline{z}, \sigma', \sigma''.[M, \sigma, k \models A \implies$
  $[\ M \cdot \overline{M};\ \sigma \rightsquigarrow^*_{fin} \sigma' \implies M, \sigma', k \models A'\ ] \wedge$
  $[\ M \cdot \overline{M};\ \sigma \rightsquigarrow^* \sigma'' \implies M, \sigma'', k \models (\texttt{extl} \to A''[\overline{\lfloor z \rfloor_\sigma / z}])\ ]$
  ]
  where $\overline{z} = Fv(A)$

Lemma G.6. For all M, $\overline{M}$ A, A', A'' and σ:

- $\overline{M}; M \models_\circ \{\, A \,\}\, \sigma\, \{\, A' \,\} \parallel \{\, A'' \,\} \implies \overline{M}; M \models \{\, A \,\}\, \sigma\, \{\, A' \,\} \parallel \{\, A'' \,\}$

We define the *meaning* of our Hoare triples, $\{\, A \,\}\ stmt \{\, A' \,\}$, 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, $\{\, A \,\}\ stmt \{\, A' \,\} \parallel \{\, A'' \,\}$, 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.7 (Scoped Semantics of Hoare triples)**.** For modules M, and assertions A, A' we define:

- $M \models \{\, A \,\}\ stmt\ \{\, A' \,\} \triangleq$
  $\forall \overline{M}. \forall \sigma. [\ \sigma.\mathtt{cont} \overset{\text{txt}}{=} stmt \implies \overline{M}; M \models \{\ A\ \}\, \sigma\, \{\, A' \,\} \parallel \{\, true \,\}\ ]$
- $M \models \{\, A \,\}\ stmt \{\, A' \,\} \parallel \{\, A'' \,\} \triangleq$
  $\forall \overline{M}. \forall \sigma. [\ \sigma.\mathtt{cont} \overset{\text{txt}}{=} stmt \implies \overline{M}; M \models \{\ A\ \}\, \sigma\, \{\, A' \,\} \parallel \{\, A'' \,\}\ ]$
- $M \models_\circ \{\, A \,\}\ stmt\ \{\, A' \,\} \triangleq$
  $\forall \overline{M}. \forall \sigma. [\ \sigma.\mathtt{cont} \overset{\text{txt}}{=} stmt \implies \overline{M}; M \models_\circ \{\ A\ \}\, \sigma\, \{\, A' \,\} \parallel \{\, true \,\}\ ]$
- $M \models_\circ \{\, A \,\}\ stmt \{\, A' \,\} \parallel \{\, A'' \,\} \triangleq$
  $\forall \overline{M}. \forall \sigma. [\ \sigma.\mathtt{cont} \overset{\text{txt}}{=} stmt \implies \overline{M}; M \models_\circ \{\ A\ \}\, \sigma\, \{\, A' \,\} \parallel \{\, A'' \,\}\ ]$

Lemma G.8 (Scoped vs Shallow Semantics of Quadruples)**.** For all M, A, A', and $stmt$:

- $M \models_\circ \{\, A \,\}\ stmt \{\, A' \,\} \parallel \{\, A'' \,\} \implies M \models \{\, A \,\}\ stmt \{\, A' \,\} \parallel \{\, A'' \,\}$

Proof. By unfolding and folding the definitions □

G.4 Scoped satisfaction of specifications

We now give a scoped meaning to specifications:

Definition G.9 (Scoped Semantics of Specifications)**.** We define $M \models_\circ S$ by cases:

(1) $M \models_\circ \forall \overline{x : C}.\{A\} \triangleq \forall \sigma. [\ M \models_\circ \{\, \mathtt{extl} \wedge \overline{x : C} \wedge A \,\}\, \sigma \{\, A \,\} \parallel \{\, A \,\}\ ]$

(2) $M \models_\circ \{\, A_1 \,\}\, p\ D :: m(\overline{y : D})\, \{\, A_2 \,\} \parallel \{A_3\} \triangleq$
$\forall y_0, \overline{y}, \sigma [\ \sigma\mathtt{cont} \overset{\text{txt}}{=} u := y_0.m(y_1, ..y_n) \implies M \models_\circ \{\, A'_1 \,\}\, \sigma \{\, A'_2 \,\} \parallel \{\, A'_3 \,\}\ ]$
where
$A'_1 \triangleq y_0 : D, \overline{y : D} \wedge A[y_0/\mathtt{this}],\ A'_2 \triangleq A_2[u/res, y_0/\mathtt{this}],\ A'_3 \triangleq A_3[y_0/\mathtt{this}]$

(3) $M \models_\circ S \wedge S' \triangleq M \models_\circ S \wedge M \models_\circ S'$

Lemma G.10 (Scoped vs Shallow Semantics of Quadruples)**.** For all M, S:

- $M \models_\circ S \implies M \models S$

G.5 Soundness of the Hoare Triples Logic

Auxiliary Lemma G.11. For any module M, assertions A, A' and A'', such that $Stb^+(A)$, and $Stb^+(A')$, and a statement $stmt$ which does not contain any method calls:

$$M \models_\circ \{\, A \,\}\ stmt \{\, A' \,\} \implies M \models_\circ \{\, A \,\}\ stmt \{\, A' \,\} \parallel \{\, A'' \,\}$$

Proof. □

G.5.1 *Lemmas about protection.*

Definition G.12. $LocRchbl(\sigma, k) \triangleq \{\alpha \mid \exists i. [\ k \le i \le |\sigma| \ \wedge\ \alpha \in LocRchbl(\sigma[i])\ ]$

Lemma G.13 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.13. 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 steps 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 it is a field assignment, say, $\sigma' = \sigma[\alpha_1, f \mapsto \alpha_2]$, then we have that $\alpha_1, \alpha_2\ LocRchbl(\sigma, |\sigma|)$. And therefore, by Lemma B.1, we also have that $\alpha_1, \alpha_2\ LocRchbl(\sigma, k)$ 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 _; _ $\rightsquigarrow$ _, 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.1.

□

Lemma G.14 guarantees that any change to the contents of an external object can only happen during execution of an external method.

Lemma G.14. For all σ, σ':

- $\overline{M} \cdot M;\ \sigma \rightsquigarrow \sigma' \ \wedge\ \sigma \models \alpha : \texttt{extl} \ \wedge\ \lfloor \alpha.f \rfloor_\sigma \neq \lfloor \alpha.f \rfloor_{\sigma'} \implies M, \sigma \models \texttt{extl}$

Proof. Through inspection of the operational semantics in Fig. 10, and in particular rule Write.

□

Lemma G.15 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.15. For all σ, σ', and α:

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

Proof.

- Because $\sigma.\texttt{cont}$ 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:
  (*) $\forall f. \forall i \in [k..m]. \forall \alpha_o \in LocRchbl(\sigma, i).[\ M, \sigma \models \alpha_o : \texttt{extl} \implies \lfloor \alpha_o.f \rfloor_\sigma \neq \alpha\ \wedge \alpha_o \neq \alpha\ ]$.
  (**) $\exists f. \exists j \in [k..m]. \exists \alpha_o \in LocRchbl(\sigma', j).[\ M, \sigma' \models \alpha_o : \texttt{extl} \wedge \lfloor \alpha_o.f \rfloor_{\sigma'} = \alpha\ \vee \alpha_o = \alpha\ ]$
  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.13, we obtain that $\alpha_o \in LocRchbl(\sigma, j)$. 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.14, we obtain $M, \sigma \models \texttt{extl}$. Contradiction!
- By induction on the number of steps, and using the bullet above.

□

Lemma G.16. For all σ, σ', and α:

- $M, \sigma \models \langle\alpha\rangle\!\leftrightarrow\!\!\!\!\times\ \alpha_o \ \wedge\ \sigma.\texttt{heap} = \sigma'.\texttt{heap} \implies M, \sigma' \models \langle\alpha\rangle\!\leftrightarrow\!\!\!\!\times\ \alpha_o$

Proof. By unfolding and folding the definitions. □

Lemma G.17. For all σ, and α, α_o, α_1, α_2:

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

Definition G.18.

- $M, \sigma \models e : \texttt{intl}\star \triangleq \forall \overline{f}.[\ M, \sigma \models e.\overline{f} : \texttt{intl}\ ]$

Lemma G.19. For all σ, and α_o and α:

- $M, \sigma \models \alpha_o : \texttt{intl}\star \implies M, \sigma \models \langle\alpha\rangle\!\leftrightarrow\!\!\!\!\times\ \alpha_o$

Proof Sketch Theorem 9.2 The proof goes by case analysis over the rule applied to obtain $M \vdash \{A\}\ stmt\ \{A'\}$:

extend By soundness of the underlying Hoare logic (axiom G.1), we obtain that $M \models \{\,A\,\}\ stmt\{\,A'\,\}$. By axiom F.3 we also obtain that $Stbl(A)$ and $Stbl(A')$. This, together with Lemma G.3, part 3, gives us that $M \models_{\circ} \{\,A\,\}\ stmt\{\,A'\,\}$. By the assumption of extend, $stmt$ does not contain any method call. Rest follows by lemma G.11.

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.15. The rule premise $M \vdash \{\ z = e\ \}\ stmt\{\ z = e\ \}$ allows us to consider addresses, α, rather than expressions, e.

Prot-2 by Lemma G.16. The rule premise $M \vdash \{\ z = e \wedge z = e'\ \}\ stmt\{\ z = e \wedge z = e'\ \}$ allows us to consider addresses α, α' rather than expressions e, e'.

Prot-3 also by Lemma G.16. 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.17.

types-1 Follows from type system, the assumption of types-1 and lemma G.11.

End Proof Sketch

G.6 Well-founded ordering

Definition G.20. For a module M, and modules $\overline{M}$, we define a measure, $[\![\, A, \sigma, A', A''\,]\!]_{M,\overline{M}}$, and based on it, a well founded ordering $(A_1, \sigma_1, A_2, A_3) \ll_{M,\overline{M}} (A_4, \sigma_2, A_5, A_6)$ as follows:

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

Lemma G.21. For any modules M and $\overline{M}$, the relation $\_ \ll_{M,\overline{M}} \_$ is well-founded.

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

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

Definition G.22. We use the form $M, \sigma \models \texttt{pub}$ to express that the currently executing method is public.[12] Note that `pub` is not part of the assertion language.

Auxiliary Lemma G.23 (Enclosed Terminating Executions). For modules $\overline{M}$, states σ, σ', σ_1:

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

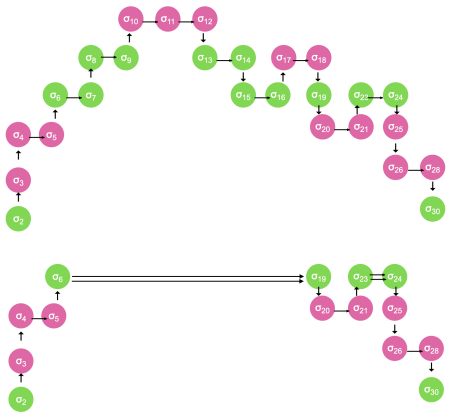
Auxiliary Lemma G.24 (Executing sequences). For modules $\overline{M}$, statements s_1, s_2, states σ, σ', σ''':

- $\sigma.\texttt{cont} = s_1; s_2 \ \wedge\ \overline{M};\ \sigma \rightsquigarrow^*_{fin} \sigma' \ \wedge\ \overline{M};\ \sigma \rightsquigarrow^* \sigma''$
  $\implies$
  $\exists \sigma''.[\ \ \overline{M};\ \sigma[\texttt{cont} \mapsto s_1] \rightsquigarrow^*_{fin} \sigma'' \ \wedge\ \overline{M};\ \sigma''[\texttt{cont} \mapsto s_2] \rightsquigarrow^*_{fin} \sigma' \wedge$
  $[\ \overline{M};\ \sigma[\texttt{cont} \mapsto s_1] \rightsquigarrow^* \sigma'' \ \vee\ \overline{M};\ \sigma''[\texttt{cont} \mapsto s_2] \rightsquigarrow^*_{fin} \sigma'''\ ] \qquad]$

G.8 Summarised Executions

We repeat the two diagrams given in §9.3.

The diagram opposite shows such an execution: $\overline{M} \cdot M;\ \sigma_2 \rightsquigarrow^*_{fin} \sigma_{30}$ 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, rom 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.25 introduces the following concepts:

- $(\overline{M} \cdot M,\ \sigma_{sc}); \sigma \rightsquigarrow^*_e \sigma'$ execution from σ to σ' scoped by σ_{sc}, involving external states only.
- $(\overline{M} \cdot M); \sigma \rightsquigarrow^*_p \sigma' \mathbf{pb}\ \sigma_1$ 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) $(\overline{M} \cdot M,\ \sigma_3); \sigma_3 \rightsquigarrow^*_e \sigma_5$ Purely external execution from σ_3 to σ_5, scoped by σ_3.

(2) $(\overline{M} \cdot M); \sigma_5 \rightsquigarrow^*_p \sigma_{20} \mathbf{pb}\ \sigma_6$. Public method call from external state σ_5 into nternal 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) $(\overline{M} \cdot M,\ \sigma_3); \sigma_{20} \rightsquigarrow^*_e \sigma_{21}$.

(4) $(\overline{M} \cdot M); \sigma_{21} \rightsquigarrow^*_p \sigma_{25} \mathbf{pb}\ \sigma_{23}$. Public method call from external state σ_{21} into internal state σ_{23}, and returning to external state σ_{25}.

(5) $(\overline{M} \cdot M,\ \sigma_3); \sigma_{25} \rightsquigarrow^*_e \sigma_{28}$. Purely external execution from σ_{25} to σ_{28}, scoped by σ_3.

Definition G.25. For any module M where M is the internal module, external modules $\overline{M}$, and states σ_{sc}, σ, σ_1, ... σ_n, and σ', we define:

[12] This 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`.

(1) $(\overline{M}\cdot M,\ \sigma_{sc});\sigma \rightsquigarrow^*_e \sigma' \triangleq \begin{cases} M,\sigma \models \texttt{extl} \wedge \\ [\ \ \sigma=\sigma' \wedge |\sigma_{sc}| \leq |\sigma| \wedge |\sigma_{sc}| \leq |\sigma''| \quad \vee \\ \ \ \exists \sigma''[\ (\overline{M}\cdot M,\sigma_{sc});\ \sigma \rightsquigarrow \sigma'' \ \wedge\ (\overline{M}\cdot M,\ \sigma_{sc});\sigma'' \rightsquigarrow^*_e \sigma'\ ]\ \ ] \end{cases}$

(2) $(\overline{M}\cdot M);\sigma \rightsquigarrow^*_p \sigma' \mathbf{pb}\ \sigma_1 \triangleq \begin{cases} M,\sigma \models \texttt{extl} \wedge \\ \exists \sigma_1'\ [\ \ (\overline{M}\cdot M,\sigma);\ \sigma \rightsquigarrow \sigma_1 \wedge M,\sigma_1 \models \texttt{pub} \wedge \\ \qquad \overline{M}\cdot M;\ \sigma_1 \rightsquigarrow^*_{fin} \sigma_1' \ \wedge\ \overline{M}\cdot M;\ \sigma_1' \rightsquigarrow \sigma'\ \ ] \end{cases}$

(3) $(\overline{M}\cdot M,\ \sigma_{sc});\sigma \rightsquigarrow^*_{e,p} \sigma' \mathbf{pb}\ \epsilon \triangleq (\overline{M}\cdot M,\ \sigma_{sc});\sigma \rightsquigarrow^*_e \sigma''$

(4) $(\overline{M}\cdot M,\ \sigma_{sc});\sigma \rightsquigarrow^*_{e,p} \sigma' \mathbf{pb}\ \sigma_1 \triangleq \exists \sigma_1', \sigma_2'.\ \begin{cases} (\overline{M}\cdot M,\ \sigma_{sc});\sigma \rightsquigarrow^*_e \sigma_1' \ \wedge\ (\overline{M}\cdot M);\sigma_1' \rightsquigarrow^*_p \sigma_2' \mathbf{pb}\ \sigma_1 \ \wedge \\ (\overline{M}\cdot M,\ \sigma_{sc});\sigma_2' \rightsquigarrow^*_e \sigma' \end{cases}$

(5) $(\overline{M}\cdot M,\ \sigma_{sc});\sigma \rightsquigarrow^*_{e,p} \sigma' \mathbf{pb}\ \sigma_1 ... \sigma_n \triangleq \exists \sigma_1'.[\ (\overline{M}\cdot M,\ \sigma_{sc});\sigma \rightsquigarrow^*_{e,p} \sigma_1' \mathbf{pb}\ \sigma_1 \wedge (\overline{M}\cdot M,\ \sigma_{sc});\sigma_1' \rightsquigarrow^*_{e,p} \sigma' \mathbf{pb}\ \sigma_2 ...$

(6) $\overline{M}\cdot M;\sigma \rightsquigarrow^*_{\epsilon,p} \sigma' \triangleq \exists n \in \mathbb{N}. \exists \sigma_1, ... \sigma_n.\ (\overline{M}\cdot M,\ \sigma);\sigma \rightsquigarrow^*_{e,p} \sigma' \mathbf{pb}\ \sigma_1 ... \sigma_n$

Note that $(\overline{M}\cdot M,\ \sigma_{sc});\sigma \rightsquigarrow^*_e \sigma'$ implies that σ is external, but does not imply that σ' is external. $(\overline{M}\cdot M,\ \sigma);\sigma \rightsquigarrow^*_e \sigma'$. On the other hand, $(\overline{M}\cdot M,\ \sigma_{sc});\sigma \rightsquigarrow^*_{e,p} \sigma' \mathbf{pb}\ \sigma_1 ... \sigma_n$ implies that σ and σ' are external, and σ_1, ... σ_1 are internal and public. Finally, note that in part (6) above it is possible that $n = 0$, and so $\overline{M}\cdot M;\sigma \rightsquigarrow^*_{\epsilon,p} \sigma'$ 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.26 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.26. [Summarised Executions] For module M, modules $\overline{M}$, and states σ, σ':

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

(1) $M\cdot\overline{M};\ \sigma \rightsquigarrow^*_{fin} \sigma' \implies \overline{M}\cdot M;\sigma \rightsquigarrow^*_{\epsilon,p} \sigma'$
(2) $M\cdot\overline{M};\ \sigma \rightsquigarrow^* \sigma' \implies$
  (a) $\overline{M}\cdot M;\sigma \rightsquigarrow^*_{\epsilon,p} \sigma' \quad \vee$
  (b) $\exists \sigma_c, \sigma_d.[\ \overline{M}\cdot M;\sigma \rightsquigarrow^*_{\epsilon,p} \sigma_c \wedge\ \overline{M}\cdot M;\ \sigma_c \rightsquigarrow \sigma_d \wedge\ M,\sigma_c \models \texttt{pub} \wedge \overline{M}\cdot M;\ \sigma_d \rightsquigarrow^* \sigma'\ ]$

Auxiliary Lemma G.27. [Preservation of Encapsulated Assertions] For any module M, modules $\overline{M}$, assertion A, and states σ_{sc}, σ, σ_1 ... σ_n, σ_a, σ_b and σ':
If

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

Then

(1) $M,\sigma \models \texttt{extl} \ \wedge\ (\overline{M}\cdot M,\sigma_{sc});\ \sigma \rightsquigarrow \sigma' \implies M,\sigma',k \models A$
(2) $(\overline{M}\cdot M,\ \sigma_{sc});\sigma \rightsquigarrow^*_e \sigma' \implies M,\sigma',k \models A$
(3) $(\overline{M}\cdot M,\ \sigma_{sc});\sigma \rightsquigarrow^*_{e,p} \sigma' \mathbf{pb}\ \sigma_1 ... \sigma_n \ \wedge$
  $\forall i \in [1..n].\forall \sigma_f.[\ M,\sigma_i,k \models A \ \wedge\ M\cdot\overline{M};\ \sigma_i \rightsquigarrow^*_{fin} \sigma_f \implies M,\sigma_f,k \models A\ ]$
  $\implies$
  $M,\sigma',k \models A$

$$\wedge$$
$$\forall i \in [1..n].M, \sigma_i, k \models A$$
$$\wedge$$
$$\forall i \in [1..n].\forall \sigma_f.[\ M \cdot \overline{M};\ \sigma_i \rightsquigarrow^*_{fin} \sigma_f \implies M, \sigma_f, k \models A\ ]$$

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 $(\overline{M} \cdot M, \sigma_{sc});\ \sigma \rightsquigarrow \sigma'$ may push or pop a frame onto σ. If it pops a frame, then $M, \sigma', k \models A$ holds by definition. If is pushes a frame, then $M, \sigma' \models A$, by lemma 6.11; this gives that $M, \sigma', k \models A$.

(2) by induction on the number of steps in $(\overline{M} \cdot M,\ \sigma_{sc}); \sigma \rightsquigarrow^*_e \sigma'$, and using (1).

(3) by induction on the number of states appearing in $\sigma_1 ... \sigma_n$, 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.28 (Sequences, Disjointness, and Disjoint Concatenation)**.** For any variables v, w, and sequences of variables $\overline{v}$, $\overline{w}$ we define:

- $v \in \overline{w} \triangleq \exists \overline{w_1}, \overline{w_1}[\ \overline{w} = \overline{w_1}, v, \overline{w_2}\ ]$
- $v \# w \triangleq \neg(v \overset{\text{txt}}{=} w)$.
- $\overline{v} \subseteq \overline{w} \triangleq \forall v.[\ v \in \overline{v} \implies v \in \overline{w}\ ]$
- $\overline{v} \# \overline{w} \triangleq \forall v \in \overline{v}. \forall w \in \overline{w}.[\ v \# w\ ]$
- $\overline{v} \cap \overline{w} \triangleq \overline{u}$, such that $\forall u.[\ u \in \overline{v} \cap \overline{w} \iff [\ u \in \overline{v} \wedge u \in \overline{w}\ ]$
- $\overline{v} \setminus \overline{w} \triangleq \overline{u}$, such that $\forall u.[\ u \in \overline{v} \setminus \overline{w} \iff [\ u \in \overline{v} \wedge u \notin \overline{w}\ ]$
- $\overline{v}; \overline{w} \triangleq \overline{v}, \overline{w}$ if $\overline{v} \# \overline{w}$ and undefined otherwise.

Lemma G.29 (Substitutions and Free Variables)**.** For any sequences of variables $\overline{x}$, $\overline{y}$, $\overline{z}$, $\overline{v}$, $\overline{w}$, a variable w, any assertion A, we have

(1) $\overline{x}[\overline{y/x}] = \overline{y}$
(2) $\overline{x} \# \overline{y} \implies \overline{y}[\overline{z/x}] = \overline{y}$
(3) $\overline{z} \subseteq \overline{y} \implies \overline{y}[\overline{z/x}] \subseteq \overline{y}$
(4) $\overline{y} \subseteq \overline{z} \implies \overline{y}[\overline{z/x}] \subseteq \overline{z}$
(5) $\overline{x} \# \overline{y} \implies \overline{z}[\overline{y/x}] \# \overline{x}$
(6) $Fv(A[\overline{y/x}]) = Fv(A)[\overline{y/x}]$
(7) $Fv(A) = \overline{x}; \overline{v},\ Fv(A[\overline{y/x}]) = \overline{y}; \overline{w} \implies \overline{v} = (\overline{y} \cap \overline{v}); \overline{w}$
(8) $\overline{v} \# \overline{x} \# \overline{y} \# \overline{u} \implies w[\overline{u/x}][\overline{v/y}] \overset{\text{txt}}{=} w[\overline{v/y}][\overline{u/x}]$
(9) $\overline{v} \# \overline{x} \# \overline{y} \# \overline{u} \implies A[\overline{u/x}][\overline{v/y}] \overset{\text{txt}}{=} A[\overline{v/y}][\overline{u/x}]$
(10) $(fv(A[\overline{y/x}]) \setminus \overline{y}) \# \overline{x}$
(11)

Proof. (1) by induction on the number of elements in $\overline{x}$
(2) by induction on the number of elements in $\overline{y}$

(3) by induction on the number of elements in $\overline{y}$
(4) by induction on the number of elements in $\overline{y}$
(5) by induction on the structure of A
(6) by induction on the structure of A
(7) Assume that
($ass1$) $Fv(A) = \overline{x}; \overline{v}$,
($ass2$) $Fv(A[\overline{y/x}]) = \overline{y}; \overline{w}$
We define:
(a) $\overline{y_0} \triangleq \overline{v} \cap \overline{y}$, $\overline{v_2} \triangleq \overline{v} \setminus \overline{y}$, $\overline{y_1} = \overline{y_0}[\overline{x/y}]$
This gives:
(b) $\overline{y_0} \# \overline{v_2}$
(c) $\overline{v} = \overline{y_0}; \overline{v_2}$
(d) $\overline{y_1} \subseteq \overline{y}$
(e) $\overline{v_2}[\overline{y/x}] = \overline{v_2}$, from assumption and (a) we have $\overline{x} \# \overline{v_2}$ and by Lemma G.29) part (2)
We now calculate

$$\begin{array}{lcll} Fv(A[\overline{y/x}]) & = & (\overline{x}; \overline{v})[\overline{y/x}] & \text{by (ass1), and Lemma G.29 part (5).} \\ & = & (\overline{x}; \overline{y_0}; \overline{v_2})[\overline{y/x}] & \text{by (c) above} \\ & = & \overline{x}[\overline{y/x}], \overline{y_0}[\overline{y/x}], \overline{v_2}[\overline{y/x}] & \text{by distributivity of [../..]} \\ & = & \overline{y}, \overline{y_1}, \overline{v_2} & \text{by Lemma G.29 part (1), (a), and (e).} \\ & = & \overline{y}; \overline{v_2} & \text{because (d), and } \overline{y} \# \overline{v_2} \\ Fv(A[\overline{y/x}]) & = & \overline{y}; \overline{w} & \text{by (ass2)} \end{array}$$

The above gives that $\overline{v_2} = \overline{w}$. This, together with (a) and (c) give that $\overline{v} = (\overline{y} \cap \overline{v}); \overline{w}$
(8) By case analysis on whether $w \in \overline{x}$ ... etc
(9) By induction on the structure of A, and the guarantee from (8).
(10) We take a variable sequence $\overline{z}$ such that
(a) $Fv(A) \subseteq \overline{x}; \overline{z}$
This gives that
(b) $\overline{x} \# \overline{z}$
Part (6) of our lemma and (a) give
(c) $Fv(A[\overline{y/x}]) \subseteq \overline{y}, \overline{z}$
Therefore
(d) $Fv(A[\overline{y/x}]) \setminus \overline{y} \subseteq \overline{z}$
The above, together with (b) conclude the proof

□

Lemma G.30 (Substitutions and Adaptations)**.** For any sequences of variables $\overline{x}$, $\overline{y}$, sequences of expressions $\overline{e}$, and any assertion A, we have

- $\overline{x} \# \overline{y} \implies (A[\overline{e/x}]) \multimap \triangledown \overline{y} \overset{\text{txt}}{=} (A \multimap \triangledown \overline{y})[\overline{e/x}]$

Proof. We first consider A to be $\langle\!\langle e \rangle\!\rangle_0$, and just take one variable. Then,
$(\langle\!\langle e_0 \rangle\!\rangle[e/x]) \multimap \triangledown \overline{y} \overset{\text{txt}}{=} \langle\!\langle e_0[e/x] \rangle\!\rangle \nleftrightarrow y$,
and
$(\langle\!\langle e_0 \rangle\!\rangle \multimap \triangledown \overline{y})[e/x] \overset{\text{txt}}{=} \langle\!\langle e_0[e/x] \rangle\!\rangle \nleftrightarrow y[e/x]$.
When $x \# y$ then the two assertions from above are textually equal. The rest follows by induction on the length of $\overline{x}$ and the structure of A. □

Lemma G.31. For assertion A, variables $\overline{x}$, $\overline{v}$, $\overline{y}$, $\overline{w}$, $\overline{v_1}$, addresses $\overline{\alpha_x}$, $\overline{\alpha_y}$, $\overline{\alpha_v}$, and $\overline{\alpha_{v_1}}$
If

a. $Fv(A) \overset{\text{txt}}{=} \overline{x};\overline{v}, \quad Fv(A[\overline{y/x}]) \overset{\text{txt}}{=} \overline{y};\overline{w},$
b. $\forall x \in \overline{x}.[\ x[\overline{y/x}][\overline{\alpha_y/y}] \ = \ x[\overline{\alpha_x/x}]\ ]$
c. $\overline{v} \overset{\text{txt}}{=} \overline{v_1};\overline{w}, \quad \overline{v_1} \overset{\text{txt}}{=} \overline{y} \cap \overline{v}, \quad \overline{\alpha_{v,1}} = \overline{v_1}[\overline{\alpha_v/v}]$

then

- $A[\overline{y/x}][\overline{\alpha_y/y}] \overset{\text{txt}}{=} A[\overline{\alpha_x/x}][\overline{\alpha_{v,1}/v_1}]$

Proof.
From Lemma G.29, part 7, we obtain $(*)\ \overline{v} = (\overline{y} \cap \overline{v});\overline{w}$
We first prove that

$$(**)\ \forall z \in Fv(A)[\ z[\overline{y/x}][\overline{\alpha_y/y}] \overset{\text{txt}}{=} z[\overline{\alpha_x/x}][\overline{\alpha_{v,1}/v_1}].$$

Take any arbitrary $z \in Fv(A)$.
Then, by assumptions a. and c., and (*) we have that either $z \in \overline{x}$, or $z \in \overline{v_1}$, or $z \in \overline{w}$.

1st Case $z \in \overline{x}$. Then, there exists some $y_z \in \overline{y}$, and some $\alpha_z \in \overline{\alpha_y}$, such that $z[\overline{y/x}] = y_z$ and $y_z[\overline{\alpha_y/y}] = \alpha_z$. On the other hand, by part b. we obtain, that $z[\overline{\alpha_x/x}] = \alpha_z$. And because $\overline{v_1}\#\overline{\alpha x}$ we also have that $\alpha_z[\overline{\alpha_{v,1}/v_1}]=\alpha_z$. This concludes the case.

2nd Case $z \in \overline{v_1}$, which means that $z \in \overline{y} \cap \overline{v}$. Then, because $\overline{x}\#\overline{v}$, we have that $z[\overline{y/x}] = z$. And because $z \in \overline{y}$, we obtain that there exists a α_z, so that $z[\overline{\alpha_y/y}] = \alpha_z$. Similarly, because $\overline{x}\#\overline{v}$, we also obtain that $z[\overline{\alpha_x/x}] = z$. And because $\overline{v_1} \subseteq \overline{y}$, we also obtain that $z[\overline{\alpha_{v,1}/v_1}]=z[\overline{\alpha_y/y}]$. This concludes the case.

3rd Case $z \in \overline{w}$. From part a. of the assumptions and from (*) we obtain $\overline{w}\#\overline{y}\#\overline{x}$, which implies that $z[\overline{y/x}][\overline{\alpha_y/y}]=z$. Moreover, (*) also gives that $\overline{w}\#\overline{v_1}$, and this gives that $z[\overline{\alpha_x/x}][\overline{\alpha_{v,1}/v_1}]=z$. This concludes the case

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

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.32. For any state σ, addresses α, α', we define

- $Reach(\alpha, \alpha')_\sigma \ \triangleq\ \exists \overline{f}.[\ \lfloor \alpha.\overline{f} \rfloor_\sigma = \alpha'\ ]$
- $\sigma \sim \sigma' \ \triangleq\ \exists \chi, \overline{\phi_1}, \overline{\phi_2}.[\ \sigma = (\overline{\phi_1}, \chi)\ \wedge\ \sigma' = (\overline{\phi_1}, \chi)\ ]$

Lemma G.33. For any module M, state σ, addresses α, α', α''

(1) $M, \sigma \models \langle\alpha\rangle \leftarrow\!\!\!\!\times\ \alpha' \ \wedge\ Reach(\alpha', \alpha'')_\sigma \implies M, \sigma \models \langle\alpha\rangle \leftarrow\!\!\!\!\times\ \alpha''$
(2) $\sigma \sim \sigma' \implies [\ Reach(\alpha, \alpha')_\sigma \iff Reach(\alpha, \alpha')_{\sigma'}\ ]$
(3) $\sigma \sim \sigma' \implies [\ M, \sigma \models \langle\alpha\rangle \leftarrow\!\!\!\!\times\ \alpha'' \iff M, \sigma' \models \langle\alpha\rangle \leftarrow\!\!\!\!\times\ \alpha''\ ]$
(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.33 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 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 ($M, \sigma' \models \texttt{intl}$), or require the adapted version of an assertion (*i.e.* $A \multimap \triangledown \overline{v}$ 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.* $Fv(A_{in}) = \overline{v_1}; \overline{v_2}$, where the value of $\overline{v_3}$ in the caller is the same as that of $\overline{v_1}$ in the called frame. This is described in lemmas G.38 - G.40.

We have four lemmas: Lemma G.38 describes preservation from a caller to an internal called, lemma G.39 describes preservation from a caller to any called, Lemma G.40 describes preservation from an internal called to a caller, and Lemma G.41 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. 7.

In the rest of this section we will first introduce some further auxiliary concepts and lemmas, and then state, discuss and prove Lemmas G.38 - G.40.

Plans for next three subsections Lemmas G.38-G.39 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.38-G.39, 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.38-G.39, 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.34 (From caller to called - protected, and variable-free)**.** For any address α, addresses $\overline{\alpha}$, states σ, σ', and frame ϕ.
If $\sigma' = \sigma \triangledown \phi$ then

a. $M, \sigma, k \models \langle\alpha\rangle \ \wedge \ M, \sigma' \models \texttt{intl} \ \wedge \ Rng(\phi) \subseteq LocRchbl(\sigma) \implies M, \sigma', k \models \langle\alpha\rangle$
b. $M, \sigma, k \models \langle\alpha\rangle \leftarrow\!\!\!\times\, \overline{\alpha} \ \wedge \ Rng(\phi) \subseteq \overline{\alpha} \implies M, \sigma' \models \langle\alpha\rangle$
c. $M, \sigma, k \models \langle\alpha\rangle \wedge \langle\alpha\rangle \leftarrow\!\!\!\times\, \overline{\alpha} \ \wedge \ Rng(\phi) \subseteq \overline{\alpha} \implies M, \sigma', k \models \langle\alpha\rangle$

Proof.

a. (1) Take any $\alpha' \in LocRchbl(\sigma')$. Then, by assumptions, we have $\alpha' \in LocRchbl(\sigma)$. This gives, again by assumptions, that $M, \sigma \models \langle\alpha\rangle \leftarrow\!\!\!\times\, \alpha'$. By the construction of σ, and lemma G.33 part 1, we obtain that (2) $M, \sigma' \models \langle\alpha\rangle \leftarrow\!\!\!\times\, \alpha'$. From (1) and (2) and because $M, \sigma' \models \texttt{intl}$ we obtain that $M, \sigma' \models \langle\alpha\rangle$. Then apply lemma G.33 part G.4, and we are done.
b. By unfolding and folding the definitions, and application of Lemma G.33 part 1.
c. By part G.33 part b. and G.4.

Notice that part G.34 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 $M, \sigma \models \langle\alpha\rangle \leftarrow\!\!\!\times\, \overline{\alpha}$ but $M, \sigma \not\models \langle\alpha\rangle$.

□

Lemma G.35 (From called to caller – protected, and variable-free)**.** For any states σ, σ', variable v, address α_v, addresses $\overline{\alpha}$, and statement $stmt$.
If $\sigma' = (\sigma \vartriangle)[v \mapsto \alpha_v][\texttt{cont} \mapsto stmt]$,
then

a. $M, \sigma, k \models \langle\alpha\rangle \ \wedge \ k < |\sigma| \ \wedge \ M, \sigma \models \langle\alpha\rangle \leftrightarrow\!\!\!\!\times \ \alpha_v \quad \Longrightarrow \quad M, \sigma', k \models \langle\alpha\rangle$.
b. $M, \sigma \models \langle\alpha\rangle \ \wedge \ \overline{\alpha} \subseteq LocRchbl(\sigma) \quad \Longrightarrow \quad M, \sigma', k \models \langle\alpha\rangle \leftrightarrow\!\!\!\!\times \ \overline{\alpha}$.

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 $\alpha' \in LocRchbl(\sigma[i])$. We obtain that $M, \sigma[i] \models \langle\alpha\rangle \leftrightarrow\!\!\!\!\times \ \alpha'$. 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 $\alpha' \in LocRchbl(\sigma')$.
Then, we have that either (A): $\alpha' \in LocRchbl(\sigma[|\sigma'|])$, or (B): $Reach(\alpha_r, \alpha')_{\sigma'}$.
In the case of (A): Because $k, |\sigma| = |\sigma'| + 1$, and because $M, \sigma, k \models \langle\alpha\rangle$ we have $M, \sigma \models \langle\alpha\rangle \leftrightarrow\!\!\!\!\times \ \alpha'$. Because $\sigma \sim \sigma'$ and Lemma G.33 part 3, we obtain (A') $M, \sigma' \models \langle\alpha\rangle \leftrightarrow\!\!\!\!\times \ \alpha'$
In the case of (B): Because $\sigma \sim \sigma'$ and lemma G.33 part 2, we obtain $Reach(\alpha_r, \alpha')_\sigma$. Then, applying Lemma G.33 part 3 and assumptions, we obtain (B') $M, \sigma' \models \langle\alpha\rangle \leftrightarrow\!\!\!\!\times \ \alpha'$.
From (3), (A), (A'), (B) and (B') we obtain: (4) $M, \sigma' \models \langle\alpha\rangle$.
With (1), (2), (4) and Lemma G.33 part 4 we are done.
b. From the definitions we obtain that $M, \sigma \models \langle\alpha\rangle \leftrightarrow\!\!\!\!\times \ \overline{\alpha}$. Because $\sigma \sim \sigma'$ and lemma G.33 part 3, we obtain $M, \sigma' \models \langle\alpha\rangle \leftrightarrow\!\!\!\!\times \ \overline{\alpha}$. And because of lemma G.3, part 3, we obtain $M, \sigma', k \models \langle\alpha\rangle \leftrightarrow\!\!\!\!\times \ \overline{\alpha}$.

□

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.34 and G.35

Lemma G.36 (From caller to called - variable-free, and $Stbl^+$)**.** For any assertion A, addresses $\overline{\alpha}$, states σ, σ', and frame ϕ.
If $\sigma' = \sigma \triangledown \phi$ and $Stb^+(A)$, and $Fv(A) = \emptyset$,
then

a. $M, \sigma, k \models A \ \wedge \ M, \sigma' \models \texttt{intl} \ \wedge \ Rng(\phi) \subseteq LocRchbl(\sigma) \quad \Longrightarrow \quad M, \sigma', k \models A$
b. $M, \sigma, k \models A \text{-}\!\!\triangledown (\overline{\alpha}) \ \wedge \ Rng(\phi) \subseteq \overline{\alpha} \quad \Longrightarrow \quad M, \sigma' \models A$
c. $M, \sigma, k \models A \wedge A \text{-}\!\!\triangledown (\overline{\alpha}) \ \wedge \ Rng(\phi) \subseteq \overline{\alpha} \quad \Longrightarrow \quad M, \sigma', k \models A$

Proof.

a. By Lemma G.34, part G.34 and structural induction on the definition of $Stb^+(\_)$.
b. By Lemma G.34, part G.34 and structural induction on the definition of $Stb^+(\_)$.
c. By part b. and Lemma G.3.

□

Lemma G.37 (From called to caller – protected, and variable-free)**.** For any states σ, σ', variable v, address α_v, addresses $\overline{\alpha}$, and statement $stmt$.
If $\sigma' = (\sigma \vartriangle)[v \mapsto \alpha_v][\texttt{cont} \mapsto stmt]$, and $Stb^+(A)$, and $Fv(A) = \emptyset$
then

a. $M, \sigma, k \models A \ \wedge \ k < |\sigma| \ \wedge \ M, \sigma \models A \text{-}\!\!\triangledown \alpha_v \quad \Longrightarrow \quad M, \sigma', k \models A$.

b. $M, \sigma \models A \ \wedge \ \overline{\alpha} \subseteq LocRchbl(\sigma) \implies M, \sigma', k \models A \mathrel{-\!\!\triangledown} (\overline{\alpha})$.

Proof.

a. By Lemma G.35, part a. and structural induction on the definition of $Stb^{+}(\_)$.
b. By Lemma G.35, 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.38 (From caller to internal called)**.** For any assertion A_{in}, states σ, σ', variables $\overline{v_1}$, $\overline{v_2}$, $\overline{v_3}$, $\overline{v_4}$, $\overline{v_6}$, and frame ϕ.
If

(i) $Stb^{+}(A_{in})$,
(ii) $Fv(A_{in}) = \overline{v_1}; \overline{v_2}$[13], $Fv(A_{in}[\overline{v_3/v_1}]) = \overline{v_3}; \overline{v_4}$, $\overline{v_6} \triangleq \overline{v_2} \cap \overline{v_3}; \overline{v_4}$,
(iii) $\sigma' = \sigma \triangledown \phi \ \wedge \ Rng(\phi) = \overline{\lfloor v_3 \rfloor_{\sigma}}$
(iv) $\overline{\lfloor v_1 \rfloor_{\sigma'}} = \overline{\lfloor v_3 \rfloor_{\sigma}}$,

then

a. $M, \sigma, k \models A_{in}[\overline{v_3/v_1}] \ \wedge \ M, \sigma' \models \texttt{intl} \implies M, \sigma', k \models A_{in}[\overline{\lfloor v_6 \rfloor_{\sigma}/v_6}]$
b. $M, \sigma, k \models (A_{in}[\overline{v_3/v_1}]) \mathrel{-\!\!\triangledown} (\overline{v_3}) \implies M, \sigma' \models A_{in}\overline{\lfloor v_6 \rfloor_{\sigma}/v_6}]$.

Discussion of Lemma. In lemma G.38, 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 $\overline{v_1}$ stand for the formal parameters of the method, and $\overline{v_3}$ stand for the actual parameters of the call. Therefore, $\overline{v_1}$ is the domain of the new frame, and $\overline{\sigma v_3}$ is its range. The variables $\overline{v_6}$ are the free variables of A_{in} which are not in $\overline{v_1}$ – *c.f.* Lemma G.29 part (7). Therefore if (a.) the callee is internal, and $A_{in}[\overline{v_3/v_1}]$ holds at the call point, or if (b.) $(A_{in}[\overline{v_3/v_1}]) \mathrel{-\!\!\triangledown} (\overline{v_3})$ holds at the call point, then $A_{in}[\overline{.../v_6}1]$ holds right after pushing ϕ onto the stack. Notice the difference in the conclusion in (a.) and (b.): in the first case we have scoped satisfaction, while in the second case we only have shallow satisfaction.

Proof.
We will use $\overline{\alpha_1}$ as short for $\{\lfloor v_1 \rfloor_{\sigma'}$, and $\overline{\alpha_3}$ as short for $\overline{\lfloor v_3 \rfloor_{\sigma}}$.
We aslo define $\overline{v_{6,1}} \triangleq \overline{v_2} \cap \overline{v_3}$, $\overline{\alpha_{6,1}} \triangleq \overline{v_{6,1}}[\overline{\lfloor v_6 \rfloor_{\sigma}/v_6}]$
We establish that
(*) $A_{in}[\overline{v_3/v_1}][\overline{\lfloor v_3 \rfloor_{\sigma}/v_3}] \overset{\text{txt}}{=} A_{in}[\overline{\alpha_1/v_1}][\overline{\alpha_{6,1}/v_{6,1}}]$
This holds by By Lemma G.31 and assumption (iv) of the current lemma.
And we define $\overline{v_{6,2}} \triangleq \overline{v_2} \setminus \overline{v_3}$, $\overline{\alpha_{6,2}} \triangleq \overline{v_{6_2}}[\overline{\lfloor v_6 \rfloor_{\sigma}/v_6}]$.

a. Assume
$M, \sigma, k \models A_{in}[\overline{v_3/v_1}]$. By Lemma 6.1 part 1 this implies that
$M, \sigma, k \models A_{in}[\overline{v_3/v_1}][\overline{\alpha_3/v_3}]$ By (*) from above we have
$M, \sigma, k \models A_{in}[\overline{\alpha_1/v_1}[\overline{\alpha_{6,1}/v_{6,1}}]$
The above, and Lemma 6.1 part 1 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.36 part a. we also obtain
$M, \sigma', k \models A_{in}[\overline{\alpha_1/v_1}][\overline{\alpha_{6,1}/v_{6,1}}][\overline{\alpha_{6,2}/v_{6,2}}]$

[13] As we said earlier. this gives also that the variable sequences are pairwise disjoint, *i.e.* $\overline{v_1} \# \overline{v_2}$.

With 6.1 part 1 the above gives
$M, \sigma', k \models A_{in}[\overline{\lfloor v_1 \rfloor_{\sigma'}/v_1}][\overline{\lfloor v_6 \rfloor_{\sigma}/v_6}]$
By Lemma 6.1 part 1, we obtain
$M, \sigma', k \models A_{in}[\overline{\lfloor v_6 \rfloor_{\sigma}/v_6}]$

b. Assume
$M, \sigma, k \models (A_{in}[\overline{v_3/v_1}]) \mathrel{-\!\!\triangledown} (\overline{v_3})$. By Lemma 6.1 part 1 this implies that
$M, \sigma, k \models ((A_{in}[\overline{v_3/v_1}]) \mathrel{-\!\!\triangledown} (\overline{v_3}))[\overline{\alpha_3/v_3}]$ which implies that
$M, \sigma, k \models (A_{in}[\overline{v_3/v_1}][\overline{\alpha_3/v_3}]) \mathrel{-\!\!\triangledown} (\overline{\alpha_3})$ By (*) from above we have
$M, \sigma, k \models (A_{in}[\overline{\alpha_1/v_1}][\overline{\alpha_{6,1}/v_{6,1}}]) \mathrel{-\!\!\triangledown} (\overline{\alpha_3})$
The above, and Lemma 6.1 part 1 give that
$M, \sigma, k \models ((A_{in}[\overline{\alpha_1/v_1}][\overline{\alpha_{6,1}/v_{6,1}}]) \mathrel{-\!\!\triangledown} (\overline{\alpha_3}))[\overline{\alpha_{6,2}/v_{6,2}}]$
And the above gives
$M, \sigma, k \models (A_{in}[\overline{\alpha_1/v_1}][\overline{\alpha_{6,1}/v_{6,1}}][\overline{\alpha_{6,2}/v_{6,2}}]) \mathrel{-\!\!\triangledown} (\overline{\alpha_3})$
The assertion above is variable-free.Therefore, by Lemma G.36 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 6.1 part 1, and Lemma 6.1 part 1, and obtain
$M, \sigma' \models A_{in}[\overline{\lfloor v_6 \rfloor_{\sigma}/v_6}]$

□

Lemma G.39 (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) $\sigma' = \sigma \triangledown \phi \quad \wedge \quad Rng(\phi) = \overline{\lfloor v_3 \rfloor_{\sigma}}$,

then

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

Proof. a. Assume that
$M, \sigma, k \models A_{in} \mathrel{-\!\!\triangledown} (\overline{v_3})$
By Lemma 6.1 part 1 this implies that
$M, \sigma, k \models A_{in} \mathrel{-\!\!\triangledown} (\lfloor v_3 \rfloor_{\sigma})$.
We now have a variable-free assertion, and by Lemma G.36, part b., we obtain
$M, \sigma' \models A_{in}$.

b. Assume that
$M, \sigma, k \models A_{in} \wedge A_{in} \mathrel{-\!\!\triangledown} (\overline{v_3})$
By Lemma 6.1 part 1 this implies that
$M, \sigma, k \models A_{in} \wedge A_{in} \mathrel{-\!\!\triangledown} (\lfloor v_3 \rfloor_{\sigma})$.
We now have a variable-free assertion, and by Lemma G.36, part b., we obtain
$M, \sigma', k \models \models A_{in}$.

□

Discussion of Lemma G.39. In this lemma, as in lemma G.38, 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 $\overline{\lfloor v_3 \rfloor_\sigma}$ is the range of the new frame. Therefore if (a.) the adapted version, $A_{in} \mathrel{-\!\!\triangledown} (\overline{v_3})$, 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 scoped satisfaction, the conclusion promises only weak satisfaction. Moreover, if (b.) the adapted as well as the unadapted version, $A_{in} \wedge A_{in} \mathrel{-\!\!\triangledown} (\overline{v_3})$ 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 scoped satisfaction.

Lemma G.40 (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) $\overline{\lfloor v_5 \rfloor_{\sigma'}}, \lfloor res \rfloor_\sigma \subseteq LocRchbl(\sigma) \;\wedge\; M, \sigma' \models \texttt{intl}.$
(iv) $\sigma' = (\sigma \vartriangle)[u \mapsto \lfloor res \rfloor_\sigma][\texttt{cont} \mapsto stmt] \;\wedge\; \overline{\lfloor v_1 \rfloor_\sigma} = \overline{\lfloor v_3 \rfloor_{\sigma'}}.$

then

a. $M, \sigma, k \models A_{out} \;\wedge\; (A_{out} \mathrel{-\!\!\triangledown} res) \;\wedge\; |\sigma'| \geq k \quad \Longrightarrow \quad M, \sigma', k \models A_{out}[\overline{v_3/v_1}]$.
b. $M, \sigma \models A_{out} \quad \Longrightarrow \quad M, \sigma', k \models (A_{out}[\overline{v_3/v_1}]) \mathrel{-\!\!\triangledown} \overline{v_5}.$

Discussion of Lemma G.40. 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 $\overline{\lfloor v_1 \rfloor_\sigma} = \overline{\lfloor v_3 \rfloor_{\sigma'}}$. 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 ($(A_{out}[\overline{v_3/v_1}]) \mathrel{-\!\!\triangledown} \overline{v_5}$), at the return point.

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

a. Assume that
$M, \sigma, k \models A_{out} \wedge A_{out} \mathrel{-\!\!\triangledown} res$
By Lemma 6.1 part 1 this implies that
$M, \sigma, k \models A_{out}[\overline{\alpha_1/v_1}] \wedge (A_{out}[\overline{\alpha_1/v_1}]) \mathrel{-\!\!\triangledown} \alpha$.
We now have a variable-free assertion, and by Lemma G.37, part a., we obtain
$M, \sigma, k \models A_{out}[\overline{\alpha_1/v_1}]$.
By Lemma 6.1 part 1, and because $\overline{\lfloor v_1 \rfloor_\sigma} = \overline{\lfloor v_3 \rfloor_{\sigma'}}$ this implies that
$M, \sigma, k \models A_{out}[\overline{v_3/v_1}]$.
b. Assume that
$M, \sigma \models A_{out}$
By Lemma 6.1 part 1 this implies that
$M, \sigma \models A_{out}[\overline{\alpha_1/v_1}]$

We now have a variable-free assertion, and by Lemma G.37, part b., we obtain
$M, \sigma', k \models A_{out}[\overline{\alpha_1/v_1}] \mathrel{-\!\!\triangledown} \overline{\alpha}_5$
By Lemma 6.1 part 1, and because $\overline{\lfloor v_1 \rfloor_\sigma} = \overline{\lfloor v_3 \rfloor_{\sigma'}}$ and $\alpha_5 = \overline{\lfloor v_5 \rfloor_{\sigma'}}$, we obtain
$M, \sigma', k \models A_{out}[\overline{v_3/v_1}] \mathrel{-\!\!\triangledown} \overline{v}_5$

□

Lemma G.41 (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) $\overline{\lfloor v_5 \rfloor_{\sigma'}}, \lfloor res \rfloor_\sigma \subseteq LocRchbl(\sigma)$.
(iv) $\sigma' = (\sigma \vartriangle)[u \mapsto \lfloor res \rfloor_\sigma][\texttt{cont} \mapsto stmt]$.

then

a. $M, \sigma \models A_{out} \implies M, \sigma', k \models A_{out} \mathrel{-\!\!\triangledown} (\overline{v_5})$.
b. $M, \sigma, k \models A_{out} \ \wedge\ |\sigma'| \geq k\, \lfloor v_5 \rfloor_{\sigma'} \implies M, \sigma', k \models A_{out} \ \wedge\ A_{out} \mathrel{-\!\!\triangledown} (\overline{v_5})$

Proof.

a. Assume that
$M, \sigma \models A_{out}$
Since A_{out} is a variable-free assertion, by Lemma ???? part ??? we obtain
$M, \sigma', k \models A_{out} \mathrel{-\!\!\triangledown} (\overline{\lfloor v_5 \rfloor_{\sigma'}})$.
By Lemma 6.1 part 1, we obtain
$M, \sigma', k \models A_{out} \mathrel{-\!\!\triangledown} (\overline{v_5})$
b. Assume that

□

Discussion of lemma G.41. , Similarly to lemma G.40, 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 it its adapted version also holds after popping frame ($A_{out} \mathrel{-\!\!\triangledown} \overline{v_5}$). 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 ($A_{out} \mathrel{-\!\!\triangledown} \overline{v_5}$) at the return point; while in (b.), we require *deep satisfaction at the called*, and obtain deep satisfaction of the *conjuction* of the *unadapted* with the *adapted* version ($A_{out} \ \wedge\ A_{out} \mathrel{-\!\!\triangledown} \overline{v_5}$), at the return point.

G.15 Use of Lemmas G.38-G.39

As we said earlier, Lemmas G.38-G.39 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.38 part (a.) and Lemma G.40 part (a.). In the proof of soundness of Call_Int_Adapt we will use Lemma G.38 part (b.) and Lemma G.40 part (b.). In the proof of soundness of Call_Ext_Adapt we will use Lemma G.39 part (a.) and Lemma G.41 part (a.). And finally, in the proof of soundness of Call_Ext_Adapt_Strong we will use Lemma G.39 part (b.) and Lemma G.41 part (b.).

G.16 Proof of Theorem 9.3 – part (A)

Begin Proof

Take any M, $\overline{M}$, with

(1) $\vdash M$.

We will prove that

(*) $\forall \sigma, A, A', A''.$
$[\, M \vdash \{\, A \,\}\, \sigma.\texttt{cont}\{\, A' \,\} \parallel \{\, A'' \,\} \implies M \models_{\circ} \{\, A \,\}\, stmt\{\, A' \,\} \parallel \{\, A'' \,\}\,].$

by induction on the well-founded ordering $\_ \ll_{M,\overline{M}} \_$.

Take σ, A, A', A'', $\overline{z}$, $\overline{w}$, $\overline{\alpha}$, σ', σ'' arbitrary. Assume that

(2) $M \vdash \{\, A \,\}\, \sigma.\texttt{cont}\{\, A' \,\} \parallel \{\, A'' \,\}$

(3) $\overline{w} = Fv(A) \cap dom(\sigma), \quad \overline{z} = Fv(A) \setminus dom(\sigma)$[14]

(4) $M, \sigma, k \models A[\overline{\alpha/z}]$

To show

(**) $\overline{M} \cdot M;\ \sigma \rightsquigarrow^{*}_{fin} \sigma' \implies M, \sigma', k \models A'[\overline{\alpha/z}]$

(***) $\overline{M} \cdot M;\ \sigma \rightsquigarrow^{*} \sigma'' \implies M, \sigma'', k \models \texttt{extl} \to A''[\overline{\alpha/z}][\overline{\lfloor w \rfloor_{\sigma}/w}]$

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 9.2.

SEQU Therefore, there exist statements $stmt_1$ and $stmt_2$, and assertions A_1, A_2 and A'', so that $A_1 \overset{\text{txt}}{=} A$, and $A_2 \overset{\text{txt}}{=} A'$, and $\sigma.\texttt{cont} \overset{\text{txt}}{=} stmt_1; stmt_2$,. We apply lemma G.24, 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.3 part 4 and axiom G.1

CALL_INT Therefore, there exist u, y_o, C, $\overline{y}$, A_{pre}, A_{post}, and A_{mid}, such that

(5) $\sigma.\texttt{cont} \overset{\text{txt}}{=} u := y_0.m(\overline{y})$,

(6) $\vdash \ M : \{\, A_{pre} \,\}\, D::m(\overline{x : D})\, \{\, A_{post} \,\} \parallel \{A_{mid}\}$,

(7) $A \overset{\text{txt}}{=} y_0 : D, \overline{y : D} \ \wedge\ A_{pre}[y_0, \overline{y}/\texttt{this}, \overline{x}]$,
$A' \overset{\text{txt}}{=} A_{post}[y_0, \overline{y}, u/\texttt{this}, \overline{x}, \texttt{res}]$,
$A'' \overset{\text{txt}}{=} A_{mid}$.

Also,

(8) $\overline{M} \cdot M;\ \sigma \rightsquigarrow \sigma_1$,

where

(8a) $\sigma_1 \triangleq (\sigma \triangledown (\texttt{this} \mapsto \lfloor y_0 \rfloor_{\sigma}, \overline{x \mapsto \lfloor y \rfloor_{\sigma}})[\texttt{cont} \mapsto stmt_m]$,

(8b) $\texttt{mBody}(m, D, M) = \overline{y : D}\{\, stmt_m \,\}$.

We define the shorthands:

(9) $A_{pr} \triangleq \texttt{this} : D, \overline{x : D} \ \wedge\ A_{pre}$.

(9a) $A_{pra} \triangleq \texttt{this} : D, \overline{x : D} \ \wedge\ A_{pre} \ \wedge\ A_{pre} \multimap\!\!\triangledown (y_0, \overline{y})$.

(9b) $A_{poa} \triangleq A_{post} \ \wedge\ A_{post} \multimap\!\!\triangledown \texttt{res}$.

By (1), (6), (7), (9), and definition of $\vdash M$ in Section 8.4 rule METHOD and we obtain

(10) $M \vdash \{\ A_{pra}\ \}\ stmt_m\{\ A_{poa}\ \} \parallel \{\, A_{mid} \,\}$.

From (8) we obtain

[14] Remember that $dom(\sigma)$ is the set of variables defined in the top frame of σ

(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 $...\sigma_1 \models A_{pr}[../fv(A_{pr}) \setminus dom(\sigma_1)]$. To that aim we will apply Lemma G.38 part a. on (4), (8a) and (9). For this, we take

(12) $\overline{v_1} \triangleq \texttt{this}, \overline{x}, \quad \overline{v_2} \triangleq Fv(A_{pr}) \setminus \overline{v_1}, \quad \overline{v_3} \triangleq y_0, \overline{y}, \quad \overline{v_4} \triangleq Fv(A) \setminus \overline{v_3}$

These definitions give that

(12a) $A \overset{\text{txt}}{=} A_{pr}[\overline{v_3/v_1}]$,

(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.29 part (7), we obtain that

(12d) $\overline{v_2} = \overline{y_r}; \overline{v_4}, \quad$ where $\overline{y_r} \triangleq \overline{v_2} \cap \overline{v_3}$

Furthermore, (8a), and (12) give that:

(12e) $\overline{\lfloor v_1 \rfloor_{\sigma_1} = \lfloor v_3 \rfloor_{\sigma}}$

Then, (4), (12a), (12c) and (12f) give that

(13) $M, \sigma, k \models A_{pr}[\overline{v_3/v_1}][\overline{\alpha/z}]$

Moreover, we have that $\overline{z} \# \overline{v_3}$. From Lemma G.29 part (10) we obtain $\overline{z} \# \overline{v_1}$. And, because $\overline{\alpha}$ are addresses wile $\overline{v_1}$ are variables, we also have that $\overline{\alpha} \# \overline{v_1}$. These facts, together with Lemma G.29 part (9) give that

(13a) $A_{pr}[\overline{v_3/v_1}][\overline{\alpha/z}] \overset{\text{txt}}{=} A_{pr}[\overline{\alpha/z}][\overline{v_3/v_1}]$

From (13a) and (13), we obtain

(13b) $M, \sigma, k \models A_{pr}[\overline{\alpha/z}][\overline{v_3/v_1}]$

From (4), (8a), (12a)-(12e) we see that the requirements of Lemma G.38 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) $\overline{v_6} \triangleq y_r; (\overline{v_4} \setminus \overline{z})$ which, with (12d) also gives: $\overline{v_2} = \overline{v_6}; \overline{z}$

We apply Lemma G.38 part a. on (13b), (13c) and obtain

(14a) $M, \sigma_1, k \models A_{pr}[\overline{\alpha/z}][\overline{\lfloor v_6 \rfloor_{\sigma}/v_6}]$.

Moreover, we have the $M, \sigma_1 \models \texttt{intl}$. We apply lemma 6.8, and obtain

(14b) $M, \sigma_1, k \models A_{pr}[\overline{\alpha/z}][\overline{\lfloor v_6 \rfloor_{\sigma}/v_6}] \wedge A_{pr} \neg\triangledown(\texttt{this}, \overline{y})$.

With similar re-orderings to earlier, we obain

(14b) $M, \sigma_1, k \models A_{pra}[\overline{\alpha/z}][\overline{\lfloor v_6 \rfloor_{\sigma}/v_6}]$.

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 $Fv(A_{pr}) \setminus dom(\sigma_1)$, as well as the value of $Fv(A_{pr}) \cap dom(\sigma_1)$. This is what we do next. From (8a) we have that

(15a) $dom(\sigma_1) = \{\texttt{this}, \overline{x}\}$.

This, with (12) and (12b) gives that

(15b) $Fv(A_{pra}) \cap dom(\sigma_1) = \overline{v_1}$.

(15c) $Fv(A_{pra}) \setminus dom(\sigma_1) = \overline{v_2}$.

Moreover, (12d) and (13d) give that

(15d) $Fv(A_{pra}) \setminus dom(\sigma_1) = \overline{z_2} = \overline{z}; \overline{v_6}$.

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) $\sigma' = (\sigma'_1 \vartriangle)[u \mapsto \lfloor \texttt{res} \rfloor_{\sigma'_1}][\texttt{cont} \mapsto \epsilon]$.

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 $\ldots\sigma' \models \ldots A'$. We now want to be able to apply Lemma G.40, 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) $\overline{v_{1,a}} \triangleq \overline{v_1}, \texttt{res}, \quad \overline{v_{3,a}} \triangleq \overline{v_3}, u.$

The wellformedness condition for specifications requires that $Fv(A_{post}) \subseteq Fv(A_{pr}) \cup \{\texttt{res}\}$. 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) $\overline{\lfloor v_{3,a} \rfloor_{\sigma'}} = \overline{\lfloor v_{1,a} \rfloor_{\sigma_1'}}$.

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.40, 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 4.4 we have that $\overline{\lfloor v_6 \rfloor_\sigma} = \overline{\lfloor v_6 \rfloor_{\sigma'}}$, we obtain

(21) $M, \sigma', k \models (A_{poa})[y_0, \overline{y}, u/\texttt{this}, \overline{x}, \texttt{res}][\overline{\alpha/z}]$..

With (7) we conclude.

Proving $(***)$. Take a σ''. Assume that

(15) $\overline{M} \cdot M;\ \sigma \rightsquigarrow^* \sigma''$

(16) $\overline{M} \cdot M, \sigma'' \models \texttt{extl}$.

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

(β') $M, \sigma'', k \models A_{mid}[\overline{\alpha/z}][\overline{\lfloor w \rfloor_\sigma / w}]$.

and using (7) we are done.

Call_Int_Adapt is similar to Call_Int. We highlight the differences in green . Therefore, there exist $u, y_o, C, \overline{y}, A_{pre}, A_{post}$, and A_{mid}, such that

(5) $\sigma.\texttt{cont} \overset{\text{txt}}{=} u := y_0.m(\overline{y})$,

(6) $\vdash \quad M \ : \ \{\, A_{pre} \,\}\, D :: m(\overline{x : D})\, \{\, A_{post} \,\} \ \parallel \ \{A_{mid}\}$,

(7) $A \overset{\text{txt}}{=} y_0 : D, \overline{y : D} \ \wedge \ (A_{pre}[y_0/\texttt{this}]) \multimap\!\!\triangledown (y_0, \overline{y})$,

$A' \overset{\text{txt}}{=} (A_{post}[y_0/\texttt{this}, u/res]) \multimap\!\!\triangledown (y_0, \overline{y})$,

$A'' \overset{\text{txt}}{=} A_{mid}$.

Also,

(8) $\overline{M} \cdot M;\ \sigma \rightsquigarrow \sigma_1$,

where

(8a) $\sigma_1 \triangleq (\sigma \triangledown (\texttt{this} \mapsto \lfloor y_0 \rfloor_\sigma, \overline{x \mapsto \lfloor y \rfloor_\sigma})[\texttt{cont} \mapsto stmt_m]$,

(8b) $\texttt{mBody}(m, D, M) = \overline{y : D}\{\ stmt_m\ \}$.

We define the shorthand:

(9) $A_{pr} \triangleq \texttt{this} : D, \overline{x : D} \wedge A_{pre}$.

(9a) $A_{pra} \triangleq \texttt{this} : D, \overline{x : D} \wedge A_{pre} \wedge A_{pre} \mathrel{-\!\!\triangledown} (y_0, \overline{y})$.

(9b) $A_{poa} \triangleq A_{post} \wedge A_{post} \mathrel{-\!\!\triangledown} \texttt{res}$.

By (1), (6), (7), (9), and definition of $\vdash M$ in Section 8.4, we obtain

(10) $M \vdash \{\ A_{pra}\ \}\ stmt_m\{\ A_{poa}\ \} \parallel \{\ A_{mid}\ \}$.

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 $...\sigma_1 \models A_{pr}[../fv(A_{pr}) \setminus dom(\sigma_1)]$. To that aim we will apply Lemma G.38 part b. on (4), (8a) and (9). For this, we take

(12) $\overline{v_1} \triangleq \texttt{this}, \overline{x}$, $\overline{v_2} \triangleq Fv(A_{pr}) \setminus \overline{v_1}$, $\overline{v_3} \triangleq y_0, \overline{y}$, $\overline{v_4} \triangleq Fv(A) \setminus \overline{v_3}$

These definitions give that

(12a) $A \overset{\text{txt}}{=} (A_{pr}[\overline{v_3/v_1}]) \mathrel{-\!\!\triangledown} (\overline{v_3})$,

(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.29 part (7), we obtain that

(12d) $\overline{v_2} = \overline{y_r}; \overline{v_4}$, where $\overline{y_r} \triangleq \overline{v_2} \cap \overline{v_3}$

Furthermore, (8a), and (12) give that:

(12e) $\overline{\lfloor v_1 \rfloor_{\sigma_1}} = \lfloor v_3 \rfloor_{\sigma}$

Then, (4), (12a) give that

(13) $M, \sigma, k \models A_{pr}[\overline{v_3/v_1}] \mathrel{-\!\!\triangledown} (\overline{v_3})$

Moreover, we have that $\overline{z} \# \overline{v_3}$. From Lemma G.29 part (10) we obtain $\overline{z} \# \overline{v_1}$. And, because $\overline{\alpha}$ are addresses wile $\overline{v_1}$ are variables, we also have that $\overline{\alpha} \# \overline{v_1}$. These facts, together with Lemma G.29 part (9) give that

(13a) $A_{pr}[\overline{v_3/v_1}][\overline{\alpha/z}] \overset{\text{txt}}{=} A_{pr}[\overline{\alpha/z}][\overline{v_3/v_1}]$

From (13a) and (13), we obtain

(13b) $M, \sigma, k \models (A_{pr}[\overline{\alpha/z}][\overline{v_3/v_1}]) \mathrel{-\!\!\triangledown} (\overline{v_3})$

From (4), (8a), (12a)-(12e) we see that the requirements of Lemma G.38 where we take A_{in} to be $A_{pr}[\overline{\alpha/z}]$. We use the definition of y_r in (12d), and define

(13c) $\overline{v_6} \triangleq y_r; (\overline{v_4} \setminus \overline{z})$, this also gives that $\overline{v_2} = \overline{v_6}; \overline{z}$

We apply Lemma G.38 part b. on (13b), (13c) and obtain

(14) $M, \sigma_1 \models A_{pr}[\overline{\alpha/z}][\overline{\lfloor v_6 \rfloor_{\sigma}/v_6}]$.

which is equivalent to

(14aa) $M, \sigma_1, |\sigma_1| \models A_{pr}[\overline{\alpha/z}][\overline{\lfloor v_6 \rfloor_{\sigma}/v_6}]$.

By similar argument as in the previous case, we deduce that

(14a) $M, \sigma_1, |\sigma_1| \models A_{pra}[\overline{\alpha/z}][\overline{\lfloor v_6 \rfloor_{\sigma}/v_6}]$.

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 $Fv(A_{pr}) \setminus dom(\sigma_1)$, as well as the value of $Fv(A_{pr}) \cap dom(\sigma_1)$. This is what we do next. From (8a) we have that

(15a) $dom(\sigma_1) = \{\texttt{this}, \overline{x}\}$.

This, with (12) and (12b) gives that

(15b) $Fv(A_{pra}) \cap dom(\sigma_1) = \overline{v_1}$.

(15c) $Fv(A_{pra}) \setminus dom(\sigma_1) = \overline{v_2}$.

Moreover, (12d) and (13d) give that

(15d) $Fv(A_{pra}) \setminus dom(\sigma_1) = \overline{z_2} = \overline{z}; \overline{v_6}$.

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) $\sigma' = (\sigma'_1 \vartriangle)[u \mapsto \lfloor \texttt{res} \rfloor_{\sigma'_1}][\texttt{cont} \mapsto \epsilon]$.

We now apply the induction hypothesis on (14a), (16), (15d), and obtain

(18) $M, \sigma'_1, |\sigma_1| \models (A_{poa})[\overline{\alpha/z}][\overline{\lfloor v_6 \rfloor_\sigma / v_6}]$.

We now want to obtain something of the form $...\sigma' \models ...A'$. For this, we want to be able to apply Lemma G.40, part b. on (18). Therefore, we define

(18a) $A_{out} \triangleq A_{post}[\overline{\alpha/z}][\overline{\lfloor v_6 \rfloor_\sigma / v_6}]$

(18b) $\overline{v_{1,a}} \triangleq \overline{v_1}, \texttt{res}, \quad \overline{v_{3,a}} \triangleq \overline{v_3}, u, \quad \overline{v_5} \triangleq \overline{v_3}$

The wellformedness condition for specifications requires that $Fv(A_{post}) \subseteq Fv(A_{pr}) \cup \{\texttt{res}\}$. 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) $\overline{\lfloor v_{3,a} \rfloor_{\sigma'}} = \overline{\lfloor v_{1,a} \rfloor_{\sigma'_1}}$.

From (16) we obtain that $|\sigma'_1| = |\sigma_1|$. This, together with (18) gives

(19c) $M, \sigma'_1 \models (A_{poa})[\overline{\alpha/z}][\overline{\lfloor v_6 \rfloor_\sigma / v_6}]$.

From (16), (18b) and by the fact that we never overwrite the values of the formal parameters, we have that $\overline{\lfloor v_5 \rfloor_\sigma} = \overline{\lfloor v_1 \rfloor_{\sigma_1}} = \overline{\lfloor v_1 \rfloor_{\sigma'_1}} = \overline{\lfloor v_5 \rfloor_{\sigma'}}$, and this also gives that

(19d) $\overline{\lfloor v_5 \rfloor_{\sigma'}} \subseteq LocRchbl(\sigma'_1)$

We now apply Lemma G.40, part b., and obtain

(20) $M, \sigma' \models A_{out}[\overline{v_{3,a}/v_{1,a}}] \mathrel{-\!\!\triangledown} (\overline{v_3})$.

Adaptation gives stable assertions - *c.f.* lemma **??**.Moreover, any stable assertion which holds at the top-most scope (frame) also holds at any earlier scope (frame) – *c.f.* lemma G.3, part 3. Therefore, (20) also gives

(20a) $M, \sigma', k \models A_{out}[\overline{v_{3,a}/v_{1,a}}] \mathrel{-\!\!\triangledown} (\overline{v_3})$.

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

(20b) $M, \sigma', k \models A_{post}[\overline{v_{3,a}/v_{1,a}}][\overline{\alpha/z}][\overline{\lfloor v_6 \rfloor_\sigma / v_6}] \mathrel{-\!\!\triangledown} (\overline{v_3})$.

By Lemma 4.4 we have that $\overline{\lfloor v_6 \rfloor_\sigma} = \overline{\lfloor v_6 \rfloor_{\sigma'}}$, and so we obtain

(20c) $M, \sigma', k \models A_{post}[\overline{v_{3,a}/v_{1,a}}][\overline{\alpha/z}] \mathrel{-\!\!\triangledown} (\overline{v_3})$.

Moreover, $\overline{v_3} \# \overline{z}$. We apply lemma G.30, and obtain:

(20d) $M, \sigma', k \models (A_{post}[\overline{v_{3,a}/v_{1,a}}] \mathrel{-\!\!\triangledown} (\overline{v_3}))[\overline{\alpha/z}]$.

By (20d), (18b), we obtain

(21) $M, \sigma', k \models (A_{post}[y_0, \overline{y}, u/\texttt{this}, \overline{x}, \texttt{res}] \mathrel{-\!\!\triangledown} (y_0, \overline{y}))[\overline{\alpha/z}]$.

With (7) we conclude.

Proving (***). This is similar to the proof for Call_Int.

CALL_EXT_ADAPT is in some parts, similar to CALL_INT, and CALL_INT_ADAPRT. We highlight the differences in green .

Therefore, there exist u, y_o, $\overline{C}$, D, $\overline{y}$, ands A_{inv}, such that

(5) $\sigma.\texttt{cont} \overset{\text{txt}}{=} u := y_0.m(\overline{y})$,

(6) $\vdash \ M \ : \ \forall \overline{x : C}.\{A_{inv}\}$,

(7) $A \overset{\text{txt}}{=} y_0 : \texttt{external}, \overline{x : C} \ \wedge \ A_{inv} \multimap \triangledown (y_0, \overline{y})$,

$A' \overset{\text{txt}}{=} A_{inv} \multimap \triangledown (y_0, \overline{y})$,

$A'' \overset{\text{txt}}{=} A_{inv}$.

Also,

(8) $\overline{M} \cdot M; \sigma \rightsquigarrow \sigma_a$,

where

(8a) $\sigma_a \triangleq (\sigma \triangledown (\texttt{this} \mapsto \lfloor y_0 \rfloor_\sigma, \overline{p \mapsto \lfloor y \rfloor_\sigma})[\texttt{cont} \mapsto stmt_m]$,

(8b) $\texttt{mBody}(m, D, \overline{M}) = \overline{p : D}\{ \ stmt_m \ \}$.

(8c) D is the class of $\lfloor y_0 \rfloor_\sigma$, and D is external.

By (7), and well-formedness of module invariants, we obtain

(9a) $Fv(A_{inv}) \subseteq \overline{x}$,

(9a) $Fv(A) = y_0, \overline{y}, \overline{x}$

By Barendregt, we also obtain that

(10) $dom(\sigma) \ \# \overline{x}$

This, together with (3) gives that

(10) $\overline{z} = \overline{x}$

From (4), (7) and the defiinition of satisfaction we obtain

(10) $M, \sigma, k \models (\overline{x : C} \wedge A_{inv} \triangledown y_0, \overline{y})[\overline{\alpha/z}]$.

The above gives that

(10a) $M, \sigma, k \models (\ (\overline{x : C})[\overline{\alpha/z}] \ \wedge \ (A_{inv}[\overline{\alpha/z}]) \ ) \triangledown y_0, \overline{y}$.

We take A_{in} to be $(\overline{x : C})[\overline{\alpha/z}] \ \wedge \ (A_{inv}[\overline{\alpha/z}])$, and apply Lemma G.39, part a.. This gives that

(11) $M, \sigma_a \models (\overline{x : C})[\overline{\alpha/z}] \ \wedge \ A_{inv}[\overline{\alpha/z}]$

Proving $(**)$. We shall use the short hand

(12) $A_o \triangleq \overline{\alpha : C} \wedge A_{inv}[\overline{\alpha/z}]$.

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) $\overline{M} \cdot M$; $\sigma_a \rightsquigarrow^*_{fin} \sigma_b$

(17) $\sigma' = (\sigma_b \vartriangle)[u \mapsto \lfloor \texttt{res} \rfloor_{\sigma'_1}][\texttt{cont} \mapsto \epsilon]$.

By Lemma G.26 part 1, and Def. G.25, we obtain that there exists a sequence of states σ_1, ... σ_n, such that

(17) $(\overline{M} \cdot M, \ \sigma_a); \sigma_a \rightsquigarrow^*_{e,p} \sigma_b \ \mathbf{pb} \ \sigma_1 ... \sigma_n$

By Def. G.25, the states σ_1, ... σ_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) $\forall i \in 1..n.$

$[\ M \ \vdash \ \{ \texttt{this} : D_i, \overline{p_i : D_i}, \overline{x : C} \ \wedge \ A_{inv} \}\sigma_i.\texttt{cont}\{ \ A_{inv,r} \ \} \ \ \| \ \ \{ \ A_{inv} \ \} \ ]$

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 $A_{inv,r} \triangleq A_{inv} \multimap \texttt{res}$.

Moreover, (17) gives that

(19) $\forall i \in 1..n.[\quad M \cdot \overline{M};\ \sigma \rightsquigarrow^* \sigma_i \quad]$

From (18) and (19) we obtain

(20) $\forall i \in [1..n].$

$$[\ (\texttt{this} : D_i, \overline{p_i : D_i}, \overline{x : C} \ \wedge\ A_{inv}, \sigma_i, A_{inv,r}, A_{inv}) \ll_{M,\overline{M}} (A, \sigma, A', A'')\ ]$$

We take

(21) $k = |\sigma_a|$ By application of the induction hypothesis on (20) we obtain that

(22) $\forall i \in [1..n].\forall \sigma_f.[\ M, \sigma_i, k \models A_o \ \wedge\ M \cdot \overline{M};\ \sigma_i \rightsquigarrow^*_{fin} \sigma_f \implies M, \sigma_f, k \models A_o\ ]$

We can now apply Lemma G.27, part 3, and because $|\sigma_a| = |\sigma_b|$, we obtain that

(23) $M, \sigma_b \models A_{inv}[\overline{\alpha/x}]$

We apply Lemma G.41 part a., and obtain

(24) $M, \sigma' \models A_{inv}[\overline{\alpha/x}] \mathrel{-\!\!\triangledown} y_0, \overline{y}$

And since $A_{inv}[\overline{\alpha/x}] \mathrel{-\!\!\triangledown} y_0, \overline{y}$ is stable, and by rearranging, and applying (10), we obtain

(25) $M, \sigma', k \models (A_{inv} \mathrel{-\!\!\triangledown} y_0, \overline{y})[\overline{\alpha/z}]$

Apply (7), and we are done.

Proving $(***)$. Take a σ''. Assume that

(12) $\overline{M} \cdot M;\ \sigma \rightsquigarrow^* \sigma''$

(13) $\overline{M} \cdot M, \sigma'' \models \texttt{extl}$.

We apply lemma 1, part 2 on (12) and see that there are two cases

1st Case $\overline{M} \cdot M; \sigma_a \rightsquigarrow^*_{\not\in,p} \sigma''$

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.27, part 1.

2nd Case for some σ_c, σ_d. we have

$\overline{M} \cdot M; \sigma_a \rightsquigarrow^*_{\not\in,p} \sigma_c \wedge \overline{M} \cdot M;\ \sigma_c \rightsquigarrow \sigma_d \wedge\ M, \sigma_d \models \texttt{pub} \wedge \overline{M} \cdot M;\ \sigma_d \rightsquigarrow^* \sigma'$

We apply similar arguments as in steps (17)-(23) and obtain

(14) $M, \sigma_c \models A_{inv}[\overline{\alpha/x}]$

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 $\overline{z} = \overline{x}$, we obtain:

(14) $M, \sigma'' \models A_{inv}[\overline{\alpha/z}]$

Call_Ext_Adapt_Strong is very similar to Call_Ext_Adaprt. We will summarize the similar steps, and highlight the differences in green .

Therefore, there exist u, y_o, $\overline{C}$, D, $\overline{y}$, ands A_{inv}, such that

(5) $\sigma.\texttt{cont} \overset{\text{txt}}{=} u := y_0.m(\overline{y})$,

(6) $\vdash\ M\ :\ \forall \overline{x : C}.\{A_{inv}\}$,

(7) $A \overset{\text{txt}}{=} y_0 : \texttt{external}, \overline{x : C} \ \wedge\ A_{inv}\ \wedge A_{inv} \mathrel{-\!\!\triangledown} (y_0, \overline{y})$,

$A' \overset{\text{txt}}{=} A_{inv}\ \wedge A_{inv} \mathrel{-\!\!\triangledown} (y_0, \overline{y})$,

$A'' \overset{\text{txt}}{=} A_{inv}$.

Also,

(8) $\overline{M} \cdot M;\ \sigma \rightsquigarrow \sigma_a$,

By similar steps to (8a)-(10) from the previous case, we obtain

(10a) $M, \sigma, k \models A_{inv}[\overline{\alpha/z}] \wedge (\ (\overline{x : C})[\overline{\alpha/z}] \wedge (A_{inv}[\overline{\alpha/z}])\ ) \triangledown y_0, \overline{y}$.

We now apply lemma apply Lemma G.39, part b.. This gives that

(11) $M, \sigma_a, k \models (\ (\overline{x : C})[\overline{\alpha/z}] \wedge A_{inv}[\overline{\alpha/z}] \wedge (\ (\overline{x : C})[\overline{\alpha/z}])\triangledown y_0, \overline{y}\ )$.

TODO rest ...

End Proof

G.17 Proof Sketch of Theorem 9.3 – part (B)

Proof Sketch By induction on the cases for the specification S. If it is a method spec, then the theorem follows from 9.3. If it is a conjunction, then by inductive hypothesis.

The interesting case is $S \overset{\text{txt}}{=} \forall \overline{x : C}.\{A\}$.

Assume that $M, \sigma, k \models A[\overline{\alpha/x}]$, that $M, \sigma \models \texttt{extl}$, that $M \cdot \overline{M}$; $\sigma \rightsquigarrow^* \sigma'$, and that $M, \sigma \models \texttt{extl}$,

We want to show that $M, \sigma', k \models A[\overline{\alpha/x}]$.

Then, by lemma G.26, we obtain that either

(1) $\overline{M} \cdot M; \sigma \overset{*}{\rightsquigarrow}_{\hat{e},p} \sigma'$, or

(2) $\exists \sigma_1, \sigma_2.[\ \overline{M} \cdot M; \sigma \overset{*}{\rightsquigarrow}_{\hat{e},p} \sigma_1 \wedge \overline{M} \cdot M;\ \sigma_1 \rightsquigarrow \sigma_2 \wedge\ M, \sigma_2 \models \texttt{pub} \wedge \overline{M} \cdot M;\ \sigma_2 \rightsquigarrow^* \sigma'\ ]$

In Case (1), we apply G.27, part (3). In order to fulfill the second premise of Lemma G.27, part (3), we make use of the fact that $\vdash M$, apply the rule Method, and theorem 9.3. 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 $M \vdash Enc(A)$, 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 9.3, and obtain $M, \sigma', k \models A[\overline{\alpha/x}]$

End Proof Sketch

H PROVING TAMED 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 if this method includes a method call to an unknown external object (`buyer.pay(...)`).

In this section we use our Hoare logic from Section 8 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.

More generally, given the following scoped invariants,

$S_2 \triangleq \forall \texttt{a} : \texttt{Account}.\{\langle\!\langle \texttt{a.key} \rangle\!\rangle\}$

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

we outline the proof that $M_{good} \vdash S_2$. We also show why $M_{bad} \nvdash S_2$.

We first extend the semantics and the logic to deal with scalars (§H.1). We then rewrite the code of M_{good} so that it adheres to the syntax as defined in Fig. 9 (§H.2). After that, we outline the proofs (§H.3) – these proofs have been mechanized in Coq, and the source code will be submitted as an artefact. Finally, we discuss why $M_{bad} \nvdash S_2$ (§I).

H.1 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 H.1 (Satisfaction of Assertions – Protected From)**.** extending the definition of Def 5.4. We use α to range over addresses, β to range over scalars, and γ to range over addresses or scalars. We define $M, \sigma \models \langle\!\langle \gamma \rangle\!\rangle \nrightarrow \gamma_o$ as:

(1) $M, \sigma \models \langle\!\langle \alpha \rangle\!\rangle \nrightarrow \alpha_o \triangleq$
- $\alpha \neq \alpha_0$, and
- $\forall n \in \mathbb{N}. \forall f_1 ... f_n .. [\ \lfloor \alpha_o.f_1...f_n \rfloor_\sigma = \alpha \implies M, \sigma \models \lfloor \alpha_o.f_1...f_{n-1} \rfloor_\sigma : C \wedge C \in M\ ]$

(2) $M, \sigma \models \langle\!\langle \gamma \rangle\!\rangle \nrightarrow \beta_o \triangleq true$

(3) $M, \sigma \models \langle\!\langle \beta \rangle\!\rangle \nrightarrow \alpha_o \triangleq false$

(4) $M, \sigma \models \langle\!\langle e \rangle\!\rangle \nrightarrow e_o \triangleq$
$\exists \gamma, \gamma_o .[\ M, \sigma, e \hookrightarrow \gamma \wedge M, \sigma, e_0 \hookrightarrow \gamma_0 \wedge M, \sigma \models \langle\!\langle \gamma \rangle\!\rangle \nrightarrow \gamma_o\ ]$

The definition from above gives rise to further cases of protection; we supplement the triples from Fig. 6 with some further inference rules, given in Fig. **??**.

$$M \vdash x : \texttt{int} \rightarrow \langle\!\langle y \rangle\!\rangle \nrightarrow x \quad [\textsc{Prot-Int}_1] \qquad M \vdash x : \texttt{bool} \rightarrow \langle\!\langle y \rangle\!\rangle \nrightarrow x \quad [\textsc{Prot-Bool}_1]$$

$$M \vdash x : \texttt{str} \rightarrow \langle\!\langle y \rangle\!\rangle \nrightarrow x \quad [\textsc{Prot-Str}_1]$$

Fig. 15. Extended Consequence Rules for Protection that include rules for protection involving scalars.

We also introduce a rule for conditionals in Fig. 16, where we expect the obvious syntax and semantics for *Cond*

$$\frac{M \vdash \{\ A \wedge Cond\ \}\ stmt_1\ \{\ A'\ \} \parallel \{\ A''\ \} \qquad M \vdash \{\ A \wedge \neg Cond\ \}\ stmt_2\ \{\ A'\ \} \parallel \{\ A''\ \}}{M \vdash \{\ A\ \}\ \texttt{if}\ Cond\ \texttt{then}\ stmt_1\ \texttt{else}\ stmt_2\ \{\ A'\ \} \parallel \{\ A''\ \}} \ [\textsc{If\_Rule}]$$

Fig. 16. Hoare Quadruple for conditionals

H.2 Expressing the `Shop` example in the syntax from Fig. 9

We now express our example in the syntax of Fig. 9. 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 := ..`

```
module M_good
  ...
  class Shop
    field accnt : Account,
    field invntry : Inventory,
    field clients: ..

    public method buy(buyer:external, anItem:Item, price: int,
            myAccnt: Account, oldBalance: int, newBalance: int, tmp:int) : int
      int price := anItem.price;
      Account myAccnt := this.accnt;
      int oldBalance := myAccnt.blnce;
      tmp := buyer.pay(myAccnt, price)      // external call!
      int newBalance := myAccnt.blnce;
      if (newBalance == oldBalance+price) then
          tmp := this.send(buyer,anItem)
      else
         tmp := buyer.tell("you have not paid me") ;
      res := 0

      private method send(buyer:external, anItem:Item) : int
       ...
  class Account
    field blnce : int
    field key : Key

    public method transfer(dest:Account, key':Key, amt:int) :int
      if (this.key==key') then
        this.blnce-=amt;
        dest.blnce+=amt
      else
        ϵ;
      res := 0

     public method set(key':Key) : int
      if (this.key==null)  then
            this.key:=key'
      else
      ϵ;
      res := 0
```

H.3 Demonstrating that $M_{good} \vdash S_2$

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. Our approach follows the 3 phases outlined in Section 8. That is, in phase 1 we use more an assumed underlying Hoare logic and more traditional Hoare triples to prove the adherence of internal code to the specification. In phase 2 we use Hoare quadruples to prove external calls adhere to the specification, and finally

in phase 3 we use raise the results from phase 1 and 2 to proved the entire module satisfies the specification.

To write our proofs more succinctly, we will use `ClassId::Methodid.body` as a shorthand for the method body of `Methodid` defined in `ClassId`.

H.3.1 Proof outline for $M_{good} \vdash S_2$.

Lemma H.2. $M_{good} \vdash S_2$

Proof Outline We construct our proof tree using a top down approach. That is, we start with our goal

$$M_{good} \vdash \forall \mathtt{a} : \mathtt{Account}.\{\langle\!\langle \mathtt{a.key} \rangle\!\rangle\}$$

and apply Invariant from Fig. 8. From this we are left with a subgoal for each method m in class C with parameters $\overline{y : D}$ in the public interface of M_{good} (we denote the body of such a method as `C::m.body`):

$$\begin{array}{l} M_{good} \vdash \\ \quad \{\, \mathtt{this} : \mathtt{C}, \overline{y : D}, \mathtt{a} : \mathtt{Account} \wedge \langle\!\langle \mathtt{a.key} \rangle\!\rangle \wedge \langle\!\langle \mathtt{a.key} \rangle\!\rangle \leftrightarrow\!\!\!\!\times\, \mathtt{this}, \overline{y} \,\} \\ \quad \mathtt{C} :: \mathtt{m.body} \\ \quad \{\, \langle\!\langle \mathtt{a.key} \rangle\!\rangle \wedge \langle\!\langle \mathtt{a.key} \rangle\!\rangle \leftrightarrow\!\!\!\!\times\, \mathtt{res} \,\} \;||\; \{\, \langle\!\langle \mathtt{a.key} \rangle\!\rangle \,\} \end{array}$$

Thus, we need to prove three Hoare quadruples:

We outline the proof for `buy` in Lemma H.3.

□

Lemma H.3.

$$\begin{array}{ll} (1) & M_{good} \vdash \{\, \mathtt{A}_{dcls} \wedge \langle\!\langle \mathtt{a.key} \rangle\!\rangle \wedge \langle\!\langle \mathtt{a.key} \rangle\!\rangle \leftrightarrow\!\!\!\!\times\, \mathtt{X}_{dcls} \,\} \\ & \qquad \mathtt{Shop} :: \mathtt{buy.body} \\ & \qquad \{\langle\!\langle \mathtt{a.key} \rangle\!\rangle \wedge \langle\!\langle \mathtt{a.key} \rangle\!\rangle \multimap\!\!\!\!\vee\, \mathtt{res}\} \;\;||\;\; \{\langle\!\langle \mathtt{a.key} \rangle\!\rangle\} \end{array}$$

where we are using the shorthands

$\mathtt{A}_{params} \triangleq \mathtt{this} : \mathtt{Shop}, \mathtt{buyer} : \mathtt{external}, \mathtt{anItem} : \mathtt{Item}, \mathtt{a} : \mathtt{Account}, \mathtt{price} : \mathtt{int}.$

$\mathtt{A}_{dcls} \triangleq \mathtt{A}_{params}, \mathtt{myAccnt} : \mathtt{Account}, \mathtt{oldBalance} : \mathtt{int}, \mathtt{newBalance} : \mathtt{int}, \mathtt{tmp} : \mathtt{int}.$

$\mathtt{X}_{dcls} \triangleq \mathtt{this}, \mathtt{buyer}, \mathtt{anItem}.$

Proof Outline

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`, ie that for a $z \notin \{\mathtt{price}, \mathtt{myAccnt}, \mathtt{oldBalance}\}$, we have

$$\begin{array}{ll} (10) & M_{good} \vdash_{ul} \{\, \mathtt{A}_{dcls} \wedge z = \mathtt{a.key}\} \\ & \qquad \mathtt{price := anItem.price;} \\ & \qquad \mathtt{myAccnt := this.accnt;} \\ & \qquad \mathtt{oldBalance := myAccnt.blnce;} \\ & \{z = \mathtt{a.key}\} \end{array}$$

We then apply Embed_UL, Prot-1 and Prot-2 and Combine and and Types-2 on (10) and use the shorthand $\texttt{stmts}_{10,11,12}$ for the statements on lines 10, 11 and 12, and obtain:

$$(11)\quad M_{good} \vdash \{\, \mathrm{A}_{dcls} \wedge \langle \texttt{a.key} \rangle \wedge \langle \texttt{buyer} \rangle \not\leftrightarrow \texttt{a.key} \}\ \texttt{stmts}_{10,11,12}\ \{\, \langle \texttt{a.key} \rangle \wedge \langle \texttt{buyer} \rangle \not\leftrightarrow \texttt{a.key} \}$$

We apply Mid on (11) and obtain

$$(12)\quad M_{good} \vdash \{\, \mathrm{A}_{dcls} \wedge \langle \texttt{a.key} \rangle \not\leftrightarrow \texttt{buyer} \}\ \texttt{stmts}_{10,11,12}\ \{\, \mathrm{A}_{params} \wedge \langle \texttt{a.key} \rangle \wedge \langle \texttt{buyer} \rangle \not\leftrightarrow \texttt{a.key} \} \;||\; \{\, \langle \texttt{a.key} \rangle \}$$

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.

$$(20?)\quad M_{good} \vdash \{\, \mathrm{A}_{dcls} \wedge \langle \texttt{a.key} \rangle, \wedge \langle \texttt{a.key} \rangle \not\leftrightarrow \texttt{buyer} \}\ \texttt{tmp := buyer.pay(myAccnt, price)}\ \{\, \mathrm{A}_{params} \wedge \langle \texttt{a.key} \rangle \wedge \langle \texttt{buyer} \rangle \not\leftrightarrow \texttt{a.key} \} \;||\; \{\, \langle \texttt{a.key} \rangle \}$$

We use that $M \vdash \forall \texttt{a} : \texttt{Account}.\{\langle \texttt{a.key} \rangle\}$ and obtain

$$(21)\quad M_{good} \vdash \{\, \texttt{buyer} : \texttt{external}, \langle \texttt{a.key} \rangle \wedge \langle \texttt{a.key} \rangle \not\leftrightarrow (\texttt{buyer}, \texttt{myAccnt}, \texttt{price}) \}\ \texttt{tmp := buyer.pay(myAccnt, price)}\ \{\, \langle \texttt{a.key} \rangle \wedge \langle \texttt{a.key} \rangle \not\leftrightarrow (\texttt{buyer}, \texttt{myAccnt}, \texttt{price}) \} \;||\; \{\, \langle \texttt{a.key} \rangle \}$$

In order to obtain (20?) out of (21), we apply Prot-Intl and Prot-Int_1, which gives us

(23) $M_{good} \vdash \mathrm{A}_{dcls} \wedge \langle \texttt{a.key} \rangle \longrightarrow \langle \texttt{a.key} \rangle \not\leftrightarrow \texttt{myAccnt}$

(24) $M_{good} \vdash \mathrm{A}_{dcls} \wedge \langle \texttt{a.key} \rangle \longrightarrow \langle \texttt{a.key} \rangle \not\leftrightarrow \texttt{price}$

We apply Consequ on (23), (24) and (21) and obtain (20)!

□

I DISCUSSING WHY $M_{bad} \not\vdash S_2$