Reasoning about External Calls

ANONYMOUS AUTHOR(S)

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

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

This paper addresses the specification and verification of internal code that relies on encapsulation and object capabilities to limit the effects of external calls. We propose new assertions for access to capabilities, new specifications for limiting effects, and a Hoare logic to verify that a module satisfies its specification, even while making external calls. We illustrate the approach 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 are pervasive in today's open world software. External, untrusted, or unknown code calls our trusted internal code, that internal code calls out to other external code and external code can even call back into internal code — all within the same call chain. This paper addresses reasoning about *external calls* — when trusted internal code calls out to untrusted, unknown external code. This reasoning is hard because by "external code" we mean untrusted code where we don't have a specification, where we may not be able to get source code, or which may even have been written to attack and subvert the system.

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

```
1 module M_intl
2   method m1 ..
3     ...  trusted code ...
4   method m2(untrst:external)
5     ... trusted code ...
6      untrst.unkn(this)
7     ... trusted code ...
```

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

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

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

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

2025. 2475-1421/2025/1-ART $15.00
https://doi.org/

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

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

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

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

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

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

2 THE PROBLEM AND OUR APPROACH

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

[1]We use the notation $\overline{z}$ for a sequence of z, *i.e.* for $z_1, z_2, ...z_n$

be called by objects of the same module, while `public` methods may be called by *any* object with a reference to the method receiver, and with actual arguments of dynamic types that match the declared formal parameter types.[2]

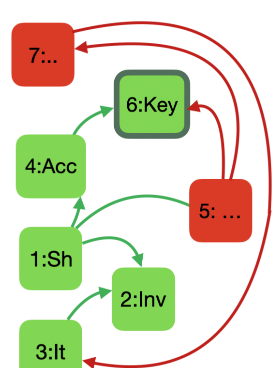
We are concerned with guarantees made in an *open* setting; Our internal module M must be programmed so that execution of M together with *any* unknown, arbitrary, external modules $\overline{M}$ will satisfy these guarantees – without relying on any assumptions about $\overline{M}$'s code.[3] All we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

`Shop` – illustrating limited effects

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

```
1  module Mshop
2    ...
3    class Shop
4      field 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)
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 sketch to the right shows a possible heap snippet. External objects are red; internal objects are green. Each object has a number, followed by an abbreviated class name: o_1, o_2 and o_5 are a `Shop`, an `Inventory`, and an external object. Curved arrows indicate field values: o_1 has three fields, pointing to o_4, o_5 and o_2. Fields denote direct access. The transitive closure of direct access gives indirect (transitive) access: o_1 has direct access to o_4, and indirect access to o_6. Object o_6 — highlighted with a dark outline — is the key capability that allows withdrawal from o_4.



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

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

[2] 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 [80].
[3] This is a critical distinction from e.g. cooperative approaches such as rely/guarantee [52, 114].
[4] Thus our approach is also applicable to intra-language safety.

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

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

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

2.1 1^{st} Challenge: Specification Language

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

(*) E (*e.g.* the account's balance decreases) can be caused only by external objects calling methods on internal objects,
and only if the causing object has access to o_c (*e.g.* the key).

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

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

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

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

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

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

Thus:

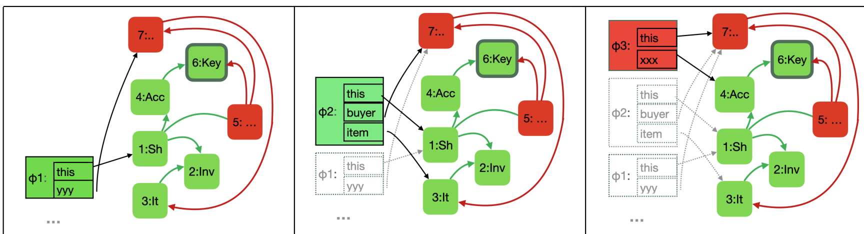
(***) If A holds, and no external object *reachable from the current point of execution* has access to o_c,
and no internal objects pass o_c to external objects,
then A holds in *the future scoped by the current point of execution.*

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

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

Fig. 1 illustrates the current point of execution. The left pane, σ_1, shows a state with the same heap as earlier, and where the top frame is ϕ_1 – it could be the state before a call to `buy`. The middle pane, σ_2, is a state where we have pushed ϕ_2 on top of the stack of σ_1 – it could be a state during execution of `buy`. The right pane, σ_3, is a state where we have pushed ϕ_3 on top of the stack of σ_2 – it could be a state during execution of `pay`.

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



σ_1, top frame ϕ_1	σ_2, top frame ϕ_2	σ_3, top frame ϕ_3

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

2.1.1 *Protection.*

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

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

heap	σ_1	σ_2	σ_3
... $\models \langle o_6 \rangle \not\leftrightarrow o_4$	$\sigma_1 \not\models \langle o_6 \rangle$	$\sigma_2 \not\models \langle o_6 \rangle$	$\sigma_3 \models \langle o_6 \rangle$
... $\not\models \langle o_6 \rangle \not\leftrightarrow o_5$	$\sigma_1 \models \langle o_4 \rangle$	$\sigma_2 \models \langle o_4 \rangle$	$\sigma_3 \not\models \langle o_4 \rangle$

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

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

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

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

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

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

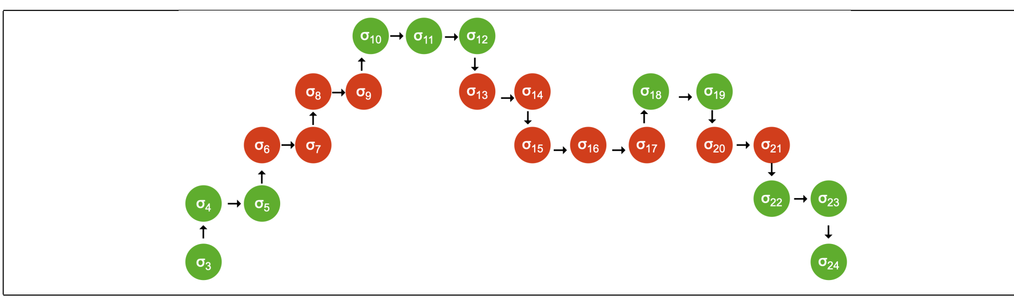


Fig. 3. *Execution.* Green disks represent internal states; red disks external states.

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

Example 2.1. The following scoped invariants

$S_1 \triangleq \forall \texttt{a}:\texttt{Account}.\{\langle\!\langle\texttt{a}\rangle\!\rangle\}$ $\qquad S_2 \triangleq \forall \texttt{a}:\texttt{Account}.\{\langle\!\langle\texttt{a.key}\rangle\!\rangle\}$

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

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

This example illustrates three crucial properties of our invariants:

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

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

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

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

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

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

$$\begin{array}{lll} S_4 & \triangleq & \{\, \langle\!\langle \texttt{this.accnt.key} \rangle\!\rangle \not\leftrightarrow \texttt{buyer} \wedge \texttt{this.accnt.blnce} = b \,\} \\ & & \quad\quad \texttt{public Shop} :: \texttt{buy}(\texttt{buyer} : \texttt{external}, \texttt{anItem} : \texttt{Item}) \\ & & \{\, \texttt{this.accnt.blnce} \geq b \,\} \;\|\; \{...\} \end{array}$$

S_4 guarantees that if the key was protected from `buyer` before the call, then the balance will not decrease[7]. It does *not* guarantee `buy` will only be called when $\langle\!\langle \texttt{this.accnt.key} \rangle\!\rangle \not\leftrightarrow \texttt{buyer}$ 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 ($\texttt{M}_{good}$, $\texttt{M}_{bad}$, and $\texttt{M}_{fine}$) of the M_{shop} module [74]; these all share the same `transfer` method to withdraw money:

```
module Mgood
  class Shop   ... as earlier ...
  class Account
    field blnce:int
    field key:Key
    public method transfer(dest:Account, key':Key, amt:nat)
       if (this.key==key')   this.blnce-=amt; dest.blnce+=amt
     public method set(key':Key)
       if (this.key==null)  this.key=key'
```

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

```
Mbad
public method set(key':Key)
  this.key=key'
```

```
Mfine
public method set(key',key'':Key)
  if (this.key==key')  this.key=key''
```

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

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

is enough to drain `acc` in $\texttt{M}_{bad}$ 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.

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

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

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

$$\mathbb{V}\overline{x : C}.\{A\}$$

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

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

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

$$\{\,A\,\}\ \ stmt\,\{\,A'\,\}\ \ \|\ \ \{\,A''\,\}$$

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

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

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

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

$$\{\ A_0 \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \wedge \texttt{a.blnce} \geq \texttt{b}\ \}$$
$$stmts_b$$
$$\{\ \langle\!\langle \texttt{a.key} \rangle\!\rangle \wedge \texttt{a.blnce} \geq \texttt{b}\ \}\ \|\ \{\ \langle\!\langle \texttt{a.key} \rangle\!\rangle \wedge \texttt{a.blnce} \geq \texttt{b}\ \}$$

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

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

$$\{\ \texttt{buyer:extl}\ \wedge\ \langle\!\langle \texttt{this.accnt.key} \rangle\!\rangle \leftrightarrow\!\!\!\!\times\ \texttt{buyer}\ \wedge\ \texttt{this.accnt.blnce} = b\ \}$$

(1) `buyer.pay(this.accnt,price)`

$$\{\ \texttt{this.accnt.blnce} \geq b\ \}\ \|\ \{\ ...\ \}$$

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

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

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

In general, if $\langle\!\langle \texttt{x} \rangle\!\rangle \leftrightarrow\!\!\!\!\times\ \texttt{y}_i$ holds for all $\texttt{y}_i$, before a call $\texttt{y}_0.m(\texttt{y}_1, ..., \texttt{y}_n)$, then $\langle\!\langle \texttt{x} \rangle\!\rangle$ holds upon entering the call. Here we have $\langle\!\langle \texttt{this.accnt.key} \rangle\!\rangle \leftrightarrow\!\!\!\!\times\ \texttt{buyer}$ by precondition. We also have that `price` is a scalar and therefore $\langle\!\langle \texttt{this.accnt.key} \rangle\!\rangle \leftrightarrow\!\!\!\!\times\ \texttt{price}$. And the type information gives that all fields transitively accessible from an `Account` are scalar or internal; this gives that $\langle\!\langle \texttt{this.accnt.key} \rangle\!\rangle \leftrightarrow\!\!\!\!\times\ \texttt{this.accnt}$. This enables the application of S_3 in **(1)**. The corresponding Hoare logic rule is shown in Fig. 8.

Summary

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

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

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

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

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

Fig. 4 shows the $\mathscr{L}_{ul}$ syntax. Statements, $stmt$, are three-address instructions, method calls, or empty, ϵ. Expressions, e, are ghost code; as such, they may appear in assertions but not in statements, and have no side-effects [22, 45]. Expressions may contain fields, $e.f$, or ghost-fields, $e_0.gf(\overline{e})$. The meaning of e is module-dependent; *e.g.* `a.blnce` is a field lookup in $\texttt{M}_{good}$, but in a module which stores balances in a table it would be a recursive lookup through that table – *c.f.* example in §A.3. [8] In line with most tools, we support ghost-fields, but they are not central to our work.

$\mathscr{L}_{ul}$ states, σ, consist of a heap χ and a stack. A stack is a sequence of frames, $\phi_1 \cdot ... \cdot \phi_n$. A frame, ϕ, consists of a local variable map and a continuation, *i.e.*the statements to be executed. The top frame, *i.e.* the frame most recently pushed onto the stack, in a state $(\phi_1 \cdot ... \cdot \phi_n, \chi)$ 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; $dom(\phi)$ and $Rng(\phi)$ indicate the variable map in ϕ; $dom(\sigma)$ and $Rng(\sigma)$ indicate the variable map in the top frame in σ
- $\alpha \in \sigma$ means that α is defined in the heap of σ, and $x \in \sigma$ means that $x \in dom(\sigma)$. Conversely, $\alpha \notin \sigma$ and $x \notin \sigma$ have the obvious meanings. $\lfloor \alpha \rfloor_\sigma$ 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 $\lfloor e \rfloor_\sigma$.

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

$$
\begin{array}{lcll lcll}
Mdl & ::= & \overline{C \mapsto CDef} & \textbf{Module Def.} & fld & ::= & \texttt{field}\, f\, :\, T & \textbf{Field Def.}\\
CDef & ::= & \texttt{class}\, C\, \{\, \overline{fld};\, \overline{mth};\, \overline{gfld};\, \} & \textbf{Class Def.} & T & ::= & C & \textbf{Type}\\
mth & ::= & p\, \texttt{method}\, m\, (\overline{x : T}) \colon T \{\, s\, \} & \textbf{Method Def.} & p & ::= & \texttt{private} \mid \texttt{public} & \textbf{Privacy}
\end{array}
$$

$$
\begin{array}{lcll}
stmt & ::= & x := y \mid x := v \mid x := y.f \mid x.f := y \mid x := y_0.m(\overline{y}) \mid \texttt{new}\, C \mid stmt;\, stmt \mid \epsilon & \textbf{Statement}\\
gfld & ::= & \texttt{ghost}\, gf(\overline{x : T})\{\, e\, \} : T & \textbf{Ghost Field Def.}\\
e & ::= & x \mid v \mid e.f \mid e.gf(\overline{e}) & \textbf{Expression}
\end{array}
$$

$$
\begin{array}{lcll lcll}
\sigma & ::= & (\overline{\phi}, \chi) & \textbf{Program State} & C, f, m, gf, x, y & ::= & Identifier & \\
\phi & ::= & (\, \overline{x \mapsto v};\, s\,) & \textbf{Frame} & o & ::= & (\, C;\, \overline{f \mapsto v}\,) & \textbf{Object}\\
\chi & ::= & (\, \overline{\alpha \mapsto o}\,) & \textbf{Heap} & v & ::= & \alpha \mid \texttt{null} & \textbf{Value}
\end{array}
$$

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

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

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

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

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

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

[9] More fine-grained privacy, *e.g.* C++ private fields or ownership types, would provide all the guarantees needed in our work.
[10] For ease of presentation, we omit assignment to `res` in methods returning `void`.

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

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

$$\frac{\sigma.\texttt{cont} \overset{\text{txt}}{=} x := \texttt{new}\ C; s \qquad x \notin Prms(\sigma, \overline{M}) \qquad fields(\overline{M}, C) = \overline{f} \qquad \alpha \text{ fresh in } \sigma}{\overline{M}, \sigma \dashrightarrow \sigma[x \mapsto \alpha][\alpha \mapsto (\ C;\ \overline{f \ \mapsto \texttt{null}}\ )][\texttt{cont} \mapsto s]} \quad \text{(New)}$$

$$\frac{\begin{array}{c}\phi_n.\texttt{cont} \overset{\text{txt}}{=} u := y_0.m(\overline{y});\_ \qquad u \notin Prms((\overline{\phi} \cdot \phi_n, \chi), \overline{M}) \\ Meth(\overline{M}, classOf((\phi_n, \chi), y_0), m) = p\ C::m(\overline{x : T}) : T\{\ stmt\ \} \quad p = \texttt{public} \vee SameModule(\texttt{this}, y_0, (\phi_n, \chi), \overline{M})\end{array}}{\overline{M}, (\overline{\phi} \cdot \phi_n, \chi) \dashrightarrow (\overline{\phi} \cdot \phi_n \cdot (\ \texttt{this} \mapsto \lfloor y_0 \rfloor_{\phi_n}, \overline{x \mapsto \lfloor y \rfloor_{\phi_n}};\ stmt\ ), \chi)} \quad \text{(Call)}$$

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

Fig. 5. $\mathscr{L}_{ul}$ operational Semantics

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

3.3 Fundamental Concepts

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

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

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

- $\sigma \triangledown \phi \ \triangleq\ (\overline{\phi} \cdot \phi, \chi)$ 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. 3 again: $\sigma_8 = \sigma_7 \triangledown \phi$ for some ϕ, and $\sigma_{15} = \sigma_{14} \vartriangle$.

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

Definition 3.2 (Scoped Execution). Given modules $\overline{M}$, and states σ, σ_1, σ_n, and σ', we define:

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

Consider Fig. 3 : 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 $\overline{M};\ \sigma_8 \not\rightsquigarrow^* \sigma_{18}$: 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 $\overline{M};\ \sigma_9 \rightsquigarrow \sigma_{12}$ and $\overline{M};\ \sigma_{12} \not\rightsquigarrow^* \sigma_{13}$. Lemma 3.3 says that the value of the parameters does not change during execution of the same method. Appendix B discusses proofs, and further properties.

Lemma 3.3. For all $\overline{M}, \sigma, \sigma'$: $\quad \overline{M};\ \sigma \rightsquigarrow^* \sigma' \ \wedge\ |\sigma| = |\sigma'| \quad \Longrightarrow \quad \forall x \in Prms(\overline{M}, \sigma).[\ \lfloor x \rfloor_\sigma = \lfloor x \rfloor_{\sigma'}\ ]$

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

Definition 3.4. We define

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

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

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

$$\begin{array}{l} \overline{M} \models \sigma \quad \triangleq \quad \forall k \in \mathbb{N}.[\ 1 < k \leq |\sigma| \Longrightarrow \\ \qquad\qquad [\ \forall x \in Prms(\sigma[k], \overline{M}).[\ \exists y.\ \lfloor x \rfloor_{\sigma[k]} = \lfloor y \rfloor_{\sigma[k-1]}\ ] \quad \wedge \\ \qquad\qquad\ LocRchbl(\sigma[k]) = \bigcup_{z \in Prms(\sigma[k], \overline{M})} Rchbl(\lfloor z \rfloor_{\sigma[k]}, \sigma) \quad] \end{array}$$

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

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

(1) $\overline{M} \models \sigma \ \wedge\ \overline{M}, \sigma \dashrightarrow \sigma' \ \Longrightarrow\ \overline{M} \models \sigma'$
(2) $\sigma' = \sigma \triangledown \phi \ \wedge\ \overline{M} \models \sigma' \ \Longrightarrow\ LocRchbl(\sigma') \subseteq LocRchbl(\sigma)$

4 ASSERTIONS

Our assertions are standard (*e.g.* properties of the values of expressions, connectives, quantification *etc.*) or about protection (*i.e.* $\langle\!\langle e \rangle\!\rangle \not\leftrightarrow e$ and $\langle\!\langle e \rangle\!\rangle$).

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

$$A \quad ::= \quad e \mid e : C \mid \neg A \mid A \wedge A \mid \forall x : C.A \mid e : \texttt{extl} \mid \langle\!\langle e \rangle\!\rangle \not\leftrightarrow e \mid \langle\!\langle e \rangle\!\rangle \qquad [11]$$

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

Definition 4.2 (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 such as $A \rightarrow A'$, $A \vee A'$, and $\exists x : C.A$ can be encoded.

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

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

4.1 Semantics of assertions – first part

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

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

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

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

4.2 Semantics of Assertions - second part

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

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

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

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

(1) $M, \sigma \models \langle\!\langle \alpha \rangle\!\rangle \not\leftrightarrow \alpha_o \quad \triangleq$
  (a) $\alpha \neq \alpha_0$,
  (b) $\forall \alpha'.\forall f.[\ \alpha' \in \mathit{Rchbl}(\alpha_o, \sigma) \wedge M, \sigma \models \alpha' : \texttt{extl} \implies \lfloor \alpha'.f \rfloor_\sigma \neq \alpha\ ]$.
(2) $M, \sigma \models \langle\!\langle \alpha \rangle\!\rangle \quad \triangleq$
  (a) $M, \sigma \models \texttt{extl} \implies \forall x \in \sigma.\ M, \sigma \models x \neq \alpha$,
  (b) $\forall \alpha'.\forall f.[\ \alpha' \in \mathit{LocRchbl}(\sigma) \wedge M, \sigma \models \alpha' : \texttt{extl} \implies \lfloor \alpha'.f \rfloor_\sigma \neq \alpha\ ]$.

Moreover,

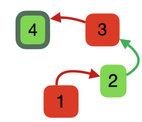
(3) $M, \sigma \models \langle\!\langle e \rangle\!\rangle \not\leftrightarrow e_o \triangleq \exists \alpha, \alpha_o.[\ M, \sigma, e \hookrightarrow \alpha \ \wedge \ M, \sigma, e_0 \hookrightarrow \alpha_0 \ \wedge \ M, \sigma \models \langle\!\langle \alpha \rangle\!\rangle \not\leftrightarrow \alpha_o\ ]$,
(4) $M, \sigma \models \langle\!\langle e \rangle\!\rangle \triangleq \exists \alpha.[\ M, \sigma, e \hookrightarrow \alpha \ \wedge \ M, \sigma \models \langle\!\langle \alpha \rangle\!\rangle\ ]$.

We illustrate "protected" and "protected from" in Fig. 2 in §2. and Fig. 13 in App. C. In general, $\langle\!\langle \alpha \rangle\!\rangle \not\leftrightarrow \alpha_o$ ensures that α_o will get access to α only if another object grants that access. Similarly, $\langle\!\langle \alpha \rangle\!\rangle$ ensures that during execution of the current method, no external object will get direct access to α unless some internal object grants that access[12]. Thus, protection together with protection preservation (*i.e.* no internal object gives access) guarantee lack of eventual external access.

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

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

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



4.3 Preservation of Assertions

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

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

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

- ${}^{\sigma}\lceil A \rceil \;\triangleq\; A[\overline{\lfloor x \rfloor_{\sigma}/x}]$
- $M, \sigma \models A \quad\Longleftrightarrow\quad M, \sigma \models {}^{\sigma}\lceil A \rceil \quad\Longleftrightarrow\quad M, \sigma[\texttt{cont} \mapsto stmt] \models A$

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

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

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

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

- $Stbl(A) \quad\Longrightarrow\quad [\ M, \sigma \models A \quad\Longleftrightarrow\quad M, \sigma \triangledown \phi \models A\ ]$
- $Stb^{+}(A) \,\wedge\, M \cdot \overline{M} \models \sigma \triangledown \phi \,\wedge\, M, \sigma \triangledown \phi \models \texttt{intl} \,\wedge\, M, \sigma \models A \quad\Longrightarrow\quad M, \sigma \triangledown \phi \models A$

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

Example 4.7. Fig. 2 illustrates that
– *Stb^{+} not necessarily preserved by External Push*: Namely, $\sigma_2 \models \langle\!\langle o_4 \rangle\!\rangle$, pushing frame ϕ_3 with an external receiver and o_4 as argument gives σ_3, we have $\sigma_3 \not\models \langle\!\langle o_4 \rangle\!\rangle$.
– *Stb^{+} not necessarily preserved by Pop*: Namely, $\sigma_3 \models \langle\!\langle o_6 \rangle\!\rangle$, returning from σ_3 would give σ_2, and we have $\sigma_2 \not\models \langle\!\langle o_6 \rangle\!\rangle$.

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 6.2.2), and the latter through *deep satisfaction* (§7).

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

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

	$\texttt{z.f} \geq 3$	$\langle\!\langle \texttt{x} \rangle\!\rangle$	$\neg(\langle\!\langle \texttt{x} \rangle\!\rangle)$	$\langle\!\langle \texttt{y} \rangle\!\rangle \not\leftrightarrow \texttt{x}$	$\neg(\langle\!\langle \texttt{y} \rangle\!\rangle \not\leftrightarrow \texttt{x})$
$Stbl(\_)$	✓	×	×	✓	✓
$Stb^+(\_)$	✓	✓	×	✓	✓
$Enc(\_)$	✓	✓	×	×	×

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

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

- $M \vdash Enc(A) \implies \forall \overline{M}, \sigma, \sigma'.[\ M, \sigma \models (A \wedge \texttt{extl}) \wedge M \cdot \overline{M};\ \sigma \rightsquigarrow \sigma' \implies M, \sigma' \models {}^{\sigma}\lceil A \rceil\ ]$

5 SPECIFICATIONS

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

5.1 Syntax, Semantics, Examples

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

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

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

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

- $\overline{M}; M \models \{\, A \,\}\, \sigma\, \{\, A' \,\} \parallel \{\, A'' \,\} \ \triangleq\ \forall \sigma', \sigma''.[$
  $M, \sigma \models A \implies [\ \overline{M} \cdot M;\ \sigma \rightsquigarrow^*_{fin} \sigma' \implies M, \sigma' \models A'\ ]\ \wedge$
  $[\ \overline{M} \cdot M;\ \sigma \rightsquigarrow^* \sigma'' \implies M, \sigma'' \models (\texttt{extl} \rightarrow {}^{\sigma}\lceil A'' \rceil)\ ]\ ]$

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

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

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

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

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

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

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

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

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

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

5.2 Well-formedness

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

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

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

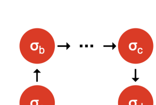
Def 5.6's requirements about free variables are relatively straightforward – more in. §E.1.1.

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

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

5.3 Discussion

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



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

History invariants [27, 67, 69], instead, consider all future states including any method returns, and therefore would require that $\sigma_d \models A$. Thus, they are, for our purposes, both *unenforceable* and overly *restrictive. Unenforceable*: Take $A \stackrel{\text{txt}}{=} \langle\!\langle$`acc.key`$\rangle\!\rangle$, assume in σ_a a path to an external object which has access to `acc.key`, assume that path is unknown in σ_b: then, the transition from σ_b to σ_c cannot eliminate that path—hence, $\sigma_d \not\models \langle\!\langle$`acc.key`$\rangle\!\rangle$. *Restrictive*: Take $A \stackrel{\text{txt}}{=} \langle\!\langle$`acc.key`$\rangle\!\rangle \wedge a$.`blnce` $\geq b$; then, requiring A to hold in all states from σ_a until termination would prevent all future withdrawals from a, rendering the account useless.

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

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

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

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

$$
\begin{aligned}
S_{\texttt{tempLeak}} \;\triangleq\;\; & \{\ \texttt{true}\ \} \\
& \quad \texttt{public Account} :: \texttt{tempLeak}(\texttt{extArg} : \texttt{external}) \\
& \{\ \langle\!\langle \texttt{this.key} \rangle\!\rangle\ \}\ \parallel\ \{\ \texttt{true}\ \}
\end{aligned}
$$

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

6 HOARE LOGIC

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

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

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

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

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

6.1 First Phase: Triples

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

$$\frac{\begin{array}{c} stmt \text{ contains no method call} \\ Stbl(A) \quad Stbl(A') \qquad M \vdash_{ul} \{\, A \,\}\ stmt\ \{\, A' \,\} \end{array}}{M \ \vdash\ \{\, A \,\}\ stmt\ \{\, A' \,\}} \ \text{[EMBED\_UL]}$$

$$\frac{u \overset{\text{txt}}{\neq} x}{M \ \vdash\ \{\, true \,\}\ u = \texttt{new}\, C\ \{\, \langle\!\langle u \rangle\!\rangle \ \wedge\ \langle\!\langle u \rangle\!\rangle \leftrightarrow\!\!\!\!\times\ x \,\}} \ \text{[PROT-NEW]}$$

$$\frac{\begin{array}{c} stmt \text{ is free of method cals, or assignment to } z \\ M \ \vdash\ \{\, A \wedge e{=}z \,\}\ stmt\ \{\, e{=}z \,\} \end{array}}{M \ \vdash\ \{\ A \wedge \langle\!\langle e \rangle\!\rangle\ \}\ stmt\ \{\ \langle\!\langle e \rangle\!\rangle\ \}} \ \text{[PROT-1]}$$

$$\frac{\begin{array}{c} stmt \text{ is either } x := y \text{ or } x := y.f \qquad z, z' \overset{\text{txt}}{\neq} x \\ M \ \vdash\ \{\, A \wedge z{=}e \wedge z'{=}e' \,\}\ stmt \{\, z{=}e \wedge z'{=}e' \,\} \end{array}}{M \ \vdash\ \{\ A \wedge \langle\!\langle e \rangle\!\rangle \leftrightarrow\!\!\!\!\times\ e'\ \}\ stmt \{\ \langle\!\langle e \rangle\!\rangle \leftrightarrow\!\!\!\!\times\ e'\ \}} \ \text{[PROT-2]}$$

$$\frac{x \overset{\text{txt}}{\neq} z}{M \ \vdash\ \{\ \langle\!\langle y.f \rangle\!\rangle \leftrightarrow\!\!\!\!\times\ z\ \}\ x = y.f\ \{\ \langle\!\langle x \rangle\!\rangle \leftrightarrow\!\!\!\!\times\ z\ \}} \ \text{[PROT-3]}$$

$$\frac{}{M \ \vdash\ \{\ \langle\!\langle x \rangle\!\rangle \leftrightarrow\!\!\!\!\times\ z \ \wedge\ \langle\!\langle x \rangle\!\rangle \leftrightarrow\!\!\!\!\times\ y' \,\}\ y.f = y'\ \{\ \langle\!\langle x \rangle\!\rangle \leftrightarrow\!\!\!\!\times\ z\ \}} \ \text{[PROT-4]}$$

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

With rule EMBED_UL in Fig. 7, any triple $\{A\}stmt\{A'\}$ whose statement does not contain a method call, and which can be proven in the underlying Hoare logic, can also be proven in our logic. In PROT-1, we see that protection of an object o is preserved by internal code which does not call any methods: namely any heap modifications will 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 soundness for these rules can be found in App. G.5.1.

6.2 Second Phase: Quadruples

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

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

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

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

Definition 6.1. [The $-\!\nabla$ operator]

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

Only the first equation in Def. 6.1 is interesting: for e to be protected at a callee with arguments $\overline{y}$, it should be protected from these arguments – thus $\langle e \rangle -\!\!\triangledown \overline{y} = \langle e \rangle \not\leftrightarrow \overline{y}$. The notation $\langle e \rangle \not\leftrightarrow \overline{y}$ stands for $\langle e \rangle \not\leftrightarrow y_0 \ \wedge \ \ldots \wedge \langle e \rangle \not\leftrightarrow y_n$, assuming that $\overline{y}$=$y_0, \ldots y_n$.

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

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

(1) $Stbl(\, A -\!\!\triangledown (y_0, \overline{y})\,)$
(2) $M, \sigma \models A -\!\!\triangledown Rng(\phi) \quad \Longrightarrow \quad M, \sigma \triangledown \phi \models A$
(3) $M, \sigma \triangledown \phi \models A \wedge \texttt{extl} \quad \Longrightarrow \quad M, \sigma \models A -\!\!\triangledown Rng(\phi)$
(4) $M, \sigma \models A \wedge \texttt{extl} \ \wedge \ M \cdot \overline{M} \models \sigma \triangledown \phi \quad \Longrightarrow \quad M, \sigma \triangledown \phi \models A -\!\!\triangledown Rng(\phi)$

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

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

6.2.3 Reasoning about calls. is described in Fig. 8. Call_Int for internal methods, whether public or private; and Call_Ext_Adapt and Call_Ext_Adapt_Strong for external methods.

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

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

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

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

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

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

Notice that internal calls, Call_Int, require the *un-adapted* method precondition (*i.e.* A_1'), while external calls, both Call_Ext_Adapt and Call_Ext_Adapt_Strong, require the *adapted* invariant (*i.e.* $A -\!\!\triangledown (y_0, \overline{y})$). This is sound, because internal callees preserve $Stb^+(\_)$-assertions – *c.f.* Lemma 4.6. On the other hand, external callees do not necessarily preserve $Stb^+(\_)$-assertions – *c.f.* Ex.

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

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

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

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

$$\{\ \texttt{buyer}:\texttt{extl} \wedge \langle\!\langle \texttt{this.accnt.key} \rangle\!\rangle \not\leftrightarrow \texttt{buyer} \wedge \texttt{this.accnt.blnce} = b\ \}$$

(1?) $\qquad \texttt{buyer.pay(this.accnt,price)}$

$$\{\ \texttt{this.accnt.blnce} \geq b\ \}\ \ ||\ \ \{\ \langle\!\langle \texttt{a.key} \rangle\!\rangle \wedge \texttt{a.blnce} \geq \texttt{b}\ \}$$

We use S_3, which says that $\mathbb{V}\texttt{a}:\texttt{Account},\texttt{b}:\texttt{int}.\{\langle\!\langle \texttt{a.key} \rangle\!\rangle \wedge \texttt{a.blnce} \geq \texttt{b}\}$. We can apply rule Call_Ext_Adapt, by taking $y_0 \triangleq \texttt{buyer}$, and $\overline{x:D} \triangleq \texttt{a}:\texttt{Account},\texttt{b}:\texttt{int}$, and $A \triangleq \langle\!\langle \texttt{a.key} \rangle\!\rangle \wedge \texttt{a.blnce} \geq \texttt{b}$, and $m \triangleq \texttt{pay}$, and $\overline{y} \triangleq \texttt{this.accnt},\texttt{price}$, and provided that we can establish that

(2?) $\langle\!\langle \texttt{this.accnt.key} \rangle\!\rangle \not\leftrightarrow (\texttt{buyer},\texttt{this.accnt},\texttt{price})$

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

(3) $\langle\!\langle \texttt{this.accnt.key} \rangle\!\rangle \not\leftrightarrow \texttt{this.accnt} \wedge \langle\!\langle \texttt{this.accnt.key} \rangle\!\rangle \not\leftrightarrow \texttt{price}$

Using then Def. 6.1, we can indeed establish that

(4) $\langle\!\langle \texttt{this.accnt.key} \rangle\!\rangle \not\leftrightarrow (\texttt{buyer},\texttt{this.accnt},\texttt{price})\ =\ \langle\!\langle \texttt{this.accnt.key} \rangle\!\rangle \not\leftrightarrow \texttt{buyer}$

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

6.3 Third phase: Proving adherence to Module Specifications

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

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

$$\textsc{method}\ \frac{\begin{array}{c}\texttt{mBody}(m,D,M) = p\ (\overline{y:D})\{\ stmt\ \}\\ M\ \vdash\ \{\ \texttt{this}:\texttt{D}, \overline{y:D} \wedge A_1\ \}\ stmt\ \{\ A_2 \wedge A_2 \mathbin{\text{-}\!\nabla} \texttt{res}\ \}\ \ ||\ \ \{A_3\}\end{array}}{M \vdash \{A_1\}\ p\ D::m(\overline{y:D})\ \{A_2\}\ \ ||\ \ \{A_3\}}$$

$$\textsc{invariant}\ \frac{\begin{array}{l}\forall D,m: \quad \texttt{mBody}(m,D,M) = \underline{\texttt{public}}\ (\overline{y:D})\{\ stmt\ \} \implies\\ \quad M\ \vdash\ \{\ \texttt{this}:\texttt{D}, \overline{y:D}, \overline{x:C}\ \wedge\ A \wedge A \mathbin{\text{-}\!\nabla} (\texttt{this},\overline{y})\ \}\ stmt\ \{\ A\ \wedge\ A \mathbin{\text{-}\!\nabla} \texttt{res}\ \}\ \ ||\ \ \{\ A\ \}\end{array}}{M \vdash \mathbb{V}\overline{x:C}.\{A\}}$$

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

WellFrm_Mod and Comb_Spec say that M is well-formed if its specification is well-formed (according to Def. 5.6), and if M satisfies all conjuncts of the specification. Method says that a module satisfies a method specification if the body satisfies the corresponding pre-, post- and midcondition. In the postcondition we also ask that A-∇`res`, so that `res` does not leak any of the values that A promises will be protected. INVARIANT says that a module satisfies a specification $\forall \overline{x:C}.\{A\}$, if the method body of each public method has A as its pre-, post- and midcondition. Moreover, the precondition is strengthened by A-∇(`this`, $\overline{y}$) – this is sound because the caller is external, and by Lemma 6.2, part (4).

Barendregt In METHOD we implicitly require the free variables in a method's precondition not to overlap with variables in its body, unless they are the receiver or one of the parameters ($Vs(stmt) \cap Fv(A_1) \subseteq \{\texttt{this}, y_1, ...y_n\}$). 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.

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

```
      { ... a:Account ∧ ⟪a.key⟫ ∧ ⟪a.key⟫↚̸(key′,key″) }
(5?)        body_of_set_in_Account_in_Mfine
      { ⟪a.key⟫ ∧ ⟪a.key⟫-∇res } || { ⟪a.key⟫ }
```

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

```
      { ... ⟪a.key⟫ ∧ ⟪a.key⟫↚̸(key′,key″) ∧ this.key = key′ }
(6?)        this.key := key″
      { ⟪a.key⟫ } || { ⟪a.key⟫ }
```

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

```
      { ... ⟪a.key⟫ ∧ ⟪a.key⟫↚̸(key′,key″) ∧ this.key=key′ ∧ this=a }
(7?)        this.key := key″
      { ⟪a.key⟫ } || { ⟪a.key⟫ }
```

and also

```
      { ...⟪a.key⟫ ∧ ⟪a.key⟫↚̸(key′,key″) ∧ this.key=key′ ∧ this≠a }
(8?)        this.key := key″
      { ⟪a.key⟫ } || { ⟪a.key⟫ }
```

If `this.key=key′` ∧ `this=a`, then `a.key=key′`. But ⟪`a.key`⟫↚̸`key′` and PROT-NEQ from Fig. 17 give `a.key`≠`key′`. So, by contradiction (*c.f.* Fig. 16), we can prove **(7?)**. If `this`≠`a`, then we obtain from the underlying Hoare logic that the value of `a.key` did not change. Thus, by rule PROT_1, we obtain **(8?)**. More details in §H.5.

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

```
      { ... ⟪a.key⟫ ∧ ⟪a.key⟫↚̸(key′,key″) }
(??)        this.key := key″
      { {⟪a.key⟫ } || { ⟪a.key⟫ }
```

and without the condition `this.key=key′` there is no way we can prove **(??)**.

6.4 Our Example Proven

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

Our proof models $\mathscr{L}_{ul}$, the assertion language, the specification language, and the Hoare logic from §6.1, §6.2, §6.3, §F and Def. 6.1. In keeping with the start of §6, our proof assumes the existence of an underlying Hoare logic, and several, standard, properties of that underlying logic, the assertions logic (*e.g.* equality of objects implies equality of field accesses) and of type systems (*e.g.* fields of objects of different types cannot be aliases of one another). All assumptions are clearly indicated in the associated artifact.

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

7 SOUNDNESS

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

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

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

Lemma 7.1. For all $\overline{M}$, M, A, A', A'', σ:

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

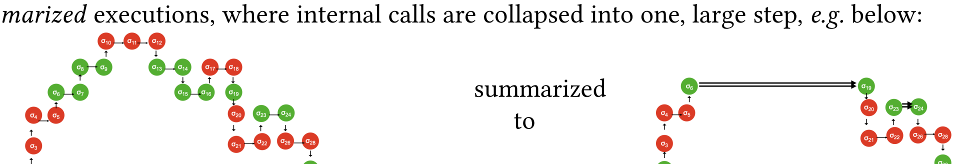
Soundness of the Triples Logic We require the assertion logic, $M \vdash A$, and the underlying Hoare logic, $M\ \vdash_{ul}\ \{\,A\,\}\,stmt\{\,A'\,\}$, to be be sound. Such sound logics do exist. Namely, one can build an assertion logic, $M \vdash A$, by extending a logic which does not talk about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

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

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

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

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



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

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

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

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

$(A): \ \vdash M \ \wedge \ M \vdash \{A\}\, stmt\{A'\} \ \parallel \ \{A''\} \quad \Longrightarrow \quad M \models_{deep} \{A\}\, stmt\{A'\} \ \parallel \ \{A''\}$

$(B): \ M \vdash S \quad \Longrightarrow \quad M \models_{deep} S$

8 CONCLUSION: SUMMARY, RELATED WORK AND FURTHER WORK

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Rely-Guarantee [52, 114] and Deny-Guarantee [39] distinguish between assertions guaranteed by a thread, and those a thread can reply upon. Our Hoare quadruples are (roughly) Hoare triples plus the "guarantee" portion of rely-guarantee. When a specification includes a guarantee, that guarantee must be maintained by every "atomic step" in an execution [52], rather than just at method boundaries as in visible states semantics [41, 91, 109]. In concurrent reasoning, this is because shared state may be accessed by another 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 [92] made the first attempt to formalise defensive consistency, to tolerate interacting with any untrustworthy object, although without a specification language for describing effects (i.e. when an object is correct).

Cassez et al. [21] propose one approach to reason about external calls. Given that external callbacks are necessarily restricted to the module's public interface, external callsites are replaced with a generated `externalcall()` method that nondeterministically invokes any method in that interface. Rao et al. [101]'s Iris-Wasm is similar. WASM's modules are very loosely coupled: a module has its own byte memory and object table. Iris-Wasm ensures models can only be modified via their explicitly exported interfaces.

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

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

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

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

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

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

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

DATA AVAILABILITY STATEMENT

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

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

REFERENCES

[1] Gul Agha and Carl Hewitt. 1987. Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming. In *Research Directions in Object-Oriented Programming*, Bruce D. Shriver and Peter Wegner (Eds.). MIT Press, 49–74.

[2] Elvira Albert, Shelly Grossman, Noam Rinetzky, Clara Rodríguez-Núñez, Albert Rubio, and Mooly Sagiv. 2023. Relaxed Effective Callback Freedom: A Parametric Correctness Condition for Sequential Modules With Callbacks. *IEEE Trans. Dependable Secur. Comput.* 20, 3 (2023), 2256–2273. https://doi.org/10.1109/TDSC.2022.3178836

[3] Sean Noble Anderson, Roberto Blanco, Leonidas Lampropoulos, Benjamin C. Pierce, and Andrew Tolmach. 2023. Formalizing Stack Safety as a Security Property. In *Computer Security Foundations Symposium*. 356–371. https://doi.org/10.1109/CSF57540.2023.00037

[4] Cezar-Constantin Andrici, 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. https://doi.org/10.1145/3632916

[5] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust types for modular specification and verification. *OOPSLA* 3 (2019), 147:1–147:30. https://doi.org/10.1145/3360573

[6] Anindya Banerjee and David A. Naumann. 2005. Ownership Confinement Ensures Representation Independence for Object-oriented Programs. *J. ACM* 52, 6 (Nov. 2005), 894–960. https://doi.org/10.1145/1101821.1101824

[7] Yuyan Bao, Guannan Wei, Oliver Bracevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. 2021. Reachability types: tracking aliasing and separation in higher-order functional programs. *OOPSLA* 5 (2021), 1–32. https://doi.org/10.1145/3485516

[8] Michael Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. 2004. Verification of Object-Oriented Programs with Invariants. *J. Object Technol.* 3, 6 (2004), 27–56. https://doi.org/10.5381/JOT.2004.3.6.A2

[9] Mike Barnett, Rustan Leino, and Wolfram Schulte. 2005. The Spec♯ Programming System: An Overview. In *CASSIS*, Vol. LNCS3362. 49–69. https://doi.org/10.1007/978-3-540-30569-9_3

[10] Jesper Bengtson, Kathiekeyan Bhargavan, Cedric Fournet, Andrew Gordon, and S.Maffeis. 2011. Refinement Types for Secure Implementations. *TOPLAS* (2011), 1– 45. https://doi.org/10.1145/1890028.1890031

[11] Andrew Black, Kim Bruce, Michael Homer, and James Noble. 2012. Grace: the Absence of (Inessential) Difficulty. In *Onwards*. 85–98. https://doi.org/10.1145/2384592.2384601

[12] Aleksander Boruch-Gruszecki, Martin Odersky, Edward Lee, Ondrej Lhoták, and Jonathan Immanuel Brachthäuser. 2023. Capturing Types. *TOPLAS* 45, 4 (2023), 21:1–21:52. https://doi.org/10.1145/3618003

[13] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. 2003. Ownership types for object encapsulation. In *POPL*. 213–223. https://doi.org/10.1145/604131.604156

[14] John Boyland. 2001. Alias burying: Unique variables without destructive reads. In *S:P&E*. 533–553. https://doi.org/10.1002/spe.370

[15] Gilad Bracha. 2015. *The Dart Programming Language*. Addison-Wesley. 224 pages. https://dart.dev

[16] Gilad Bracha. 2017. The Newspeak Language Specification Version 0.1. (Feb. 2017). https://newspeaklanguage.org/

[17] Jonathan Immanuel Brachthäuser, Philipp Schuster, Edward Lee, and Aleksander Boruch-Gruszecki. 2022. Effects, capabilities, and boxes: from scope-based reasoning to type-based reasoning and back. *OOPSLA* 6 (2022), 1–30. https://doi.org/10.1145/3527320

[18] Christian Bräm, Marco Eilers, Peter Müller, Robin Sierra, and Alexander J. Summers. 2021. Rich specifications for Ethereum smart contract verification. *OOPSLA* 5 (2021), 1–30. https://doi.org/10.1145/3485523

[19] Anton Burtsev, David Johnson, Josh Kunz, Eric Eide, and Jacobus E. van der Merwe. 2017. CapNet: security and least authority in a capability-enabled cloud. In *SoCC*. 128–141. https://doi.org/10.1145/3127479.3131209

[20] Nicholas Cameron, Sophia Drossopoulou, and James Noble. 2013. Understanding Ownership Types with Dependent Types. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. 84–108. https://doi.org/10.1007/978-3-642-36946-9_5

[21] Franck Cassez, Joanne Fuller, and Horacio Mijail Anton Quiles. 2024. Deductive verification of smart contracts with Dafny. *Int. J. Softw. Tools Technol. Transf.* 26, 2 (2024), 131–145. https://doi.org/10.1007/S10009-024-00738-1

[22] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. 2005. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In *FMCO*. 342–363. https://doi.org/10.1007/11804192_16

[23] Edwin C. Chan, John Boyland, and William L. Scherlis. 1998. Promises: Limited Specifications for Analysis and Manipulation. In *ICSE*. 167–176. https://doi.org/10.1109/ICSE.1998.671113

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

[25] David G. Clarke, John M. Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *OOPSLA*. 48– 64. https://doi.org/10.1145/286936.286947

[26] David G. Clarke, John M. Potter, and James Noble. 2001. Simple Ownership Types for Object Containment. In *ECOOP*. 53–76. https://doi.org/10.1007/3-540-45337-7_4

[27] Ernie Cohen, Michal Moskal, Wolfram Schulte, and Stephan Tobies. 2010. Local Verification of Global Invariants in Concurrent Programs. In *CAV*. 480–494. https://doi.org/10.1007/978-3-642-14295-6_42

[28] David R. Cok and K. Rustan M. Leino. 2022. *Specifying the Boundary Between Unverified and Verified Code.* Chapter 6, 105–128. https://doi.org/10.1007/978-3-031-08166-8_6

[29] Joseph W. Cutler, Craig Disselkoen, Aaron Eline, Shaobo He, Kyle Headley, Michael Hicks, Kesha Hietala, Eleftherios Ioannidis, John H. Kastner, Anwar Mamat, Darin McAdams, Matt McCutchen, Neha Rungta, Emina Torlak, and Andrew Wells. 2024. Cedar: A New Language for Expressive, Fast, Safe, and Analyzable Authorization. 8, OOPSLA1 (2024), 670–697. https://doi.org/10.1145/3649835

[30] Thibault Dardinier and Peter Müller. 2024. Hyper Hoare Logic: (Dis-)Proving Program Hyperproperties. In *PLDI*, Vol. 8. 1485–1509. https://doi.org/10.1145/3656437

[31] Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, James Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, and Jonathan Woodruff. 2019. CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment. In *ASPLOS*. 379–393. https://doi.org/10.1145/3297858.3304042

[32] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. 2006. Ambient-Oriented Programming in AmbientTalk. In *ECOOP*. 230–254. https://doi.org/10.1007/11785477_16

[33] Jack B. Dennis and Earl C. Van Horn. 1966. Programming Semantics for Multiprogrammed Computations. *Comm. ACM* 9, 3 (1966), 143–155. https://doi.org/10.1145/365230.365252

[34] Dominique Devriese, Lars Birkedal, and Frank Piessens. 2016. Reasoning about Object Capabilities with Logical Relations and Effect Parametricity. In *IEEE EuroS&P*. 147–162. https://doi.org/10.1109/EuroSP.2016.22

[35] W. Dietl, S. Drossopoulou, and P. Müller. 2007. Generic Universe Types. In *ECOOP (LNCS, Vol. 4609)*. Springer, 28–53. http://www.springerlink.com

[36] Werner Dietl, Sophia Drossopoulou, and Peter Müller. 2007. Generic Universe Types. In *ECOOP*, Vol. 4609. 28–53. https://doi.org/10.1007/978-3-540-73589-2_3

[37] W. Dietl and P. Müller. 2005. Universes: Lightweight Ownership for JML. *JOT* 4, 8 (October 2005), 5–32. https://doi.org/10.5381/jot.2005.4.8.a1

[38] Christos Dimoulas, Scott Moore, Aslan Askarov, and Stephen Chong. 2014. Declarative Policies for Capability Control. In *Computer Security Foundations Symposium (CSF)*. 3–17. https://doi.org/10.1109/CSF.2014.9

[39] Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. 2009. Deny-guarantee reasoning. In *ESOP*. 363–377. https://doi.org/10.1007/978-3-642-00590-9_26

[40] Emanuele D'Osualdo, Azadeh Farzan, and Derek Dreyer. 2022. Proving hypersafety compositionally. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 289–314. https://doi.org/10.1145/3563298

[41] S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers. 2008. A Unified Framework for Verification Techniques for Object Invariants. In *ECOOP*. 412–437. https://doi.org/10.1007/978-3-540-70592-5_18

[42] Sophia Drossopoulou and James Noble. 2013. The need for capability policies. In *FTfJP*. 61–67. https://doi.org/10.1145/2489804.2489811

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

[44] Sophia Drossopoulou, James Noble, Mark Miller, and Toby Murray. 2016. Permission and Authority revisited – towards a formalization. In *(FTfJP)*. 1 – 6. http://dl.acm.org/citation.cfm?id=2955821

[45] J. C Filliatre, L. Gondelman, and A Pakevichl. 2016. The spirit of ghost code. In *Formal Methods System Design*. 1–16. https://doi.org/10.1007/978-3-319-08867-9_1

[46] Robert Bruce Findler and Matthias Felleisen. 2001. Contract Soundness for object-oriented languages. In *OOPSLA*. 1–15. https://doi.org/10.1145/504282.504283

[47] Robert W. Floyd. 1967. Assigning Meanings to Programs. *Marhematical Aspects of Computer Science* 19 (1967), 19–32. https://people.eecs.berkeley.edu/~necula/Papers/FloydMeaning.pdf

[48] 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. https://doi.org/10.1145/3623510

[49] 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

[50] Philipp Haller. 2024. Lightweight Affine Types for Safe Concurrency in Scala (Keynote). In *Programming*. https://doi.org/10.1145/3660829.3661033 https://speakerdeck.com/phaller/towards-safer-lightweight-concurrency-in-scala.

[51] Philipp Haller and Alexander Loiko. 2016. LaCasa: lightweight affinity and object capabilities in Scala. In *OOPSLA*. 272–291. https://doi.org/10.1145/2983990.2984042

[52] Ian J. Hayes and Cliff B. Jones. 2018. A Guide to Rely/Guarantee Thinking. In *SETSS 2017*. 1–38. https://doi.org/10.1007/978-3-030-02928-9_1

[53] Ian J. Hayes, Xi Wu, and Larissa A. Meinicke. 2017. Capabilities for Java: Secure Access to Resources. In *APLAS*. 67–84. https://doi.org/10.1007/978-3-319-71237-6_4

[54] Trent Hill, James Noble, and John Potter. 2002. Scalable Visualizations of Object-Oriented Systems with Ownership Trees. *J. Vis. Lang. Comput.* 13, 3 (2002), 319–339. https://doi.org/10.1006/jvlc.2002.0238

[55] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Comm. ACM* 12 (1969), 576–580. https://doi.org/10.1145/363235.363259

[56] C. A. R. Hoare. 1974. Monitors: an operating system structuring concept. *Commun. ACM* 17, 10 (1974), 549–557. https://doi.org/10.1145/355620.361161

[57] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *ACM ToPLAS* 23, 3 (2001), 396–450. https://doi.org/10.1145/503502.503505

[58] S. S. Ishtiaq and P. W. O'Hearn. 2001. BI as an assertion language for mutable data structures. In *POPL*. 14–26. https://doi.org/10.1145/360204.375719

[59] Timothy Jones, Michael Homer, James Noble, and Kim B. Bruce. 2016. Object Inheritance Without Classes. In *ECOOP*. 13:1–13:26. https://doi.org/10.4230/LIPIcs.ECOOP.2016.13

[60] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. 2, POPL (2017), 66:1–66:34. https://doi.org/10.1145/3158154

[61] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

[62] Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language* (2nd ed.). No Starch Press. https://doc.rust-lang.org/book/

[63] Alexander Kogtenkov, Bertrand Meyer, and Sergey Velder. 2015. Alias calculus, change calculus and frame inference. *Sci. Comp. Prog.* 97 (2015), 163–172. https://doi.org/10.1016/j.scico.2013.11.006

[64] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. In *OOPSLA*, Vol. 7. 286–315. https://doi.org/10.1145/3586037

[65] Dirk Leinenbach and Thomas Santen. 2009. Verifying the Microsoft Hyper-V Hypervisor with VCC. In *Formal Methods*. 806–809. https://doi.org/10.1007/978-3-642-05089-3_51

[66] K. R. M. Leino and P. Müller. 2004. Object Invariants in Dynamic Contexts. In *ECOOP*. 491–516. https://doi.org/10.1007/978-3-540-24851-4_22

[67] K. Rustan M. Leino and Wolfram Schulte. 2007. Using History Invariants to Verify Observers. In *ESOP*. 80–94. https://doi.org/10.1007/978-3-540-71316-6_7

[68] Henry M. Levy. 1984. *Capability-Based Computer Systems*. Butterworth-Heinemann. https://homes.cs.washington.edu/~levy/capabook/

[69] Barbara Liskov and Jeanette Wing. 1994. A Behavioral Notion of Subtyping. *ACM ToPLAS* 16, 6 (1994), 1811–1841. https://www.cs.cmu.edu/~wing/publications/LiskovWing94.pdf

[70] Zongyuan Liu, Sergei Stepanenko, Jean Pichon-Pharabod, Amin Timany, Aslan Askarov, and Lars Birkedal. 2023. VMSL: A Separation Logic for Mechanised Robust Safety of Virtual Machines Communicating above FF-A. 7, PLDI (2023), 1438–1462. https://doi.org/10.1145/3591279

[71] Anton Lorenzen, Stephen Dolan, Richard A. Eisenberg, and Sam Lindley. 2024. Oxidizing OCaml with Modal Memory Management. In *ICFP*. 448–475. https://doi.org/10.1145/3674642

[72] Y. Lu and J. Potter. 2006. Protecting Representation with Effect Encapsulation.. In *POPL*. 359–371. https://dl.acm.org/doi/10.1145/1111320.1111069

[73] Matthew Lutze, Magnus Madsen, Philipp Schuster, and Jonathan Immanuel Brachthäuser. 2023. With or Without You: Programming with Effect Exclusion. *ICFP* (2023), 448–475. https://doi.org/10.1145/3607846

[74] Julian Mackay, Susan Eisenbach, James Noble, and Sophia Drossopoulou. 2022. *Necessity* Specifications for Robustness. *Proc. ACM Program. Lang.* 6, OOPSLA2, 811–840. https://doi.org/10.1145/3563317

[75] S. Maffeis, J.C. Mitchell, and A. Taly. 2010. Object Capabilities and Isolation of Untrusted Web Applications. In *Proc of IEEE Security and Privacy*. 125–140. https://ieeexplore.ieee.org/document/5504710

[76] Daniel Marshall and Dominic Orchard. 2024. Functional Ownership through Fractional Uniqueness. In *OOPSLA*. 1040–1070. https://doi.org/10.1145/3649848

[77] Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. 2022. RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code. In *PLDI*. ACM, 841–856. https://dl.acm.org/doi/10.1145/3519939.3523704

[78] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2021. RustHorn: CHC-based Verification for Rust Programs. *TOPLAS* (2021), 15:1–15:54. https://doi.org/10.1145/3462205

[79] Darya Melicher, Yangqingwei Shi, Alex Potanin, and Jonathan Aldrich. 2017. A Capability-Based Module System for Authority Control. In *ECOOP*. 20:1–20:27. https://doi.org/10.4230/LIPIcs.ECOOP.2017.20

[80] Adrian Mettler, David Wagner, and Tyler Close. 2010. Joe-E a Security-Oriented Subset of Java. In *NDSS*. 357–374. https://www.ndss-symposium.org/ndss2010/joe-e-security-oriented-subset-java

[81] Bertrand Meyer. 1992. Applying "Design by Contract". *Computer* 25, 10 (1992), 40–51. https://doi.org/10.1109/2.161279

[82] B. Meyer. 1992. *Eiffel: The Language*. Prentice Hall.

[83] Mark Samuel Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Ph. D. Dissertation. Johns Hopkins University, Baltimore, Maryland. https://papers.agoric.com/assets/pdf/papers/robust-composition.pdf

[84] Mark Samuel Miller. 2011. Secure Distributed Programming with Object-capabilities in JavaScript. (Oct. 2011). Talk at Vrije Universiteit Brussel, mobicrant-talks.eventbrite.com.

[85] Mark Samuel Miller, Tom Van Cutsem, and Bill Tulloh. 2013. Distributed Electronic Rights in JavaScript. In *ESOP*. 1–20. https://doi.org/10.1007/978-3-642-37036-6_1

[86] Mark Samuel Miller, Chip Morningstar, and Bill Frantz. 2000. Capability-based Financial Instruments: From Object to Capabilities. In *Financial Cryptography*. 349–378. https://doi.org/10.1007/3-540-45472-1_24

[87] Mark Samuel Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. 2008. Safe active content in sanitized JavaScript. , 26 pages. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=c2de7d8991bdb875fb06d5e5455b0862f0a2d15b Google white paper.

[88] Nick Mitchell. 2006. The Runtime Structure of Object Ownership. In *ECOOP*. 74–98. https://doi.org/10.1007/11785477_5

[89] Scott Moore, Christos Dimoulas, Robert Bruce Findler, Matthew Flatt, and Stephen Chong. 2016. Extensible access control with authorization contracts. In *OOPSLA*. 214–233. https://doi.org/10.1145/2983990.2984021

[90] James H. Morris Jr. 1973. Protection in Programming Languages. *CACM* 16, 1 (1973), 15–21. https://doi.org/10.1145/361932.361937

[91] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. 2006. Modular Invariants for Layered Object Structures. *Science of Computer Programming* 62 (2006), 253–286. https://doi.org/10.1016/j.scico.2006.03.001

[92] Toby Murray. 2010. *Analysing the Security Properties of Object-Capability Patterns*. Ph. D. Dissertation. University of Oxford. http://ora.ox.ac.uk/objects/uuid:98b0b6b6-eee1-45d5-b32e-d98d1085c612

[93] Takashi Nakayama, Yusuke Matsushita, Ken Sakayori, Ryosuke Sato, and Naoki Kobayashi. 2024. Borrowable Fractional Ownership Types for Verification. In *VMCAI*. 224–246. https://doi.org/10.1007/978-3-031-50521-8_11

[94] James Noble, John Potter, and Jan Vitek. 1998. Flexible Alias Protection. In *ECOOP*. 158–185. https://doi.org/10.1007/BFb0054091

[95] Peter W. O'Hearn. 2019. Incorrectness Logic. 4, POPL (2019), 1–32. https://doi.org/10.1145/3371078

[96] Leo 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.

[97] M. Parkinson and G. Bierman. 2005. Separation logic and abstraction. In *POPL*. 247–258. https://doi.org/10.1145/1040305.1040326

[98] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2020. VerX: Safety Verification of Smart Contracts. In *IEEE Symp. on Security and Privacy*. 1661–1677. https://doi.org/10.1109/SP40000.2020.00024

[99] John Potter, James Noble, and David G. Clarke. 1998. The Ins and Outs of Objects. In *Australian Software Engineering Conference*. 80–89. https://doi.org/10.1109/ASWEC.1998.730915

[100] Xiaojia Rao, 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. In *PLDI*. 1096 – 1120. https://doi.org/10.1145/3591265

[101] 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. https://doi.org/10.1145/3591265

[102] J. C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. IEEE Computer Society, 55–74. https://doi.org/10.1109/LICS.2002.1029817

[103] Dustin Rhodes, Tim Disney, and Cormac Flanagan. 2014. Dynamic Detection of Object Capability Violations Through Model Checking. In *DLS*. 103–112. https://doi.org/10.1145/2661088.2661099

[104] Victor Rivera and Bertrand Meyer. 2020. AutoAlias: Automatic Variable-Precision Alias Analysis for Object-Oriented Programs. *SN Comp. Sci.* 1, 1 (2020), 12:1–12:15. https://doi.org/10.1007/s42979-019-0012-1

[105] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: automating the foundational verification of C code with refined ownership types. In *PLDI*. 158–174. https://dl.acm.org/doi/10.1145/3453483.3454036

[106] Ina Schaefer, Tobias Runge, Alexander Knüppel, Loek Cleophas, Derrick G. Kourie, and Bruce W. Watson. 2018. Towards Confidentiality-by-Construction. In *ISOLA*. 502–515. https://doi.org/10.1007/978-3-030-03418-4_30

[107] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan. 2019. Safer Smart Contract Programming with Scilla. In *OOPSLA*, Vol. 3. 1–30. https://doi.org/10.1145/3360611

[108] Randall B. Smith and David M. Ungar. 1995. Programming as an Experience: The Inspiration for Self. In *ECOOP*, Walter G. Olthoff (Ed.), Vol. 952. Springer, 303–330. https://doi.org/10.1007/3-540-49538-X_15

[109] Alexander J. Summers and Sophia Drossopoulou. 2010. Considerate Reasoning and the Composite Pattern. In *VMCAI*. 328–344. https://doi.org/10.1007/978-3-642-11319-2_24

[110] Alexander J. Summers, Sophia Drossopoulou, and Peter Müller. 2009. Universe-Type-Based Verification Techniques for Mutable Static Fields and Methods. *J. Object Technol.* 8, 4 (2009), 85–125. https://doi.org/10.5381/JOT.2009.8.4.A4

[111] David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and compositional verification of object capability patterns. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 89:1–89:26. https://doi.org/10.1145/3133913

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

[113] David M. Ungar and Randall B. Smith. 1991. SELF: The Power of Simplicity. *LISP Symb. Comput.* 4, 3 (1991), 187–205. https://doi.org/10.1007/3-540-49538-X_15

[114] Stephan van Staden. 2015. On Rely-Guarantee Reasoning. In *MPC*. 30–49. https://link.springer.com/chapter/10.1007/978-3-319-19797-5_2

[115] Guannan Wei, Danning Xie, Wuqi Zhang, Yongwei Yuan, and Zhuo Zhang. 2024. Consolidating Smart Contracts with Behavioral Contracts. In *PLDI*. 965 – 989. https://doi.org/10.1145/3656416

[116] Maurice V. Wilkes and Roger M. Needham. 1980. The Cambridge Model Distributed System. *ACM SIGOPS Oper. Syst. Rev.* 14, 1 (1980), 21–29. https://doi.org/10.1145/850693.850695

[117] Anxhelo Xhebraj, Oliver Bracevac, Guannan Wei, and Tiark Rompf. 2022. What If We Don't Pop the Stack? The Return of 2nd-Class Values. In *ECOOP*. 15:1–15:29. https://doi.org/10.4230/LIPIcs.ECOOP.2022.15

[118] Yichen Xu and Martin Odersky. 2024. A Formal Foundation of Reach Capabilities. In *Programming*. 134–138. https://doi.org/10.1145/3660829.3660851

[119] Conrad Zimmerman, Jenna DiVincenzo, and Jonathan Aldrich. 2024. Sound Gradual Verification with Symbolic Execution. *POPL* 8, 2547–2576. https://doi.org/10.1145/3632927