Reasoning about External Calls

ANONYMOUS AUTHOR(S)

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

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

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

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

1 INTRODUCTION

External calls. In today's complex software, internal, trusted, code is tightly intertwined with external, untrusted, code: external code calls into internal code, internal code calls out to external code and external code even calls back into internal code — all within the same call chain.

This paper addresses reasoning about *external calls* — when trusted internal code calls out to untrusted, unknown external code. This reasoning is hard because by "external code" we mean untrusted code where we don't have a specification. External code may even have been written by an attacker trying to subvert or destroy the whole system.

In this code sketch, method `m1`'s code is trusted, method `m2` takes an untrusted parameter `untrst`, and then at line 6 calls an unknown external method `unkn` passing itself as an argument. The challenge is: what can that method call do? what effects will it have? What if `untrst` calls back into M_{intl}?

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

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

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

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

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

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

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

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

The approaches above do not aim to support indirect, eventual access to capabilities. Drossopoulou et al. [37] and Mackay et al. [67] do describe such access; the former proposes "holistic specifications" to describe a module's emergent behaviour. and the latter develops a tailor-made logic to prove that modules which do not contain external calls adhere to such specifications. Rather than relying on problem-specific, custom-made proofs, we propose a Hoare logic that addresses access to capabilities, external calls, and the module's tamed effects.

This paper's contributions. (1) assertions to describe access to capabilities, (2) a specification language to describe taming of effects, 3) a Hoare logic to reason about external calls and to prove that modules satisfy their tamed effects specifications, 4) proof of soundness, 5) a worked illustrative example with a mechanised proof in Coq.

Structure of this paper. Sect. 2 outlines the main ingredients of our approach in terms of an example. Sect. 3 outlines a simple object-oriented language used for our work. Sect. 4 contains essential concepts for our study. Sect. 5 and Sect 7 give syntax and semantics of assertions, and specifications, while Sect. 6 discusses preservation of satisfaction of assertions. Sect. 8 develops Hoare triples and quadruples to prove external calls, and that a module adheres to its tamed effects specifications. Sect. 9 outlines our proof of soundness of the Hoare logic. Sect. 10 summarises the Coq proof of our running example (the source code will be submitted as an artefact). Sect. 11 concludes with related work. Fuller technical details can be found in the appendices in the accompanying materials.

2 THE PROBLEM AND OUR APPROACH

We introduce the problem through an example, and outline our approach. We work with a small, class-based object-oriented language similar to Joe-E [73] with modules, module-private fields (accessible only from methods from the same module), and unforgeable, un-enumerable addresses. We distinguish between *internal* objects — instances of our internal module M's classes — and *external* objects defined in *any* number of external modules $\overline{M}$. `Private` methods may only be called by objects of the same module, while `public` methods may be called by *any* object with a reference to the method receiver, and with actual arguments of dynamic types that match the declared formal parameter types. [1]

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

`Shop` – illustrating tamed effects

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

```
1  module M_shop
2    ...
3    class Shop
4      field accnt:Account, invntry:Inventory, clients:[external]
5      public method buy(buyer:external, anItem:Item)
6        int price = anItem.price
7        int oldBlnce = this.accnt.blnce
8        buyer.pay(this.accnt, price)     // external call!
9        if (this.accnt.blnce == oldBlnce+price)
10          this.send(buyer,anItem)
11       else
12          buyer.tell("you have not paid me")
13     private method send(buyer:external, anItem:Item)
14       ...
```

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

What are the possible effects of that external call? The `Shop` hopes, but cannot be sure, that at line 9 it will have received money; but it wants to be certain that the `buyer` can not use this opportunity to access the shop's account to drain its money. Can `Shop` be certain?

[1] As in Joe-E, we leverage module-based privacy to restrict propagation of capabilities, and reduce the need for reference monitors etc, *c.f.* Sect 3 in [73].

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

(A) If prior to the call of `buy`, the `buyer` has no (eventual) access to the account's key, and
(B) If M_{shop} ensures that a) access to keys is not leaked to external objects, and b) funds cannot be withdrawn unless the external entity responsible for the withdrawal has eventual access to the account's key, then
(C) The external call on line 8 will not result in a decrease in the shop's account balance.

The remit of this paper is to provide specification and verification tools that support arguments like the one above. In that example, we relied on two yet-to-be-defined concepts: (A) "eventual access" and (B) tamed effects (e.g., no money withdrawn unless certain conditions are met). Therefore, we need to address the following three challenges:

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

2.1 1^{st} Challenge: eventual access

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

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

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

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

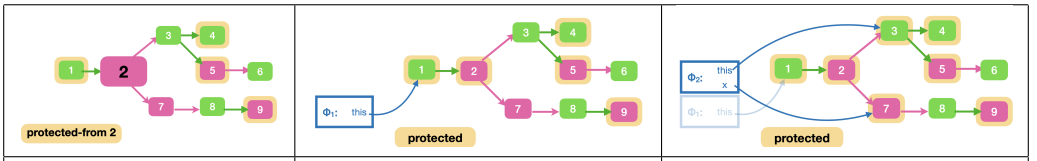


Fig. 1. Protected from and Protected. Pink and green squares are external and internal objects, respectively. Connecting straight arrows indicate fields. Blue boxes are frames on the stack. Protected objects are highlighted in yellow. The left pane shows a heap with objects o_1-o_9. Here o_1, o_4, o_5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

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

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

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

Example 2.1. The following scoped invariants

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

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

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

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

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

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

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

Example 2.3. We illustrate the meaning of our specifications using three versions of a class `Account` from [67] as part of our internal module M_{shop}. To differentiate, we rename M_{shop} as $\mathtt{M}_{good}$, $\mathtt{M}_{bad}$, or $\mathtt{M}_{fine}$. All use the same `transfer` method for withdrawing money.

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

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

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

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

Thus, in all three modules, the key is an object capability which *enables* the withdrawal of the money. Moreover, in $\mathtt{M}_{good}$ and $\mathtt{M}_{fine}$, the key is a capability used to *tame* withdrawal of money, preventing those without it from getting the money from the account. Crucially, in $\mathtt{M}_{bad}$ the key *does not tame* withdrawal of money. Using $\mathtt{M}_{bad}$, it is possible to start in a state where the account's key is unknown, modify the key, and then withdraw the money. Code such as

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

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

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

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

2.3 3^{rd} Challenge: A Hoare logic

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

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

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

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

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

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

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

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

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

Summary

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

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

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

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

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

This work is based on $\mathscr{L}_{ul}$, a small, imperative, sequential, class based, typed, object-oriented language. We believe, however, that the work can easily be adapted to any capability safe language with some form of encapsulation. Wrt to encapsulation and capability safety, $\mathscr{L}_{ul}$ supports private fields, private and public methods, unforgeable addresses, and no ambient authority (no static methods, no address manipulation). It has a simple concept of module with module-private fields and methods, described in Sect. 3.2. The definition of $\mathscr{L}_{ul}$ can be found in Appendix A.[5].

A $\mathscr{L}_{ul}$ state, σ, consists of a heap χ, and a stack. A stack is a sequence of frames, $\phi_1 \cdot \ldots \cdot \phi_n$. A frame, ϕ, consists of a local variable map and a continuation, *i.e.* a sequence of statements to be executed. The top frame in a state $(\phi_1 \cdot \ldots \cdot \phi_n, \chi)$ is ϕ_n.

Notation. We adopt the following unsurprising notation:

- An object is uniquely identified by the address that points to it. We shall be talking of objects o, o' when talking less formally, and of addresses, α, α', α_1, ... when more formal.
- x, x', y, z, u, v, w are variables.
- $\alpha \in \sigma$ means that α is defined in the heap of σ, and $x \in \sigma$ means that x is defined in the top frame of σ. Conversely, $\alpha \notin \sigma$ and $x \notin \sigma$ have the obvious meanings. $\lfloor \alpha \rfloor_\sigma$ is α; and $\lfloor x \rfloor_\sigma$ is the value to which x is mapped in the top-most frame of σ's stack, and $\lfloor e.f \rfloor_\sigma$ looks up in σ's heap the value of f for the object $\lfloor e \rfloor_\sigma$.
- $\phi[x \mapsto \alpha]$ updates the variable map of ϕ, and $\sigma[x \mapsto \alpha]$ updates the top frame of σ.
- $A[e/x]$ is textual substitution where we replace all occurrences of x in A by e.
- As usual, $\overline{q}$ stands for sequence q_1, ... q_n, where q can be an address, a variable, a frame, an update or a substitution. Thus, $\sigma[\overline{x \mapsto \alpha}]$ and $A[\overline{e/y}]$ have the expected meaning.
- $\phi.\texttt{cont}$ is the continuation of frame ϕ, and $\sigma.\texttt{cont}$ is the continuation in the top frame.
- $text_1 \overset{\text{txt}}{=} text_2$ expresses that $text_1$ and $text_2$ are textually equal.
- We define the depth of a stack as $|\phi_1 \ldots \phi_n| \triangleq n$, For states, $|(\overline{\phi}, \chi)| \triangleq |\overline{\phi}|$. The operator $\sigma[k]$ truncates the stack up to the k-th frame: $(\phi_1 \ldots \phi_k \ldots \phi_n, \chi)[k] \triangleq (\phi_1 \ldots \phi_k, \chi)$
- $Vs(stmt)$ returns the variables which appear in $stmt$. For example, $Vs(u := y.f) = \{u, y\}$.

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

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

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

The semantics is unsurprising : In σ, the top frame's continuation contains the statement to be executed next. Statements may assign to variables, allocate new objects, perform field reads and writes on objects, and call methods on those objects. When a method is called, a new frame is pushed onto the stack; this frame maps `this` and the formal parameters to the values for the receiver and other arguments, and the continuation to the body of the method. Methods are expected to store their return values in the variable `res`. When the continuation is empty (ϵ), the frame is popped and the value of `res` [6] from the popped frame is stored in the variable map of the top

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

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

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

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

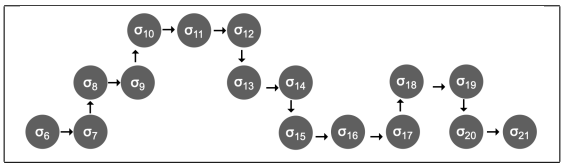


Fig. 2. $\rightarrow$: step within the same method; $\uparrow$: entering a method; $\downarrow$: returning from a method

4 FUNDAMENTAL CONCEPTS

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

4.1 Method Calls and Returns

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

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

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

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

4.2 Scoped Execution

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

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

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

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

Definition 4.2 (Scoped Execution). :

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

Consider Fig. 2: Here $|\sigma_8| \leq |\sigma_9|$ and thus $\overline{M};\ \sigma_8 \rightsquigarrow \sigma_9$. Also, $\overline{M}; \sigma_{14} \dashrightarrow \sigma_{15}$ but $|\sigma_{14}| \not\leq |\sigma_{15}|$ (this step returns from the active call in σ_{14}), and hence $\overline{M};\ \sigma_{14} \not\rightsquigarrow \sigma_{15}$. Finally, even though $|\sigma_8| = |\sigma_{18}|$ and $\overline{M}; \sigma_8 \dashrightarrow^* \sigma_{18}$, we have $\overline{M};\ \sigma_8 \not\rightsquigarrow^* \sigma_{18}$: in This is so, because the execution $\overline{M}; \sigma_8 \dashrightarrow^* \sigma_{18}$ goes through the step $\overline{M}; \sigma_{14} \dashrightarrow \sigma_{15}$ and $|\sigma_8| \not\leq |\sigma_{15}|$ (this step returns from the active call in σ_8).

The relation $\rightsquigarrow^*$ contains more than the transitive closure of $\rightsquigarrow$. *E.g.*,, $\overline{M};\ \sigma_9 \rightsquigarrow^* \sigma_{13}$, even though $\overline{M};\ \sigma_{12} \not\rightsquigarrow \sigma_{13}$. Nevertheless, Lemma 4.3 says, essentially, that scoped executions describe the same set of executions as those starting at an initial state[7]. For instance, revisit Fig. 2, and assume that σ_6 is an initial state. We have $\overline{M}; \sigma_{10} \dashrightarrow^* \sigma_{14}$ and $\overline{M};\ \sigma_{10} \not\rightsquigarrow^* \sigma_{14}$, but also $\overline{M};\ \sigma_6 \rightsquigarrow^* \sigma_{14}$.

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

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

Lemma 4.4 says that scoped execution does not affect the contents of variables in earlier scopes. and that the interpretation of a variable remains unaffected by scoped execution of statements which do not mention that variable. More in Appendix B.

Lemma 4.4. For any modules $\overline{M}$, states σ, σ', variable y, and number k:

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

4.3 Locally Reachable Objects

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

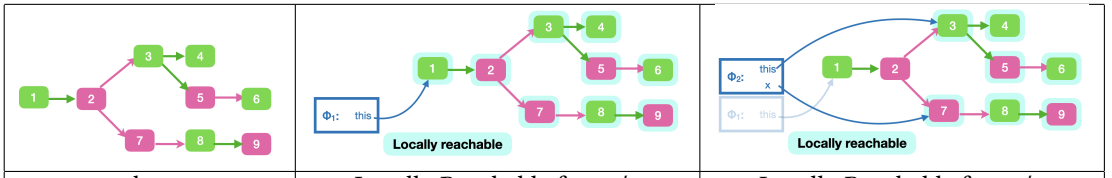
Definition 4.5. We define

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

We illustrate these concepts in Fig. 3: In the middle pane the top frame is ϕ_1 which maps `this` to o_1; all objects are locally reachable. In the right pane the top frame is ϕ_2, which maps `this` to o_3, and x to o_7; now o_1 and o_2 are no longer locally reachable.

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

[7] An *Initial* state's heap contains a single object of class `Object`, and its stack consists of a single frame, whose local variable map is a mapping from `this` to the single object, and whose continuation is any statement. (See Def. A.7)



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

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

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

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

Consider Fig. 2. Lemma 4.6, part 2 promises that any objects locally reachable in σ_{14} which already existed in σ_8, were locally reachable in σ_8. However, the lemma is only applicable to scoped execution, and as $\overline{M}; \sigma_8 \not\rightsquigarrow^* \sigma_{17}$, the lemma does not promise that objects locally reachable in σ_{17} which already existed in σ_8, were locally accessible in σ_8 – namely it could be that objects are made globally reachable upon method return, during the step from σ_{14} to σ_{15}.

5 ASSERTIONS

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

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

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

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

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

Definition 5.2 (Shorthands). We write $e : \texttt{intl}$ for $\neg(e : \texttt{extl})$, and $\texttt{extl}$. resp. $\texttt{intl}$ for $\texttt{this} : \texttt{extl}$ resp. $\texttt{this} : \texttt{intl}$. Forms as $A \longrightarrow A'$, $A \vee A'$, and $\exists x : C.A$ can be encoded.

Satisfaction of Assertions by a module and a state is expressed through $M, \sigma \models A$ and defined by cases on the shape of A, in definitions 5.3, 5.4, and 5.4.

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

5.1 Semantics of assertions – first part

To determine satisfaction of an expression, we use the evaluation relation, $M, \sigma, e \hookrightarrow v$, which says that the expression e evaluates to value v in the context of state σ and module M. As expressions in $\mathscr{L}_{ul}$ may be recursively defined, their evaluation need not terminate. Nevertheless, the logic of A remains classical because recursion is restricted to expressions, and not generally to assertions.

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

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

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

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

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

5.2 Semantics of Assertions - second part

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

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

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

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

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

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

Figure 4 illustrates "protected from" and "protected". Pink and green indicate external and internal objects respectively. In the first row we highlight in yellow the objects protected from other objects. Thus, all objects except o_6 are protected from o_5 (left pane); all objects expect o_8 are protected from o_7 (middle pane); and all objects except o_3, o_6, o_7, and o_8 are protected from o_2 (right pane).

Note o_6 is not protected from o_2. Even through o_3 is internal, and is on the path from o_2 to o_6, it is not the penultimate object on that path. Therefore, o_2 can make a call to o_3, and then this call can return o_5. Once o_2 has access to o_5, it can also get access to o_6. The example justifies why we require that the *penultimate* object is internal.

In the third row of Figure 4 we show three states: σ_1 has top frame ϕ_1, which has one variable, `this`, pointing to o_1, while σ_2 has top frame ϕ_2; it has two variables, `this` and `x` pointing to

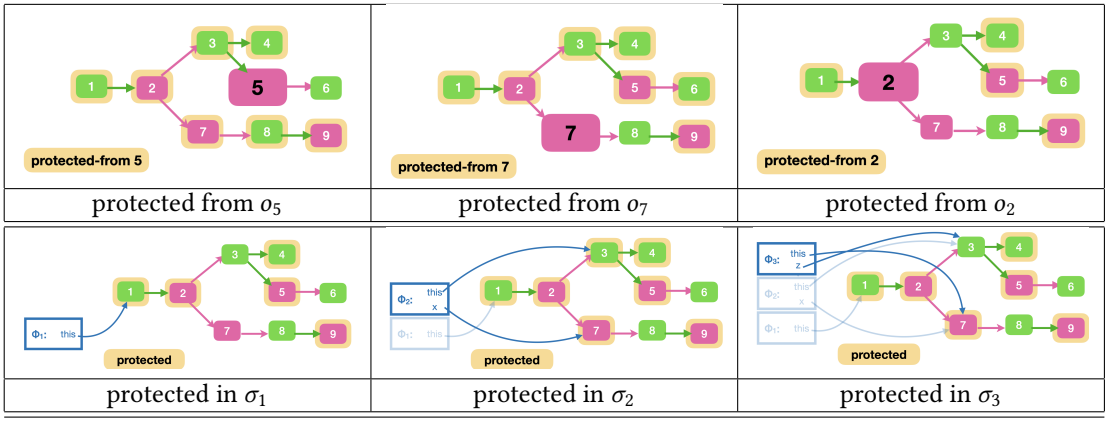


Fig. 4. Protection. Pink objects are external, and green objects are internal.

o_3 and o_7, and σ_3 has top frame ϕ_3; it has two variables, `this` and `x`, pointing to o_7 and o_3. We also highlight the protected objects with a yellow halo. Note that o_3 is protected in σ_2, but is not protected in σ_3. This is so, because $\lfloor \texttt{this} \rfloor_{\sigma_3}$ is external, and o_3 is an argument to the call. As a result, during the call, o_7 may obtain unmitigated access to o_3.

Discussion. Lack of eventual access is a central concept in the verification of code with calls to and callbacks from untrusted code. It has already been over-approximated in several different ways, *e.g.* 2nd-class [88, 108] or borrowed ("2nd-hand") references [14, 22], textual modules [67], information flow [102], runtime checks [4], abstract data type exports [63], separation-based invariants Iris [41, 93], – more in § 11. In general, protection is applicable in more situations (i.e. is less restrictive) than most of these approaches, although more restrictive than the ideal "lack of eventual access".

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

6 PRESERVATION OF ASSERTIONS

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

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

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

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

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

6.1 Stability

In most program logics, satisfaction of variable-free assertions is preserved when pushing/popping frames – *i.e.* immediately after entering a method or returning from it. This, however, is not the case for our assertions, where whether α is protected, *i.e.* whether $\langle\!\langle \alpha \rangle\!\rangle$ holds, depends on the heap as well as the set of objects reachable from the top frame; the latter changes when the frame changes. This is shown, *e.g.* in Fig. 4 where $\sigma_1, \chi \models \langle\!\langle o_3 \rangle\!\rangle$, then $.\sigma_2, \chi \not\models \langle\!\langle o_3 \rangle\!\rangle$, and then $\sigma_3, \chi \models \langle\!\langle o_3 \rangle\!\rangle$

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

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

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

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

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

Example 6.4 (Stb^+ not always preserved by Method Return)**.** Assume state σ_a, such that $\lfloor \texttt{this} \rfloor_{\sigma_a} = o_1$, $\lfloor \texttt{this}.f \rfloor_\sigma = o_2$, $\lfloor x \rfloor_\sigma = o_3$, $\lfloor x.f \rfloor_\sigma = o_2$, and $\lfloor x.g \rfloor_\sigma = o_4$, where o_2 is external and all other objects are internal. We then have $.., \sigma_a \models \langle o_4\rangle$. Assume the continuation of σ_a consists of a method $x.m()$. Then, upon entry to that method, when we push the new frame, we have state σ_b, which also satisfies $.., \sigma_b \models \langle o_4\rangle$. Assume the body of m is $\texttt{this}.f.m1(\texttt{this}.g); \texttt{this}.f := \texttt{this}; \texttt{this}.g := \texttt{this}$, and the external method $m1$ stores in the receiver a reference to the argument. Then, at the end of method execution, and before popping the stack, we have state σ_c, which also satisfies $.., \sigma_c \models \langle o_4\rangle$. However, after we pop the stack, we obtain σ_d, for which $.., \sigma_d \not\models \langle o_4\rangle$.

We work with Stb^+ assertions (the $Stbl$ requirement is too strong). But we need to address the lack of preservation of Stb^+ assertions for external method calls and returns. We do the former through *adaptation* ($-\!\triangledown$ in Sect 6.2), and the latter through *scoped satisfaction* (§9.1).

6.2 Adaptation

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

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

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

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

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

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

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

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

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

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

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

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

6.3 Encapsulation

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

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

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

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

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

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

7 SPECIFICATIONS

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

7.1 Specifications Syntax

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

Definition 7.1 (Specifications).

- The syntax of specifications, S, is given below

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

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

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

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

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

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

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

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

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

Note that in S_9 the variables a, a' are disjoint from `this` and the formal parameters of `set`. In that sense, a and a' are universally quantified; a call of `set` will preserve protectedness for *all* accounts and their keys.

7.2 Specifications Semantics

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

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

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

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

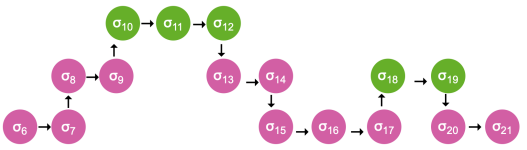
(1) $M \models \forall \overline{x : C}.\{A\} \triangleq \forall \overline{M}, \sigma.[\ \overline{M}; M \models \{\texttt{extl} \wedge \overline{x : C} \wedge A\}\,\sigma\,\{A\} \parallel \{A\}\ ]$.

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

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

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

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



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

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

8 HOARE LOGIC

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

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

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

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

8.1 Preliminaries: Specification Lookup, Renamings, Underlying Hoare Logic

First some preliminaries: The judgement $\vdash M : S$ expresses that S is part of M's specification. In particular, it allows *safe renamings*. These renamings are a convenience, akin to the Barendregt convention, and allow simpler Hoare rules – *c.f.* Sect. 8.4, Def. F.1, and Ex. F.2. We also require an underlying Hoare logic with judgements $M \vdash_{ul} \{A\}stmt\{A'\}$, – *c.f.* Ax. F.3.

8.2 First Phase: Triples

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

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

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

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

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

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

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

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

With rule EMBED_U in Fig. 6, any assertion $M \vdash_{ul} \{A\}stmt\{A'\}$ whose statement does not contain a method call, and which can be proven in the underlying Hoare logic, can also be proven in our logic. In PROT-1, we see that protection of an object o is preserved by internal code which does not call any methods: namely any heap modifications will ony affect internal objects, and this will not

expose new, unmitigated external access to o. Prot-2,Prot-3 and Prot-4 describe the preservation of relative protection. Proofs of these rules can be found in App. G.5.1. Note that "protection" of an object can decrease is if we call an external method, and pass it as argument. This will be covered by the rule in Fig. 7.

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

8.3 Second Phase: Quadruples

8.3.1 Introducing invariants, and substructural rules. We now introduce quadruple rules. Rule Mid embeds triples $M \vdash \{ A \}\ s\ \{ A' \}$ into quadruples $M \vdash \{ A \}\ s\ \{ A' \} \parallel \{ A'' \}$; this is sound, because s is guaranteed not to contain method calls (by lemma 8.1)

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

The remaining substructural quadruple rules appear in Fig. 14, and are as expected: Rules Sequ and Consequ are the usual rules for statement sequences and consequence, adapted to quadruples. Rule Combine combines two quadruples for the same statement into one. The last three rules apply to *any* statements – even those containing method calls.

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

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

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

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

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

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

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

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

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

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

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

8.4 Third phase: Proving adherence to Module Specifications

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

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

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

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

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

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

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

9 SOUNDNESS

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

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

9.1 Scoped Satisfaction

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

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

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

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

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

9.2 Soundness of the Hoare Triples Logic

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

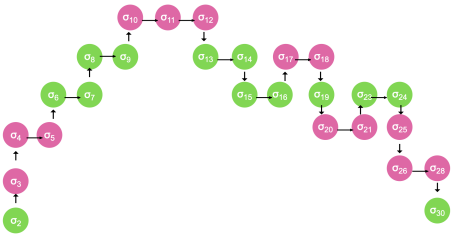
Theorem 9.2. For module M such that $\vdash M$, and for any assertions A, A', A'' and statement $stmt$:

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

9.3 Summarised Execution

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

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



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

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

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

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

9.4 Soundness of the Hoare Quadruples Logic

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

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

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

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

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

10 OUR EXAMPLE PROVEN

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

11 CONCLUSION: SUMMARY, RELATED WORK AND FURTHER WORK

Our motivation comes from the OCAP approach to security, whereby object capabilities guard against un-sanctioned effects. Miller [76, 78] advocates *defensive consistency*: whereby "An object is defensively consistent when it can defend its own invariants and provide correct service to its well behaved clients, despite arbitrary or malicious misbehaviour by its other clients." Defensively consistent modules are hard to design and verify, but make it much easier to make guarantees about systems composed of multiple components [84].

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

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

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

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

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

Lack of Eventual Access Efforts to restrict "eventual access" have been extensively explored, with Ownership Types being a prominent example [20, 25]. These types enforce encapsulation boundaries to safeguard internal implementations, thereby ensuring representation independence and defensive consistency [6, 24, 86]. Ownership is fundamental to key systems like Rust's memory safety [53, 56], Scala's Concurrency [44, 45], Java heap analyses [48, 80, 91], and plays a critical role in program verification [13, 59] including Spec# [8, 9] and universes [31, 32, 65], Borrowable Fractional Ownership [85], and recently integrated into languages like OCAML [64, 69].

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

In future work we want to explore how to express protection within Ownership Types, with the primary challenge being how to accommodate for capabilities accessible to some external objects while still inaccessible to others. Moreover, tightening some rules in our current Hoare logic (e.g. Def. 5.4) may lead to a native Hoare logic of ownership. Also, recent approaches like the Alias Calculus [57, 96], Reachability Types [7, 106] and Capturing Types [12, 17, 109] abstract fine-grained method-level descriptions of references and aliases flowing into and out of methods and fields, and likely accumulate enough information to express protection. Effect exclusion [66] directly prohibits nominated effects, but within a closed, fully-typed world.

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

nested calls, until termination of the currently executing method, but not beyond. We compare with object and history invariants in §4.2.

Such guarantees are maintained by the module as a whole. Drossopoulou et al. [37] proposed "holistic specifications" which take an external perspective across the interface of a module. Mackay et al. [67] builds upon this work, offering a specification language based on *necessary* conditions and temporal operators. Neither of these systems support any kind of external calls. Like [37, 67] we propose "holistic specifications", albeit without temporal logics, and with sufficient conditions. In addition, we introduce protection, and develop a Hoare logic for protection and external calls.

Hoare Logics were first developed in Hoare's seminal 1969 paper [49], and have inspired a plethora of influential further developments and tools. We shall discuss a few only.

Separation logics [51, 94] reason about disjoint memory regions. Incorporating with Separation Logic's powerful framing mechanisms will pose several challenges: We have no specification and no footprint for external calls. Because protection is "scope-aware", expressing it as a predicate would require quantification over all possible paths and variables within the current stack frame. We may also require a new separating conjunction operator. Hyper-Hoare Logics [28, 35] reason about the execution of several programs, and could thus be applied to our problem, if extended to model all possible sequences of calls of internal public methods.

Incorrectness Logic [87] under-approximates postconditions, and thus reasons about the presence of bugs, rather than their absence. Our work, like classical Hoare Logic, over-approximates postconditions, and differs from Hoare and Incorrectness Logics by tolerating interactions between verified code and unverified components. Interestingly, even though earlier work in the space [37, 67] employ *necessary* conditions for effects (*i.e.* under-approximate pre-conditions), we can, instead, employ *sufficient* conditions for the lack of effects (over-approximate postconditions). Incorporating our work into Incorrectness Logic might require under-approximating eventual access, while protection over-approximates it.

Rely-Guarantee [46, 104] and Deny-Guarantee [34] distinguish between assertions guaranteed by a thread, and those a thread can reply upon. Our Hoare quadruples are (roughly) Hoare triples plus the "guarantee" portion of rely-guarantee. When a specification includes a guarantee, that guarantee must be maintained by every "atomic step" in an execution [46], rather than just at method boundaries as in visible states semantics [36, 83, 100]. In concurrent reasoning, this is because shared state may be accessed by another coöperating thread at any time: while in our case, it is because unprotected state may be accessed by an untrusted component within the same thread.

Models and Hoare Logics for the interaction with the the external world Murray [84] made the first attempt to formalise defensive consistency, to tolerate interacting with any untrustworthy object, although without a specification language for describing effects (i.e. when an object is correct).

Cassez et al. [21] propose one approach to reason about external calls. Given that external callbacks are necessarily restricted to the module's public interface, external callsites are replaced with a generated `externalcall()` method that nondeterministically invokes that interface. Rao et al. [93]'s Iris-Wasm is similar. WASM's modules are very loosely coupled: a module has its own byte memory and object table. Iris-Wasm ensures models can only be modified via their explicitly exported interfaces.

Swasey et al. [102] designed OCPL, a logic that separates internal implementations ("high values") from interface objects ("low values"). OCPL supports defensive consistency (called "robust safety" after the security literature [10]) by ensuring low values can never leak high values, a and prove object-capability patterns, such as sealer/unsealer, caretaker, and membrane. RustBelt [53] developed this approach to prove Rust memory safety using Iris [54], and combined with RustHorn [71] for the safe subset, produced RustHornBelt [70] that verifies both safe and unsafe

Rust programs. Similar techniques were extended to C [97]. While these projects verify "safe" and "unsafe" code, the distinction is about memory safety:whereas all our code is "memory safe" but unsafe / untrusted code is unknown to the verifier.

Devriese et al. [30] deploy step-indexing, Kripke worlds, and representing objects as public/private state machines to model problems including the DOM wrapper and a mashup application. Their distinction between public and private transitions is similar to our distinction between internal and external objects. This stream of work has culminated in VMSL, an Iris-based separation logic for virtual machines to assure defensive consistency [63] and Cerise, which uses Iris invariants to support proofs of programs with outgoing calls and callbacks, on capability-safe CPUs [41], via problem-specific proofs in Iris's logic. Our work differs from Swasey, Schaefer's, and Devriese's work in that they are primarily concerned with ensuring defensive consistency, while we focus on module specifications.

Smart Contracts also pose the problem of external calls. Rich-Ethereum [18] relies on Ethereum contracts' fields being instance-private and unaliased. Scilla [99] is a minimalistic functional alternative to Ethereum, which has demonstrated that popular Ethereum contracts avoid common contract errors when using Scilla.

The VerX tool can verify specifications for Solidity contracts automatically [90]. VerX's specification language is based on temporal logic. It is restricted to "effectively call-back free" programs [2, 43], delaying any callbacks until the incoming call to the internal object has finished.

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

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

Anderson et al. [3] extend memory safety arguments to "stack safety": ensuring method calls and returns are well bracketed (aka "structured"), and that the integrity and confidentially of both caller and callee are ensured, by assigning objects to security classes. Schaefer et al. [98] has also adopted an information-flow approach to ensure confidentially by construction.

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

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

DATA AVAILABILITY STATEMENT

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

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

REFERENCES

[1] Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, and Jérémy Thibault. 2019. Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation. arXiv:1807.04603 [cs.PL]

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

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

[4] Cezar-Constantin Andrici, Ştefan Ciobâcă, Catalin Hritcu, Guido Martínez, Exequiel Rivas, Éric Tanter, and Théo Winterhalter. 2024. Securing Verified IO Programs Against Unverified Code in F. *POPL* 8 (2024), 2226–2259.

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

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

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

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

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

[10] Jesper Bengtson, Kathiekeyan Bhargavan, Cedric Fournet, Andrew Gordon, and S.Maffeis. 2011. Refinement Types for Secure Implementations. *TOPLAS* (2011).

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

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

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

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

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

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

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

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

[19] Anton Burtsev, David Johnson, Josh Kunz, Eric Eide, and Jacobus E. van der Merwe. 2017. CapNet: security and least authority in a capability-enabled cloud. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24 - 27, 2017*. 128–141. https://doi.org/10.1145/3127479.3131209

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

[21] Franck Cassez, Joanne Fuller, and Horacio Mijail Anton Quiles. 2024. Deductive verification of smart contracts with Dafny. *Int. J. Softw. Tools Technol. Transf.* 26, 2 (2024), 131–145.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[41] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, and Lars Birkedal. 2024. Cerise: Program Verification on a Capability Machine in the Presence of Untrusted Code. *J. ACM* 71, 1 (2024), 3:1–3:59.

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

[43] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2018. Online Detection of Effectively Callback Free Objects with Applications to Smart Contracts. *POPL* (2018). https://doi.org/10.1145/3158136

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

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

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

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

[48] Trent Hill, James Noble, and John Potter. 2002. Scalable Visualizations of Object-Oriented Systems with Ownership Trees. *J. Vis. Lang. Comput.* 13, 3 (2002), 319–339.

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

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

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

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

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

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

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

2024. Cedar: A New Language for Expressive, Fast, Safe, and Analyzable Authorization. In *oopsla*.
[56] Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language* (2nd ed.).
[57] Alexander Kogtenkov, Bertrand Meyer, and Sergey Velder. 2015. Alias calculus, change calculus and frame inference. *Sci. Comp. Prog.* 97 (2015), 163–172.
[58] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. In *OOPSLA*, Vol. 7. https://doi.org/10.1145/3586037
[59] Dirk Leinenbach and Thomas Santen. 2009. Verifying the Microsoft Hyper-V Hypervisor with VCC. In *Formal Methods*.
[60] K. Rustan M. Leino and Peter Müller. 2004. Object Invariants in Dynamic Contexts. In *ECOOP*.
[61] K. Rustan M. Leino and Wolfram Schulte. 2007. Using History Invariants to Verify Observers. In *ESOP*.
[62] B. Liskov and J. Wing. 1994. A Behavioral Notion of Subtyping. *ACM ToPLAS* 16, 6 (1994), 1811–1841.
[63] Zongyuan Liu, Sergei Stepanenko, Jean Pichon-Pharabod, Amin Timany, Aslan Askarov, and Lars Birkedal. 2023. VMSL: A Separation Logic for Mechanised Robust Safety of Virtual Machines Communicating above FF-A. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1438–1462.
[64] Anton Lorenzen, Stephen Dolan, Richard A. Eisenberg, and Sam Lindley. 2024. Oxidizing OCaml with Modal Memory Management. In *ICFP*.
[65] Y. Lu and J. Potter. 2006. Protecting Representation with Effect Encapsulation.. In *POPL*. 359–371.
[66] Matthew Lutze, Magnus Madsen, Philipp Schuster, and Jonathan Immanuel Brachthäuser. 2023. With or Without You: Programming with Effect Exclusion. *ICFP*, Article 204 (aug 2023), 28 pages. https://doi.org/10.1145/3607846
[67] Julian Mackay, Susan Eisenbach, James Noble, and Sophia Drossopoulou. 2022. Necessity Specifications are Necessary. In *OOPSLA*. ACM.
[68] S. Maffeis, J.C. Mitchell, and A. Taly. 2010. Object Capabilities and Isolation of Untrusted Web Applications. In *Proc of IEEE Security and Privacy*.
[69] Daniel Marshall and Dominic Orchard. 2024. Functional Ownership through Fractional Uniqueness. In *OOPSLA*. https://doi.org/10.1145/3649848
[70] Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. 2022. RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code. In *PLDI*. ACM, 841–856.
[71] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2021. RustHorn: CHC-based Verification for Rust Programs. *TOPLAS* (2021).
[72] Darya Melicher, Yangqingwei Shi, Alex Potanin, and Jonathan Aldrich. 2017. A Capability-Based Module System for Authority Control. In *ECOOP*. 20:1–20:27. https://doi.org/10.4230/LIPIcs.ECOOP.2017.20
[73] Adrian Mettler, David Wagner, and Tyler Close. 2010. Joe-E a Security-Oriented Subset of Java. In *NDSS*.
[74] Bertrand Meyer. 1992. Applying "Design by Contract". *Computer* 25, 10 (1992), 40–51.
[75] B. Meyer. 1992. *Eiffel: The Language*. Prentice Hall.
[76] Mark Samuel Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Ph. D. Dissertation. Baltimore, Maryland.
[77] Mark Samuel Miller. 2011. Secure Distributed Programming with Object-capabilities in JavaScript. (Oct. 2011). Talk at Vrije Universiteit Brussel, mobicrant-talks.eventbrite.com.
[78] Mark Samuel Miller, Tom Van Cutsem, and Bill Tulloh. 2013. Distributed Electronic Rights in JavaScript. In *ESOP*.
[79] Mark Samuel Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. 2008. Safe active content in sanitized JavaScript. code.google.com/p/google-caja/.
[80] Nick Mitchell. 2006. The Runtime Structure of Object Ownership. In *ECOOP*. 74–98.
[81] Scott Moore, Christos Dimoulas, Robert Bruce Findler, Matthew Flatt, and Stephen Chong. 2016. Extensible access control with authorization contracts. In *OOPSLA*, Eelco Visser and Yannis Smaragdakis (Eds.). 214–233.
[82] James H. Morris Jr. 1973. Protection in Programming Languages. *CACM* 16, 1 (1973).
[83] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. 2006. Modular Invariants for Layered Object Structures. *Science of Computer Programming* 62 (2006), 253–286.
[84] Toby Murray. 2010. *Analysing the Security Properties of Object-Capability Patterns*. Ph. D. Dissertation. University of Oxford.
[85] Takashi Nakayama, Yusuke Matsushita, Ken Sakayori, Ryosuke Sato, and Naoki Kobayashi. 2024. Borrowable Fractional Ownership Types for Verification. In *VMCAI*, Vol. LNCS14500. 224–246.
[86] James Noble, John Potter, and Jan Vitek. 1998. Flexible Alias Protection. In *ECOOP*.
[87] Peter W. O'Hearn. 2019. Incorrectness Logic. *Proc. ACM Program. Lang.* 4, POPL, Article 10 (Dec. 2019), 32 pages. https://doi.org/10.1145/3371078
[88] Leo Osvald, Grégory M. Essertel, Xilun Wu, Lilliam I. González Alayón, and Tiark Rompf. 2016. Gentrification gone too far? affordable 2nd-class values for fun and (co-)effect. In *OOPSLA*. 234–251.

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

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

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

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

[93] Xiaojia Rao, Aïna Linn Georges, Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner, and Lars Birkedal. 2023. Iris-Wasm: Robust and Modular Verification of WebAssembly Programs. *PLDI* 7 (2023), 1096–1120.

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

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

[96] Victor Rivera and Bertrand Meyer. 2020. AutoAlias: Automatic Variable-Precision Alias Analysis for Object-Oriented Programs. *SN Comp. Sci.* 1, 1 (2020), 12:1–12:15.

[97] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: automating the foundational verification of C code with refined ownership types. In *PLDI*. ACM, 158–174.

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

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

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

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

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

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

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

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

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

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

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

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

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