Draftable Comparison Export

This document is an exported comparison with limited functionality, generated by Draftable Desktop. To access full functionality, use Draftable's powerful comparison viewer in any of our products.

Left document:	main-scd-31Jan.pdf
Right document:	main.pdf

What is this document?

This is a comparison of two documents. The two documents are interleaved such that the left document is displayed on even pages and the right document is displayed on odd pages.

Is there a specific way I should view this file?

This document is intended to be viewed in Two Page Continuous mode (or sometimes called 'Two Page Scrolling'). It should open in this mode by default when using Adobe Acrobat and most popular PDF readers.

If the document opens in a different view, you can often change this in the settings. In Adobe Acrobat, go to **View > Page Display > Two Page Scrolling**.

Why are there blank pages?

Blank pages are inserted to keep both documents as aligned as much as possible.

How do I read the changes?

Text deleted from the left document and, hence, not in right document is highlighted red. Text added to the right document and, hence, not in left document is highlighted green.

Tip for printing

When printing this document, we recommend printing double-sided and include this first page. This will result in the matching text being displayed on different pages and easily readable, much like a book.

For more information

Draftable offers powerful document comparison solutions for all use-cases. To view our products, please visit our website: draftable.com.

Reasoning about External Calls

ANONYMOUS AUTHOR(S)

In today's complex software, internal, trusted, code is tightly intertwined with external, untrusted, code. Internal code does not trust external code. Nevertheless, it has to call external code, and therefore needs to reason about the potential effects of such external calls.

The effects of external calls can be *limited* if internal code is programmed defensively, so as to ensure certain effects only happen if the external objects have access to the corresponding capabilities.

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

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

1 INTRODUCTION

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

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

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

Precluded, *i.e.* guaranteed to *never happen*. E.g., a correct implementation of the DAO [23] can ensure that the DAO's balance never falls below the sum of the balances of its subsidiary accounts.

Limited, *i.e.* they *may happen*, but *only if* the external code uses certain *object capabilities*. E.g., while the DAO does not preclude that a signatory's balance will decrease, it does ensure that the balance decreases only if the code causing this reduction used the signatory itself.

Precluded effects is a special case of effects limited through capabilities – reasoning about these has been studied in the context of object invariants [8, 38, 65, 90, 108]. Reasoning about effects limited through capabilities is the remit of this paper.

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

Reasoning about External Calls

ANONYMOUS AUTHOR(S)

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

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

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

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

1 Introduction

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

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

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

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

Precluded, *i.e.* guaranteed to *never happen*. E.g., a correct implementation of the DAO [23] can ensure that the DAO's balance never falls below the sum of the balances of its subsidiary accounts, or

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

Precluded effects are special case of limited effects, and have been studied extensively in the context of object invariants [8, 38, 63, 88, 107]. In this paper, we tackle the more general, more difficult, and more subtle case of reasoning about limited effects for external calls.

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

The Object Capability Model. The object-capability model combines the "capabilities" developed for operating system security ([67, 115]) with pure object-oriented programming ([1, 106, 111]). Capability-based operating systems reify effects and resources as *capabilities* — unforgeable, distinct, duplicable, attenuable, communicable bitstrings which both denote a resource and grant rights which can be exercised over that resource. Access control is achieved solely by controlling access to the capabilities.

Mark Miller's [82] *object*-capability model uses object references as capabilities. Building on early object-capability languages such as E [82, 85] and Joe-E [79], a range of recent programming languages and web systems [19, 50, 101] including Newspeak [16], AmbientTalk [30] Dart [15], Grace [11, 56], JavaScript (aka Secure EcmaScript [84]), and Wyvern [78] have adopted the object capability model. Security and encapsulation is encoded in the relationships between the objects, and the interactions between them. As argued in Drossopoulou and Noble [39], object capabilities are a mechanism which makes it possible to write secure programs but cannot guarantee that any particular program using the provided mechanism is, indeed, secure.

Reasoning with Capabilities. Recent work has developed logics to prove properties of programs employing object capabilities. Swasey et al. [109] develop a logic to prove that code employing object capabilities for encapsulation preserves invariants for intertwined code, but without external calls. Devriese et al. [32] can describe and verify invariants about multi-object structures and the availability and exercise of object capabilities. Similarly, Liu et al. [69] propose a separation logic to support formal modular reasoning about communicating VMs, including in the presence of unknown VMs. Rao et al. [98] 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. [40] and Mackay et al. [73] 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, limited effects, and external calls.

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

Structure of this paper. Sect. 2 outlines the main ingredients of our approach in terms of an example. Sect. 3 outlines a simple object-oriented language used for our work. Sect. 4 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 specification. 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, sequential language similar to Joe-E [79] with modules, module-private

The Object Capability Model. The object-capability model combines the capability model of operating system security [65, 114] with pure object-oriented programming [1, 105, 110]. Capability-based operating systems reify resources as *capabilities* — unforgeable, distinct, duplicable, attenuable, communicable bitstrings which both denote a resource and grant rights over that resouce. Effects can only be caused by invoking capabilities: controlling effects reduces to controlling capabilities.

Mark Miller's [80] *object*-capability model treats object references as capabilities. Building on early object-capability languages such as E [80, 83] and Joe-E [77], a range of recent programming languages and web systems [19, 49, 100] including Newspeak [16], AmbientTalk [29] Dart [15], Grace [11, 55], JavaScript (aka Secure EcmaScript [82]), and Wyvern [76] have adopted the object capability model. Security and encapsulation is encoded in the relationships between the objects, and the interactions between them. As argued by Drossopoulou and Noble [39], object capabilities make it possible to write secure programs, but cannot by themselves guarantee that any particular program will be secure.

Reasoning with Capabilities. Recent work has developed logics to prove properties of programs employing object capabilities. Swasey et al. [108] develop a logic to prove that code employing object capabilities for encapsulation preserves invariants for intertwingled code, but without external calls. Devriese et al. [31] describe and verify invariants about multi-object structures and the availability and exercise of object capabilities. Similarly, Liu et al. [67] propose a separation logic to support formal modular reasoning about communicating VMs, including in the presence of unknown VMs. Rao et al. [97] specify WASM modules, and prove that adversarial code can affect other modules only through functions they explicitly export. Cassez et al. [21] model external calls as an unbounded number of invocations to a module's public interface.

The approaches above do not aim to support general reasoning to limit the effects of external calls to untrusted code. Drossopoulou et al. [40] and Mackay et al. [71] begin to tackle external effects; the former proposes "holistic specifications" to describe a module's emergent behaviour. and the latter develops a tailor-made logic to prove that modules which do not contain external calls adhere to holistic specifications. Rather than relying on problem-specific, custom-made proofs, we propose a Hoare logic that addresses access to capabilities, limited effects, and external calls.

This paper contributes. (0) a demonstration that object-capabilities can reify the effects of external calls, (1) *protection assertions* to limit access to object-capabilities, and concomitantly limit effects, (2) a specification language to define how capabilities should limit effects, (3) a Hoare logic to reason about external calls and to prove that modules satisfy their specifications, (4) proof of soundness, (5) a worked illustrative example with a mechanised proof in Coq.

Structure of this paper. Sect. 2 outlines the main ingredients of our approach in terms of an example. Sect. 3 outlines a simple object-oriented language used for our work. Sect. 4 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 specification. 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, sequential language similar to Joe-E [77] with modules, module-private fields (accessible only from methods from the same module), and unforgeable, un-enumerable fields (accessible only from methods from the same module), and unforgeable, un-enumerable addresses. We distinguish between *internal objects* — instances of our internal module M's classes — and *external objects* defined in *any* number of external modules $\overline{M}$[1]. States whose receiver (`this`) is internal are *internal states* – they are executing code from the internal module – the other states are *external states.* `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. [2]

We are concerned with guarantees made in an *open* setting; Our internal module M must be programmed so that execution of M together with *any* unknown, arbitrary, external modules $\overline{M}$ will satisfy these guarantees – without relying on any assumptions about $\overline{M}$'s code (beyond the programming language's semantics) [3].

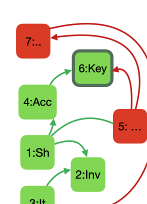
`Shop` – illustrating limited effects

Consider the following, internal, module M_{shop}, containing classes `Item`, `Shop`, `Account`, and `Inventory`. Classes `Inventory` and `Item` have the expected functionality. `Account`s hold a balance and have a key. Access to an `Account`, allows one to pay money into it, and access to an `Account` and its `Key`, allows one to withdraw money from it. `Shop` has a public method `buy` whose formal parameter `buyer` is an `external` object.

```
module Mshop
  ...
  class Shop
    field accnt:Account, invntry:Inventory, clients:external
    public method buy(buyer:external, anItem:Item)
      int price = anItem.price
      int oldBlnce = this.accnt.blnce
      buyer.pay(this.accnt, price)
      if (this.accnt.blnce == oldBlnce+price)
         this.send(buyer,anItem)
      else
         buyer.tell("you have not paid me")
    private method send(buyer:external, anItem:Item)
       ...
```

The sketch below shows a possible heap snippet. Red and green rounded rectangles indicate external and internal objects, respectively.

Each object has a number, followed by an abbreviated class name. Here, o_1, o_2 and o_5 are a `Shop`, an `Inventory`, and an external object. Curved arrows indicate field values. Here, o_1 has three fields, pointing to o_4, o_5 and o_2. Fields denote direct access. The transitive closure of direct access gives indirect (or transitive) access. Here, o_1 has direct access to o_4, and indirect access to o_6.
Object o_6 is the capability that allows withdrawal from o_4. We highlight this through a dark outline to o_6.



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

[2]As in Joe-E, we leverage module-based privacy to restrict propagation of capabilities, and reduce the need for reference monitors etc, *c.f.* Sect 3 in [79].

[3]This is a critical distinction from e.g. cooperative approaches such as rely/guarantee [49, 112].

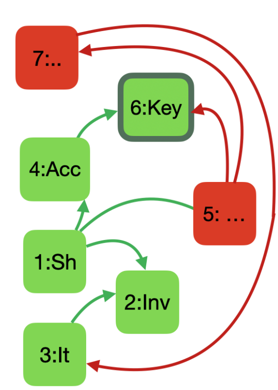
addresses. We distinguish between *internal objects* — instances of our internal module M's classes — and *external objects* defined in *any* number of external modules $\overline{M}$.[1] Program states whose receiver (`this`) is internal are *internal states* – they are executing code from the internal module – the other states are *external states*. `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.[2]

We are concerned with guarantees made in an *open* setting; Our internal module M must be programmed so that execution of M together with *any* unknown, arbitrary, external modules $\overline{M}$ will satisfy these guarantees – without relying on any assumptions about $\overline{M}$'s code beyond the programming language's semantics.[3]

`Shop` – illustrating limited effects

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

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



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

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

[1] We use the notation $\overline{z}$ for a sequence of z, *i.e.* for $z_1, z_2, ...z_n$
[2] As in Joe-E, we leverage module-based privacy to restrict propagation of capabilities, and reduce the need for reference monitors etc, *c.f.* Sect 3 in [77].
[3] This is a critical distinction from e.g. cooperative approaches such as rely/guarantee [48, 111].

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

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

Can `Shop` be certain? Indeed, if

(A) Prior to the call of `buy`, the `buyer` has no eventual access to the account's key, —- and
(B) M_{shop} ensures that
 (a) access to keys is not leaked to external objects, —- and
 (b) funds cannot be withdrawn unless the external entity responsible for the withdrawal 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. This gives rise to the following three challenges: 1^{st}: A specification language which describes access to capabilities and limited effects, 2^{nd}: A Hoare Logic for adherence to such specifications.

2.1 1^{st} Challenge: Specification Language

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

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

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

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

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

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

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

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

Thus:

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

We formalize the concepts "reachable from the current point of execution" and "future scoped by the current point of execution" through the concept of *protection*, and *scoped invariants*. We discuss

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

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

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

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

2.1 1^{st} Challenge: Specification Language

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

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

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

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

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

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

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

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

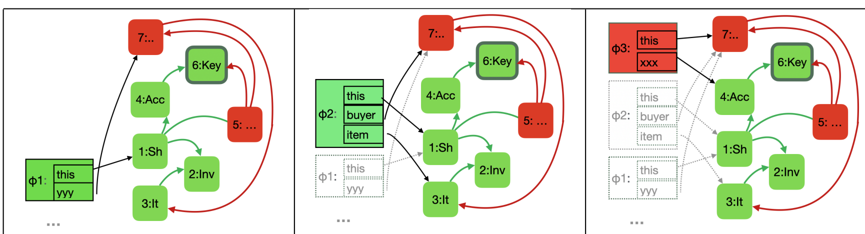
Thus:

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

We will shortly formalize "reachable from the current point of execution" as *protection* in §2.1.1, and then "future scoped by the current point of execution" as *scoped invariants* in §2.1.2. Both of these defintions are in terms of the "current point of execution":
The Current Point of Execution is characterized by the heap, and the activation frame of the currently executing method. Activation frames (frames for short) consist of a variable map and a continuation – the statements remaining to be executed in that method. Method calls push frames onto the call stack; method returns pop frames off. The frame on top of the stack (the most recently pushed frame) belongs to the currently executing method.

these concepts in more detail in §2.1.1, and §2.1.2. But first, we clarify the concept of "current point of execution".

The Current Point of Execution. is characterized by the heap, and the activation frame of the currently executing method. Activation frames, or frames for short, consist of a variable map and a continuation – the statements remaining to be executed. Upon method call and return, frames are pushed onto/popped from the call stack. Thus, the frame on top of the stack is the one most recently pushed; it corresponds to the currently executing method.



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

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

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

2.1.1 *Protection.*

Protection Object o is *protected from* o', formally $\langle\!\langle o \rangle\!\rangle \leftrightarrow\!\!\!\!\!\times\ o'$, if no external object transitively accessible from o' has direct access to o. Object o is *protected*, formally $\langle\!\langle o \rangle\!\rangle$, if no external object transitively accessible from the current frame[4] has direct access to o, and if the receiver is external then o is not an argument. More in Def. 5.4.

Fig. 2 illustrates these concepts. In particular, o_6 is not protected in σ_1 nor σ_2, but is protected in σ_3. This is so, because pushing a frame makes fewer objects transitively accessible from the new top frame – here, o_5 is transitively accessible from the top frame in σ_2, but not transitively accessible from the top frame in σ_3. Moreover, o_4 is protected in σ_1 and σ_2, and not protected in σ_3. This is so, because even though neither o_5 nor o_7 have direct access to o_4, in σ_3 the receiver is external, and o_4 is one of the arguments.

If the internal module never passes o to external objects (*i.e.* never leaks o) and o is protected, then o will remain protected during execution of the current method and all the methods *it* calls. However, it need not be protected during execution of the method calling the current method, nor after termination of the current method. This discussion leads us to scoped invariants.

2.1.2 *Scoped Invariants.*

We build on the concept of history invariants [26, 66, 68] and define:

Scoped invariants $\forall \overline{x : C}.\{A\}$ expresses that if an external state σ has objects $\overline{x}$ of class $\overline{C}$, and satisfies A, then all σ's external, *scoped future states* will also satisfy A. The scoped future

[4] An object is transitively accessible from a frame if there exists a sequence of field accesses leading from one of the variables in the frame to that object.

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

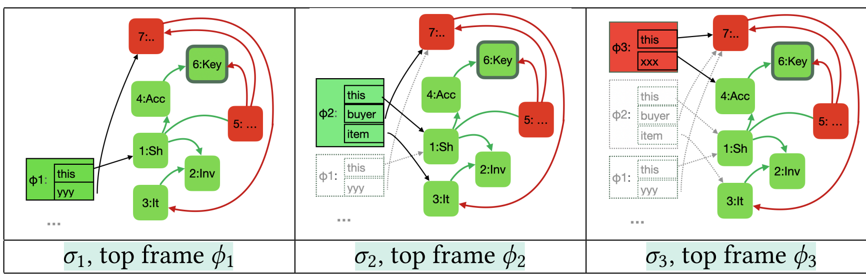
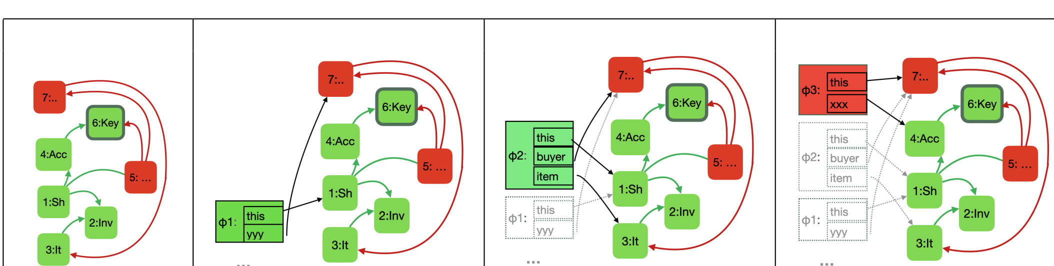


Fig. 1. *The current point of execution* before `buy`, during `buy`, and during `pay`. Frames ϕ_1, ϕ_2 are green as their receiver (`this`) is internal; ϕ_3 is red as its receiver is external. Continuations are omitted.

2.1.1 *Protection.*

Protection Object o is *protected from* o', formally $\langle\!\langle o \rangle\!\rangle \not\leftrightarrow o'$, if no external object indirectly accessible from o' has direct access to o. Object o is *protected*, formally $\langle\!\langle o \rangle\!\rangle$, if no external object indirectly accessible from the current frame[4] has direct access to o, and if the receiver is external then o is not an argument. More in Def. 5.4.

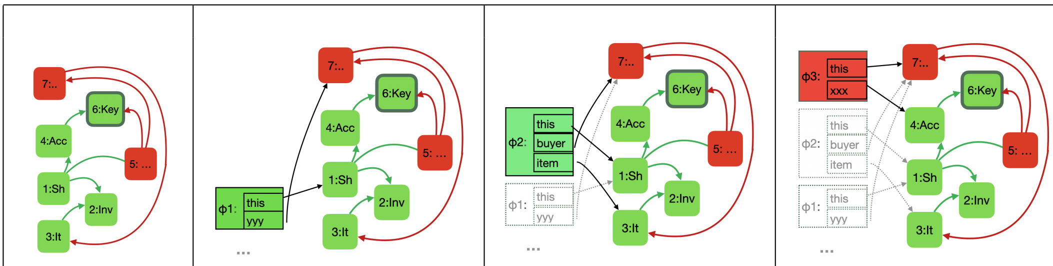
Fig. 2 illustrates *protected* and *protected from*. Object o_6, it is not protected in states σ_1 and σ_2, but o_6 is protected in state σ_3. Object o_4 is protected in states σ_1 and σ_2, and not protected in state σ_3 (because though neither object o_5 nor o_7 have direct access to o_4, in state σ_3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



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

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

[4] An object is indirectly accessible from a frame if there exists a sequence of field accesses (direct references) leading from one of the variables in the frame to that object.



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

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

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 [5] – *c.f.* Def 4.2. Scoped invariants only consider external states – *c.f.* Def 7.4.

Fig. 3 shows the states of some, unspecified, execution, starting at internal state σ_4 and terminating at internal state σ_{23}. It distinguishes between steps within the same method ($\longrightarrow$), method call ($\uparrow$), and method return ($\downarrow$). The scoped future of σ_6 consists of σ_6-σ_{21}. The scoped future of σ_9 consists of σ_9, σ_{10}, σ_{11}, σ_{12}, σ_{13}, and σ_{14}, and does not include, *e.g.*, σ_{15}, or σ_{19}.

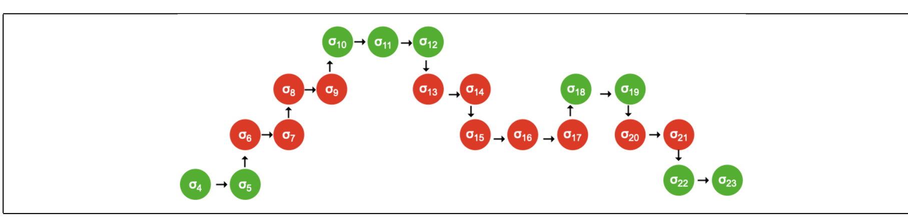


Fig. 3. *Execution.* Green resp. red disks represent internal resp. external states.

The scoped invariant $\forall \overline{x:C}.\{A_0\}$ guarantees that if A_0 holds in σ_8, then it will also hold in σ_9, σ_{13}, and σ_{14}; it doesn't have to hold in σ_{10}, σ_{11}, and σ_{12} as these are internal states. Similarly, it guarantees that if A_0 holds at σ_6, then it will also hold at σ_7, σ_8, σ_9, σ_{13}, σ_{14}, σ_{15}, σ_{16}, σ_{17}, σ_{20} and σ_{21}; it may or may not hold at σ_{10}, σ_{11}, σ_{12}, σ_{18}, σ_{19}, as these are internal states.

2.1.3 Examples.

Example 2.1. The following scoped invariants

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

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

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

This example illustrates three crucial properties of our invariants. Namely, they are

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

If a protected object o is never passed to external objects (*i.e.* never leaked) then o will remain protected during the whole execution of the current method, including during any nested calls. This is the case even if o was not protected before the execution of the current method, during the remainder of the execution of the method calling the current method, nor after termination of the current method. We express these call-stack bounds with *scoped invariants*.

2.1.2 Scoped Invariants. We build on the concept of history invariants [26, 64, 66] and define:

Scoped invariants $\mathbb{\forall}\overline{x:C}.\{A\}$ expresses that if an external state σ has objects $\overline{x}$ of class $\overline{C}$, and satisfies A, then all σ's external, *scoped future states* will also satisfy A. The scoped future contains all states which can be reached through any program execution steps, including further method calls and returns, but stopping just before returning from the call active in σ [5] – *c.f.* Def 4.2. Scoped invariants only consider external states – *c.f.* Def 7.4.

Fig. 3 shows the states of an unspecified execution starting at internal state σ_4 and terminating at internal state σ_{23}. Fig. 3 distinguishes between steps within the same method (⟶), method calls (↑), and method returns (↓). The scoped future of σ_6 consists of σ_6-σ_{21}. The scoped future of σ_9 consists of σ_9, σ_{10}, σ_{11}, σ_{12}, σ_{13}, and σ_{14}, and does not include, *e.g.*, σ_{15}, or σ_{19}.

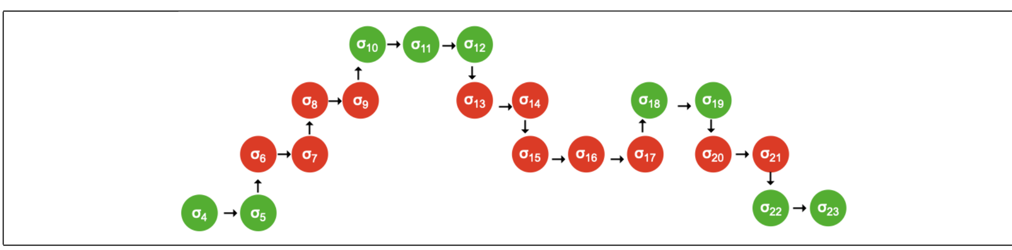


Fig. 3. *Execution.* Green disks represent internal states; red disks external states.

The scoped invariant $\mathbb{\forall}\overline{x:C}.\{A_0\}$ guarantees that if A_0 holds in σ_8, then it will also hold in σ_9, σ_{13}, and σ_{14}; it doesn't have to hold in σ_{10}, σ_{11}, and σ_{12} as these are internal states. Similarly, it guarantees that if A_0 holds at σ_6, then it will also hold at σ_7, σ_8, σ_9, σ_{13}, σ_{14}, σ_{15}, σ_{16}, σ_{17}, σ_{20} and σ_{21}; it may or may not hold at σ_{10}, σ_{11}, σ_{12}, σ_{18}, σ_{19}, as these are internal states.

2.1.3 Examples.

Example 2.1. The following scoped invariants

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

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

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

This example illustrates three crucial properties of our invariants:

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

Scoped: Invariants are preserved during execution of a specific method but not beyond its return. It is, in fact, expected that the invariant will eventually cease to hold after its completion. For instance, while $\langle\!\langle a.\texttt{key} \rangle\!\rangle$ may currently hold, it is possible that an earlier (thus quiescent)

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

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

Scoped: They are preserved during execution of a specific method but not beyond its return. It is, in fact, expected that the invariant will eventually cease to hold after its completion. For instance, while $\langle a.\texttt{key}\rangle$ may currently hold, it is possible that some external object, accessible from a deeper frame in the current call stack, has direct access to $a.\texttt{key}$ – without such access, a would not be usable for payments. Consequently, once enough frames are popped from the stack, $\langle a.\texttt{key}\rangle$ will no longer hold.

API-independent and Module-wide: They describe externally observable effects (*e.g.* `key` stays protected), rather than individual methods (*e.g.* `set`). Thus, they characterize any module with accounts which have a `blnce` and `key` – even as ghost fields – irrespective of its API.

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

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

S_4 guarantees that if the key was protected from `buyer` before the call, then the balance will not decrease. [6] It does *not* guarantee `buy` will only be called when $\langle \texttt{this.accnt.key}\rangle \not\leftrightarrow \texttt{buyer}$ holds: as a public method, `buy` can be invoked by external code that ignores all specifications.

Example 2.3. We illustrate the meaning of our specifications using three versions of a class `Account` from [73] as part of our internal module M_{shop}. To differentiate, we rename M_{shop} as $\texttt{M}_{good}$, $\texttt{M}_{bad}$, or $\texttt{M}_{fine}$. All use the same `transfer` method for withdrawing money.

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

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

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

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

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

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

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

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

method invocation frame has direct access to a.key – without such access, a would not be usable for payments. Once control flow returns to the quiescent method (*i.e.* enough frames are popped from the stack) $\langle a.\texttt{key}\rangle$ will no longer hold.

Modular: Invariants describe externally observable effects (*e.g.* key stays protected) across whole modules, rather than the individual methods (*e.g.* set) making up a module's interface. Our example specifications will characterize *any* module defining accounts with a blnce and a key – even as ghost fields – irrespective of their APIs.

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

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

S_4 guarantees that if the key was protected from buyer before the call, then the balance will not decrease[6]. It does *not* guarantee buy will only be called when $\langle \texttt{this.accnt.key}\rangle \not\leftrightarrow \texttt{buyer}$ holds: as a public method, buy can be invoked by external code that ignores all specifications.

Example 2.3. We illustrate the meaning of our specifications using three versions ($\texttt{M}_{good}$, $\texttt{M}_{bad}$, and $\texttt{M}_{fine}$) of the M_{shop} module [71]; these all share the same transfer method to withdraw money:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

promising that if a state satisfies A and executes $stmt$, any terminating state will satisfy A', and and any intermediate external states reachable during execution of $stmt$ satisfy A'' – *c.f.* Def. 7.2.

To develop our logic, we assume an underlying Hoare logic of triples, $M \vdash_{ul} \{\ A\ \}\ stmt\ \{\ A'\ \}$, which does not have the concept of protection, nor does it deal with external calls. We extend this logic through substructural rules, rules about protection, an embedding into our quadruples, and rules about external calls *c.f.* Figs. 6 - 7. For example, any newly created object is protected. And any valid triple in the underlying Hoare logic is a valid quadruple in our logic, provided that no method is called in *stmt*.

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

Well-formed modules. 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. *E.g.*, to prove that `Shop::buy` satisfies S_3, taking $stmts_b$ for the body of `buy`, we have to prove:

$$\begin{array}{l} \{\ A_0 \wedge \langle\!\langle \mathtt{a.key} \rangle\!\rangle \wedge \mathtt{a.blnce} \geq \mathtt{b}\ \} \\ \qquad stmts_b \\ \{\ \langle\!\langle \mathtt{a.key} \rangle\!\rangle \wedge \mathtt{a.blnce} \geq \mathtt{b}\ \}\ \|\ \{\ \langle\!\langle \mathtt{a.key} \rangle\!\rangle \wedge \mathtt{a.blnce} \geq \mathtt{b}\ \} \end{array}$$

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

External Calls. Consider the verification of S_4. The challenge is how to reason about the external call on line 8 (from `buy` in `Shop`). We need to establish the Hoare quadruple:

$$\begin{array}{ll} & \{\ \mathtt{buyer : extl}\ \wedge\ \langle\!\langle \mathtt{this.accnt.key} \rangle\!\rangle \not\leftrightarrow \mathtt{buyer}\ \wedge\ \mathtt{this.accnt.blnce} = b\ \} \\ \textbf{(1)} & \qquad \mathtt{buyer.pay(this.accnt,price)} \\ & \{\ \mathtt{this.accnt.blnce} \geq b\ \}\ \|\ \{\ \langle\!\langle \mathtt{a.key} \rangle\!\rangle \wedge \mathtt{a.blnce} \geq \mathtt{b}\ \} \end{array}$$

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

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

Fig. 2 provides insights into addressing this issue. Upon entering the call, in state σ_3, we find that $\sigma_3 \models \langle\!\langle o_4.\mathtt{key} \rangle\!\rangle$. More generally, if $\langle\!\langle$`this.accnt.key`$\rangle\!\rangle \not\leftrightarrow$ `buyer` holds before the call to `pay`, then $\langle\!\langle$`this.accnt.key`$\rangle\!\rangle$ holds upon entering the call. This is because any objects accessible during `pay` are accessible from the call's arguments (*i.e.* `buyer`, `this.accnt`, and `price`).

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

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

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

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

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

promising that if a state satisfies A and executes $stmt$, any terminating state will satisfy A', and and any intermediate external states reachable during execution of $stmt$ satisfy A'' – *c.f.* Def. 7.2.

To develop our logic, we assume an underlying Hoare logic of triples, $M \vdash_{ul} \{\ A\ \}\ stmt\ \{\ A'\ \}$, which does not have the concept of protection, nor does it deal with external calls. We extend this logic through substructural rules, rules about protection, an embedding into our quadruples, and rules about external calls *c.f.* Figs. 6 - 7. For example, any newly created object is protected. Any valid triple in the underlying Hoare logic is a valid quadruple in our logic, provided that no method is called in $stmt$.

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

Well-formed modules. 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. *E.g.,* to prove that `Shop::buy` satisfies S_3, taking $stmts_b$ for the body of `buy`, we have to prove:

$$\begin{array}{l} \{\ A_0 \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \wedge \texttt{a.blnce}{\geq}\texttt{b}\ \} \\ \qquad stmts_b \\ \{\ \langle\!\langle \texttt{a.key} \rangle\!\rangle \wedge \texttt{a.blnce}{\geq}\texttt{b}\ \}\ \|\ \{\ \langle\!\langle \texttt{a.key} \rangle\!\rangle \wedge \texttt{a.blnce}{\geq}\texttt{b}\ \} \end{array}$$

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

External Calls. Consider the verification of S_4. The challenge is how to reason about the external call on line 8 (from `buy` in `Shop`). We need to establish the Hoare quadruple:

$$\begin{array}{ll} & \{\ \texttt{buyer:extl}\ \wedge\ \langle\!\langle \texttt{this.accnt.key} \rangle\!\rangle \not\leftrightarrow \texttt{buyer}\ \wedge\ \texttt{this.accnt.blnce} = b\ \} \\ \textbf{(1)} & \qquad \texttt{buyer.pay(this.accnt,price)} \\ & \{\ \ \texttt{this.accnt.blnce} \geq b\ \ \}\ \|\ \{\ \langle\!\langle \texttt{a.key} \rangle\!\rangle \wedge \texttt{a.blnce}{\geq}\texttt{b}\ \} \end{array}$$

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

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

Fig. 2 provides insights into addressing this issue. Upon entering the call, in state σ_3, we find that $\sigma_3 \models \langle\!\langle o_4.\texttt{key} \rangle\!\rangle$. More generally, if $\langle\!\langle$`this.accnt.key`$\rangle\!\rangle \not\leftrightarrow$ `buyer` holds before the call to `pay`, then $\langle\!\langle$`this.accnt.key`$\rangle\!\rangle$ holds upon entering the call. This is because any objects indirectly accessible during `pay` must have been indirectly accessible from the call's receiver (`buyer`) or arguments (`this.accnt` and `price`) when `pay` was called — calls cannot increase the set of indirectly accessible objects (§2.1.1).

In general, for a call $\texttt{y}_0.m(\texttt{y}_1,...,\texttt{y}_n)$, if at the call $\langle\!\langle \texttt{x} \rangle\!\rangle \not\leftrightarrow \texttt{y}_i$ for all $\texttt{y}_i$, then $\langle\!\langle \texttt{x} \rangle\!\rangle$ upon entering the call. To formulate this, we introduce the adaptation operator $-\!\nabla$, which translates assertions from the caller's perspective to that of the callee. Specifically, $A{-\!\nabla}(\texttt{y}_0,...,\texttt{y}_n)$ at a call, ensures A

In general, for a call $\mathtt{y}_0.m(\mathtt{y}_1, ..., \mathtt{y}_n)$, if at the call $\langle\!\langle \mathtt{x} \rangle\!\rangle \leftrightarrow\!\!\!\!\times\ \mathtt{y}_i$ for all $\mathtt{y}_i$, then $\langle\!\langle \mathtt{x} \rangle\!\rangle$ upon entering the call. To formulate this, we introduce the adaptation operator $-\!\nabla$, which translates assertions from the caller's perspective to that of the callee. Specifically, $A -\!\nabla(\mathtt{y}_0, ..., \mathtt{y}_n)$ at a call, ensures A when the variables $\mathtt{y}_0, ..., \mathtt{y}_n$ are pushed onto a new frame. More in Def 8.2 and Lemma 8.3. Here,

$$\langle\!\langle \mathtt{this.accnt.key} \rangle\!\rangle -\!\nabla(\mathtt{buyer}, \mathtt{this.accnt}, \mathtt{price}) = \langle\!\langle \mathtt{this.accnt.key} \rangle\!\rangle \leftrightarrow\!\!\!\!\times\ \mathtt{buyer}$$

This enables the application of S_3 in **(1)**. The corresponding Hoare logic rule is shown in Fig. 7.

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. The external code may be written in the same or a different programming language than the internal code – all we need is that the platform protects direct external read/write of the internal private fields, while allowing indirect manipulation through calls of public methods. Our specifications are conditional: while they do not guarantee that specific effects will never occur, they ensure that the effects will only happen if specific conditions were met.

The conditional and scoped nature of our invariants might prompt questions about their usefulness. Indeed, scoped invariants are not meant to guarantee that effects will never happen. While scoped invariants do not ensure that certain effects will never occur, they do guarantee that these effects can only take place in contexts where specified conditions are satisfied. For instance, while $a.\mathtt{blnce}$ may decrease in the future, this will only happen in contexts where an external object has direct access to $a.\mathtt{key}$. Enforcing such conditions is the responsibility of the internal module.

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

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

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

This work is based on $\mathscr{L}_{ul}$, a minimal, imperative, sequential, class based, typed, object-oriented language. We believe, however, that the work can easily be adapted to any capability safe language with some form of encapsulation. Wrt to encapsulation and capability safety, $\mathscr{L}_{ul}$ supports private fields, private and public methods, unforgeable addresses, and no ambient authority (no static methods, no address manipulation). To reduce the complexity of our formal models, as is usually done, CITE - CITE, $\mathscr{L}_{ul}$ lacks many common languages features, omitting static fields and methods, interfaces, inheritance, subsumption, exceptions, and control flow. In our examples, we use numbers and booleans – these can be encoded.

The syntax of $\mathscr{L}_{ul}$ is given in Fig. 4. It includes syntax for ghost expressions that may be used in writing specifications. The syntax does not distinguish between fields and ghost fields: f stands for a field, or a ghost field, but not a method – *i.e.* no side-effects.[7] A $\mathscr{L}_{ul}$ state, σ, consists of a heap χ and a stack. A stack is a sequence of frames, $\phi_1 \cdot ... \cdot \phi_n$. A frame, ϕ, consists of a local variable map and a continuation, *i.e.* a sequence of statements to be executed. The top frame, *i.e.* the frame most recently pushed onto the stack, in a state $(\phi_1 \cdot ... \cdot \phi_n, \chi)$ is ϕ_n.

Notation. We adopt the following unsurprising notation:

- An object is uniquely identified by the address that points to it. We shall be talking of objects o, o' when talking less formally, and of addresses, α, α', α_1, ... when more formal.

[7] *E.g.*, $\mathtt{a.blnce}$ may, in some modules (*e.g.* in $\mathtt{M}_{good}$), 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.

when the variables $y_0, ..., y_n$ are pushed onto a new frame. More in Def 8.2 and Lemma 8.3. Here,
$\langle$`this.accnt.key`$\rangle$ $\neg\nabla$ (`buyer,this.accnt,price`) = $\langle$`this.accnt.key`$\rangle$ $\not\leftrightarrow$ `buyer`
This enables the application of S_3 in **(1)**. The corresponding Hoare logic rule is shown in Fig. 7.

Summary

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

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

3 The underlying programming language $\mathscr{L}_{ul}$

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

This work is based on $\mathscr{L}_{ul}$, a minimal, imperative, sequential, class based, typed, object-oriented language. We believe, however, that the work can easily be adapted to any capability safe language with some form of encapsulation. Wrt to encapsulation and capability safety, $\mathscr{L}_{ul}$ supports private fields, private and public methods, unforgeable addresses, and no ambient authority (no static methods, no address manipulation). To reduce the complexity of our formal models, as is usually done, *e.g.* [32, 53, 94], $\mathscr{L}_{ul}$ lacks many common languages features, omitting static fields and methods, interfaces, inheritance, subsumption, exceptions, and control flow. In our examples, we use numbers and booleans – these can be encoded.

Fig. 4 shows the syntax of $\mathscr{L}_{ul}$. Statements, $stmt$, are three-address instructions, or method calls, or empty, ϵ. Expressions, e, may appear in assertions, but not in statements. They may contain fields, $e.f$, or ghost fields, $e_0.gf(\overline{e})$, and so have no side-effects[7]. A $\mathscr{L}_{ul}$ state, σ, consists of a heap χ and a stack. A stack is a sequence of frames, $\phi_1 \cdot ... \cdot \phi_n$. A frame, ϕ, consists of a local variable map and a continuation, *i.e.* the statements to be executed. The top frame, *i.e.* the frame most recently pushed onto the stack, in a state $(\phi_1 \cdot ... \cdot \phi_n, \chi)$ is ϕ_n.

Notation. We adopt the following unsurprising notation:

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

[7] For convenience, $e.gf$ is short for $e.gf()$. Thus, expressions like $x_1.f_1$ are field lookups in some modules, and ghostfields in others. *E.g.*, `a.blnce`, is a field lookup in M_{good}, and a ghostfield in a module which stores balance in a table.

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

$$\begin{array}{lll}
stmt & ::= & x := y \mid x := v \mid x := y.f \mid x.f := y \mid x := y_0.m(\overline{y}) \mid \texttt{new}\ C \mid stmt;\ stmt \mid \epsilon \quad \textbf{Statement}\\
gfld & ::= & \texttt{ghost}\ f(\overline{x:T})\{\ e\ \} : T \quad \textbf{Ghost Field Def.}\\
e & ::= & x \mid v \mid e.f \mid e_0.f(\overline{e}) \quad \textbf{Ghost Expression}
\end{array}$$

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

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

- x, x', y, z, u, v, w are variables.
- $dom(\phi)$ and $Rng(\phi)$ indicate the variable map in ϕ; $dom(\sigma)$ and $Rng(\sigma)$ indicate the variable map in the top frame in σ
- $\alpha \in \sigma$ means that α is defined in the heap of σ, and $x \in \sigma$ means that $x \in dom(\sigma)$. Conversely, $\alpha \notin \sigma$ and $x \notin \sigma$ have the obvious meanings. $\lfloor \alpha \rfloor_\sigma$ is α; and $\lfloor x \rfloor_\sigma$ is the value to which x is mapped in the top-most frame of σ's stack, and $\lfloor e.f \rfloor_\sigma$ looks up in σ's heap the value of f for the object $\lfloor e \rfloor_\sigma$.
- $\phi[x \mapsto \alpha]$ 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.
- $text_1 \stackrel{\text{txt}}{=} text_2$ expresses that $text_1$ and $text_2$ are the same text.
- We define the depth of a stack as $|\phi_1 ... \phi_n| \triangleq n$. For states, $|(\overline{\phi}, \chi)| \triangleq |\overline{\phi}|$. The operator $\sigma[k]$ truncates the stack up to the k-th frame: $(\phi_1 ... \phi_k ... \phi_n, \chi)[k] \triangleq (\phi_1 ... \phi_k, \chi)$
- $Vs(stmt)$ returns the variables which appear in $stmt$. For example, $Vs(u := y.f)$=$\{u, y\}$.

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

Fig. 9 describes $\mathscr{L}_{ul}$ execution by a small steps operational semantics with shape $\overline{M}; \sigma \dashrightarrow \sigma'$. $\overline{M}$ stands for one or more modules, where a module, M, maps class names to class definitions. The functions *classOf*(σ, x), *Meth*$(\overline{M}, C, m)$, *SameModule*$(x, y, \sigma, \overline{M})$, and *Prms*$(\sigma, \overline{M})$, return the class of x, the method m for class C, whether x and y belong to the same module, and the formal parameters of the method currently executing in σ – *c.f.* Defs A.2, A.4, A.5, and A.6. Initial states, *Initial*(σ), contain a single frame with single variable `this` pointing to a single object in the heap of class `Object`, and a continuation, *c.f.* A.7.

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

Private methods may be called only if the class of the callee (the object whose method is being called – here y_0), comes from the same module as the class of the caller here `this`). Public methods

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

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

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

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

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.
- $text_1 \stackrel{\text{txt}}{=} text_2$ expresses that $text_1$ and $text_2$ are the same text.
- We define the depth of a stack as $|\phi_1 ... \phi_n| \triangleq n$. For states, $|(\overline{\phi}, \chi)| \triangleq |\overline{\phi}|$. The operator $\sigma[k]$ truncates the stack up to the k-th frame: $(\phi_1 ... \phi_k ... \phi_n, \chi)[k] \triangleq (\phi_1 ... \phi_k, \chi)$
- $Vs(stmt)$ returns the variables which appear in $stmt$. For example, $Vs(u := y.f)$=$\{u, y\}$.

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

Fig. 9 describes $\mathscr{L}_{ul}$ execution by a small steps operational semantics with shape $\overline{M}; \sigma \dashrightarrow \sigma'$. $\overline{M}$ stands for one or more modules, where a module, M, maps class names to class definitions. The functions $classOf(\sigma, x)$, $Meth(\overline{M}, C, m)$, $SameModule(x, y, \sigma, \overline{M})$, and $Prms(\sigma, \overline{M})$, return the class of x, the method m for class C, whether x and y belong to the same module, and the formal parameters of the method currently executing in σ – *c.f.* Defs A.2, A.4, A.5, and A.6. Initial states, $Initial(\sigma)$, contain a single frame with single variable `this` pointing to a single object in the heap of class `Object`, and a continuation, *c.f.* A.7.

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

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

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

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

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

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

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

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

may always be called. When a method is called, a new frame is pushed onto the stack; this frame maps `this` and the formal parameters to the values for the receiver and other arguments, and the continuation to the body of the method. Method bodies are expected to store their return values in the implicitly defined variable `res`. [8] When the continuation is empty (ϵ), the frame is popped and the value of `res` from the popped frame is stored in the variable map of the top frame.

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

4 FUNDAMENTAL CONCEPTS

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

Method calls and returns are critical for our work. They are characterized through pushing/popping frames on the stack: $\sigma \triangledown \overline{\phi}$ pushes frame ϕ onto the stack of σ, while $\sigma \vartriangle$ pops the top 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 $\phi_n(\texttt{cont}) \stackrel{\text{txt}}{=} x := y_0.m(\overline{y}); stmt$

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

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

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

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

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

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

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

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

the implicitly defined variable `res`[8]. When the continuation is empty (ϵ), the frame is popped and the value of `res` from the popped frame is stored in the variable map of the top frame.

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

4 Fundamental Concepts

The semantics of our assertion language is based on three concepts built on $\mathscr{L}_{ul}$: method calls and returns, scoped execution, and (in)directly accessible objects.

Method calls and returns are critical for our work. They are characterized through pushing/popping frames on the stack: $\sigma \triangledown \overline{\phi}$ pushes frame ϕ onto the stack of σ, while $\sigma\ \triangle$ pops the top 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

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

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

4.1 Scoped Execution

In order to give semantics to scoped invariants (introduced in §2.1.2 and to be fully defined in Def. 7.4), we need a new definition of execution, called *scoped execution*.

Definition 4.2 (Scoped Execution)**.** :

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

4.1 Scoped Execution

In order to give semantics to scoped invariants (introduced in §2.1.2 and to be fully defined in Def. 7.4), we need a new definition of execution, called *scoped execution*.

Definition 4.2 (Scoped Execution). :

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

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

The relation $\rightsquigarrow^*$ contains more than the transitive closure of $\rightsquigarrow$. *E.g.,*, $\overline{M};\ \sigma_9 \rightsquigarrow^* \sigma_{13}$, even though $\overline{M};\ \sigma_9 \rightsquigarrow \sigma_{12}$ and $\overline{M};\ \sigma_{12} \not\rightsquigarrow^* \sigma_{13}$. Lemma 4.3 says that the value of the parameters does not change during execution of the same method. Appendix B discusses proofs, and further properties.

Lemma 4.3. For all $\overline{M}, \sigma, \sigma'$: $\quad \overline{M};\ \sigma \rightsquigarrow^* \sigma' \ \wedge\ |\sigma| = |\sigma'| \quad \Longrightarrow \quad \forall x \in Prms(\overline{M}, \sigma).[\ \lfloor x \rfloor_\sigma = \lfloor x \rfloor_{\sigma'}\ ]$

4.2 Reachable Objects, Locally Reachable Objects, and Well-formed States

A central concept to our work is protection, that no locally reachable external object can have direct access to that object. We define it formally in Sect. 5.2 An object α is *locally reachable*, i.e. $\alpha \in LocRchbl(\sigma)$, if it is reachable from the top frame on the stack of σ.

Definition 4.4. We define

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

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

Definition 4.5 (Well-formed states). For modules $\overline{M}$, and states σ, σ':

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

Lemma 4.6 says that (1) execution preserves well-formedness, and (2) any object which is locally reachable after pushing a frame was locally reachable before pushing that frame.

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

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

5 ASSERTIONS

Our assertions are standard or *object-capability*. Standard assertions assert properties of the values of fields, implication, quantification etc, as well as ghost fields which represent user-defined predicates. The object capability assertions express restrictions of objects' eventual authority on other objects.

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

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

The relation $\rightsquigarrow^*$ contains more than the transitive closure of $\rightsquigarrow$. *E.g.*,, $\overline{M};\ \sigma_9 \rightsquigarrow^* \sigma_{13}$, even though $\overline{M};\ \sigma_9 \rightsquigarrow \sigma_{12}$ and $\overline{M};\ \sigma_{12} \not\rightsquigarrow^* \sigma_{13}$. Lemma 4.3 says that the value of the parameters does not change during execution of the same method. Appendix B discusses proofs, and further properties.

Lemma 4.3. For all $\overline{M}, \sigma, \sigma'$: $\quad \overline{M};\ \sigma \rightsquigarrow^* \sigma' \ \wedge\ |\sigma| = |\sigma'| \quad \Longrightarrow \quad \forall x \in Prms(\overline{M}, \sigma).[\ \lfloor x \rfloor_\sigma = \lfloor x \rfloor_{\sigma'}\ ]$

4.2 Reachable Objects, Locally Reachable Objects, and Well-formed States

To define protection (that an object is not indirectly reachable from another object, or from a stack frame then directly via an external object. § 2.1.1) we first define reachability and state well-formedness.

An object α is *locally reachable*, i.e. $\alpha \in LocRchbl(\sigma)$, if it is reachable from the top frame on the stack of σ.

Definition 4.4. We define

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

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

Definition 4.5 (Well-formed states). For modules $\overline{M}$, and states σ, σ':

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

Lemma 4.6 says that (1) execution preserves well-formedness, and (2) any object which is locally reachable after pushing a frame was locally reachable before pushing that frame.

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

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

5 Assertions

Our assertions are standard or *object capability*. Standard assertions assert properties of the values of fields, implication, quantification etc, as well as ghost fields which represent user-defined predicates. Oxbject capability assertions express restrictions of objects' $\boxed{\text{eventual authority}}$ on other objects.

Definition 5.1. Assertions, A, are defined as follows:

$$A \quad ::= \quad e \ \mid\ e : C \ \mid\ \neg A \ \mid\ A\ \wedge\ A \ \mid\ \forall x : C.A \ \mid\ e : \mathtt{extl} \ \mid\ \langle e \rangle \leftrightarrow\!\!\!\!\!\times\, e \ \mid\ \langle e \rangle \qquad [9]$$

[9]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. 7.4.

Definition 5.1. Assertions, A, are defined as follows:

$$A \ ::= \ e \ | \ e:C \ | \ \neg A \ | \ A \wedge A \ | \ \forall x:C.A \ | \ e:\texttt{extl} \ | \ \langle e\rangle \leftrightarrow\!\!\!\!\times\, e \ | \ \langle e\rangle$$ [9]

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

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

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

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) $M, \sigma \models e : C \ \triangleq \ M, \sigma, e \hookrightarrow \alpha \ \wedge \ classOf(\alpha, \sigma) = C$
(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 to look up the definitions of ghost fields, and to determine whether objects are external.

5.2 Semantics of Assertions - second part

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

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

An object is protected, $\langle \alpha\rangle$, if no external object reachable from any of the current frame's arguments has a field pointing to α, and if the receiver is external, then α is not the value of any parameter. This ensures that no external objects reachable from the current receiver or arguments have direct access to α – such direct access is either through a field, or by virtue of the receiver having access to all the arguments.

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

(1) $M, \sigma \models \langle \alpha\rangle \leftrightarrow\!\!\!\!\times\, \alpha_o \ \triangleq$
  (a) $\alpha \neq \alpha_0$,
  (b) $\forall \alpha'.\forall f.[\ \alpha' \in Rchbl(\alpha_o, \sigma) \wedge \ M, \sigma \models \alpha' : \texttt{extl} \implies \lfloor \alpha_o.f \rfloor_\sigma \neq \alpha\ ]$.
(2) $M, \sigma \models \langle \alpha\rangle \ \triangleq$
  (a) $M, \sigma \models \texttt{extl} \implies \forall x \in \sigma.\ M, \sigma \models x \neq \alpha$,

[9] 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. 7.4.

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

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

Satisfaction of Assertions by a module and a state is expressed through $M, \sigma \models A$ and defined by cases on the shape of A, in definitions 5.3 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. Ghost fields may be recursively defined, thus evaluation of e might not terminate. Nevertheless, the logic of assertions remains classical because recursion is restricted to expressions.

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

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

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

5.2 Semantics of Assertions - second part

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

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

An object is protected, $\langle\!\langle \alpha \rangle\!\rangle$, if no external object reachable from any of the current frame's arguments has a field pointing to α; and furthermore, if the receiver is external, then no parameter to the current method call directly refers to α.

This ensures that no external objects reachable from the current receiver or arguments have direct access to α: such direct access is either through a field, or by virtue of the receiver having access to all the arguments.

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

(1) $M, \sigma \models \langle\!\langle \alpha \rangle\!\rangle \leftrightarrow\!\!\!\!\!\times\ \alpha_o \quad \triangleq$
  (a) $\alpha \neq \alpha_0$,
  (b) $\forall \alpha'.\forall f.[\ \alpha' \in Rchbl(\alpha_o, \sigma) \wedge \ M, \sigma \models \alpha' : \texttt{extl} \implies \lfloor \alpha_o.f \rfloor_\sigma \neq \alpha\ ]$.
(2) $M, \sigma \models \langle\!\langle \alpha \rangle\!\rangle \quad \triangleq$
  (a) $M, \sigma \models \texttt{extl} \implies \forall x \in \sigma.\ M, \sigma \models x \neq \alpha$,
  (b) $\forall \alpha'.\forall f.[\ \alpha' \in LocRchbl(\sigma) \wedge \ M, \sigma \models \alpha' : \texttt{extl} \implies \lfloor \alpha_o.f \rfloor_\sigma \neq \alpha\ ]$.

Moreover,

(3) $M, \sigma \models \langle\!\langle e \rangle\!\rangle \leftrightarrow\!\!\!\!\!\times\ e_o \triangleq \exists \alpha, \alpha_o.[\ M, \sigma, e \hookrightarrow \alpha \ \wedge \ M, \sigma, e_0 \hookrightarrow \alpha_0 \ \wedge \ M, \sigma \models \langle\!\langle \alpha \rangle\!\rangle \leftrightarrow\!\!\!\!\!\times\ \alpha_o\ ]$,
(4) $M, \sigma \models \langle\!\langle e \rangle\!\rangle \triangleq \exists \alpha.[\ M, \sigma, e \hookrightarrow \alpha \ \wedge \ M, \sigma \models \langle\!\langle \alpha \rangle\!\rangle\ ]$.

(b) $\forall \alpha'.\forall f.[\ \alpha' \in LocRchbl(\sigma) \land\ M,\sigma \models \alpha' : \texttt{extl} \implies \lfloor \alpha_o.f \rfloor_\sigma \neq \alpha\ ]$.

Moreover,

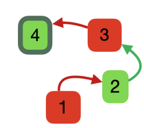
(3) $M,\sigma \models \langle\!\langle e \rangle\!\rangle \leftrightarrow\!\!\!\!\times\ e_o \triangleq \exists \alpha, \alpha_o.[\ M,\sigma,e \hookrightarrow \alpha\ \land\ M,\sigma,e_0 \hookrightarrow \alpha_0\ \land\ M,\sigma \models \langle\!\langle \alpha \rangle\!\rangle \leftrightarrow\!\!\!\!\times\ \alpha_o\ ]$,

(4) $M,\sigma \models \langle\!\langle e \rangle\!\rangle \triangleq \exists \alpha.[\ M,\sigma,e \hookrightarrow \alpha\ \land\ M,\sigma \models \langle\!\langle \alpha \rangle\!\rangle\ ]$.

We illustrate "protected" and "protected from" in Fig. 2 in §2, and Fig. ?? in App. ??. In general, $\langle\!\langle \alpha \rangle\!\rangle \leftrightarrow\!\!\!\!\times\ \alpha_o$ ensures that α_o will get access to α only if another object grants that access. Similarly, $\langle\!\langle \alpha \rangle\!\rangle$ ensures that during execution of the current method, no external object will get direct access to α unless some internal object grants that access. [10] Thus, protection together with protection preservation (*i.e.* no internal object gives access) guarantee lack of eventual external access.

Discussion. Lack of eventual direct access is a central concept in the verification of code with calls to and callbacks from untrusted code. It has already been over-approximated in several different ways, *e.g.* 2nd-class [95, 116] or borrowed ("2nd-hand") references [14, 22], textual modules [73], information flow [109], runtime checks [4], abstract data type exports [69], separation-based invariants Iris [45, 99], – 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".

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



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 protection depends not only of the heap but also on the mapping from the top frame. *E.g.*, Fig. 2 where $\sigma_2 \not\models \langle\!\langle o_6 \rangle\!\rangle$, but after pushing a frame, we have $\sigma_3 \models \langle\!\langle o_6 \rangle\!\rangle$.

Assertions which do not contain $\langle\!\langle \_ \rangle\!\rangle$, called $Stbl(\_)$, are preserved when pushing/popping frames. Less strictly, assertions which do not contain $\langle\!\langle \_ \rangle\!\rangle$ in *negative* positions, called $Stb^+(\_)$, are preserved when pushing internal frames. *C.f.* Lemma 6.2, and Appendix C for full definitions and proofs.

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

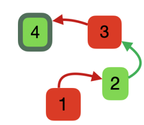
- $Stbl(A) \implies [\ M,\sigma \models A \iff M,\sigma \triangledown \phi \models A\ ]$
- $Stb^+(A)\ \land\ M \cdot \overline{M} \models \sigma \triangledown \phi\ \land\ M,\sigma \triangledown \phi \models \texttt{intl}\ \land\ M,\sigma \models A \implies M,\sigma \triangledown \phi \models A$

[10]This is in line with the motto "only connectivity begets connectivity" from [82].

We illustrate "protected" and "protected from" in Fig. 2 in §2. In general, $\langle\!\langle \alpha \rangle\!\rangle \not\leftrightarrow \alpha_o$ ensures that α_o will get access to α only if another object grants that access. Similarly, $\langle\!\langle \alpha \rangle\!\rangle$ ensures that during execution of the current method, no external object will get direct access to α unless some internal object grants that access[10]. Thus, protection together with protection preservation (*i.e.* no internal object gives access) guarantee lack of eventual external access.

Discussion. Lack of eventual direct access is a central concept in the verification of code with calls to and callbacks from untrusted code. It has already been over-approximated in several different ways, *e.g.* 2nd-class [93, 115] or borrowed ("2nd-hand") references [14, 22], textual modules [71], information flow [108], runtime checks [4], abstract data type exports [67], separation-based invariants Iris [44, 98], – 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".

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



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

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 protection depends not only of the heap but also on the mapping from the top frame. *E.g.*, Fig. 2 where $\sigma_2 \not\models \langle\!\langle o_6 \rangle\!\rangle$, but after pushing a frame, we have $\sigma_3 \models \langle\!\langle o_6 \rangle\!\rangle$.

Assertions which do not contain $\langle\!\langle \_ \rangle\!\rangle$, called $Stbl(\_)$, are preserved when pushing/popping frames. Less strictly, assertions which do not contain $\langle\!\langle \_ \rangle\!\rangle$ in *negative* positions, called $Stb^+(\_)$, are preserved when pushing internal frames. *C.f.* Lemma 6.2, and Appendix C for full definitions and proofs.

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

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

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

[10]This is in line with the motto "only connectivity begets connectivity" from [80].

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

Example 6.3. Fig. 2 ilustrates that
– *Stb^+ not necessarily preserved by External Push*: Namely, $\sigma_2 \models \langle o_4 \rangle$, pushing frame ϕ_3 with an external receiver and o_4 as argument gives σ_3, we have $\sigma_3 \not\models \langle o_4 \rangle$.
– *Stb^+ not necessarily preserved by Pop*: Namely, $\sigma_3 \models \langle o_6 \rangle$, returning from σ_3 would give σ_2, and we have $\sigma_2 \not\models \langle o_6 \rangle$.

We work with Stb^+ assertions (the *Stbl* requirement is too strong). But we need to address the lack of preservation of Stb^+ assertions for external method calls and returns. We do the former through *adaptation* ($-\triangledown$ in Sect 8.2.2), and the latter through *scoped satisfaction* (§9).

6.2 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 6.4 (An assertion A is *encapsulated* by module M)**.**

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

Lemma 6.5 (Encapsulation Soundness)**.** For all modules M, and assertions A:

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

7 SPECIFICATIONS

We now define syntax and semantics of our specifications, and illustrate through examples. Our specification language supports scoped invariants, method specifications, and conjunctions.

Definition 7.1 (Specifications Syntax)**.** We define the syntax of specifications, S:

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

Def. D.1 describes well-formedness of specifications, $\vdash S$. We require for scoped invariants, that the assertion is encapsulated, and that its free variables are bound by the quantifier. For method specifications, that the three assertions are $Stbl^+$, that the invariant part is encapsulated, that `res` and `this` are not in the formal parameters, that the free variables in the postcondition are either formal parameters or free in the precondition, and similar for the invariant part.

To give the semantics of specification we first define quadruples involving states rather than statements : $\overline{M}; M \models \{\, A \,\}\, \sigma \,\{\, A' \,\} \parallel \{\, A'' \,\}$ says that if σ satisfies A, then any terminating execution of its continuation ($\overline{M} \cdot M;\ \sigma \rightsquigarrow^*_{fin} \sigma'$) will satisfy A', and any intermediate reachable external state (here σ'') will satisfy A''. In A', we replace A's free variables by their denotation in σ.

Definition 7.2. For modules $\overline{M}$, M, state σ, and assertions A, A' and A'', we define:

- $\overline{M}; M \models \{\, A \,\}\, \sigma \,\{\, A' \,\} \parallel \{\, A'' \,\} \ \triangleq\ \forall \overline{z}, \overline{w}, \sigma', \sigma''.[$
  $M, \sigma \models A \implies$

Example 6.3. Fig. 2 ilustrates that
– *Stb^+ not necessarily preserved by External Push*: Namely, $\sigma_2 \models \langle\!\langle o_4 \rangle\!\rangle$, pushing frame ϕ_3 with an external receiver and o_4 as argument gives σ_3, we have $\sigma_3 \not\models \langle\!\langle o_4 \rangle\!\rangle$.
– *Stb^+ not necessarily preserved by Pop*: Namely, $\sigma_3 \models \langle\!\langle o_6 \rangle\!\rangle$, returning from σ_3 would give σ_2, and we have $\sigma_2 \not\models \langle\!\langle o_6 \rangle\!\rangle$.

We work with Stb^+ assertions (the *Stbl* requirement is too strong). But we need to address the lack of preservation of Stb^+ assertions for external method calls and returns. We do the former through *adaptation* ($-\triangledown$ in Sect 8.2.2), and the latter through *scoped satisfaction* (§9).

6.2 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 6.4 (An assertion A is *encapsulated* by module M)**.**

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

Lemma 6.5 (Encapsulation Soundness)**.** For all modules M, and assertions A:
$$M \vdash Enc(A) \quad\Longrightarrow\quad M \models Enc(A).$$

7 Specifications

We now define syntax and semantics of our specifications, and illustrate through examples. Our specification language supports scoped invariants, method specifications, and conjunctions.

Definition 7.1 (Specifications Syntax)**.** We define the syntax of specifications, S:

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

Def. D.1 describes well-formedness of specifications, $\vdash S$. We require for scoped invariants, that the assertion is encapsulated, and that its free variables are bound by the quantifier. For method specifications, that the three assertions are $Stbl^+$, that the invariant part is encapsulated, that `res` and `this` are not in the formal parameters, that the free variables in the postcondition are either formal parameters or free in the precondition, and similar for the invariant part.

To give the semantics of specification we first define quadruples involving states rather than statements : $\overline{M}; M \models \{\,A\,\}\,\sigma\,\{\,A'\,\} \parallel \{\,A''\,\}$ says that if σ satisfies A, then any terminating execution of its continuation ($\overline{M}{\cdot}M;\, \sigma {\rightsquigarrow}^*_{fin}\, \sigma'$) will satisfy A', and any intermediate reachable external state (here σ'') will satisfy A''. In A', we replace A's free variables by their denotation in σ.

Definition 7.2. For modules $\overline{M}$, M, state σ, and assertions A, A' and A'', we define:

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

$$
\begin{aligned}
&[\quad \overline{M}\cdot M;\ \sigma \rightsquigarrow^*_{fin} \sigma' \quad \Longrightarrow \quad M, \sigma' \models A' \quad] \quad \wedge \\
&[\quad \overline{M}\cdot M;\ \sigma \rightsquigarrow^* \sigma'' \quad \Longrightarrow \quad M, \sigma'' \models (\texttt{extl} \rightarrow A''[\overline{\lfloor z \rfloor_\sigma / z}]) \quad] \\
&\text{where } \overline{z} = Fv(A) \quad]
\end{aligned}
$$

Example 7.3. $\overline{M}; M \models \{ A_1 \} \, \sigma_4 \, \{ A_2 \} \parallel \{ A_3 \}$ in Fig. 3, assuming σ_4 satisfies A_1 and σ_{23} has empty continuation, then σ_{23} will satisfy A_2, while σ_6-σ_9, σ_{13}-σ_{17}, σ_{20}-σ_{21} will satisfy A_3.

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

Definition 7.4 (Semantics of Specifications). We define $M \models S$ by cases over$S$:

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

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

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

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

Example 7.5 (Scoped Invariants). S_5 guarantees that non-null keys do not change:

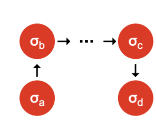
$$S_5 \quad \triangleq \quad \forall \texttt{a} : \texttt{Account}.\texttt{k} : \texttt{Key}.\{\texttt{null} \neq \texttt{k} = \texttt{a.key}\}$$

Example 7.6 (Method Specifications). A specification for method `buy` appeared in §2.2. Here, S_9 guarantees that `set` preserves the protectedness of any account, and any key.

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

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

Discussion: Comparing with Object and History Invariants. Our scoped invariants y are similar to, but different from, history invariants and object invariants. but neither of these provide what we need. We compare through an example:



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

History invariants [26, 66, 68], instead, consider all future states including any method returns , and therefore would require that $\sigma_d \models A$. Thus, they are, for our purposes, both *unenforceable* and overly *restrictive*. *Unenforceable*: Take $A \overset{\text{txt}}{=} \langle\!\langle \texttt{acc.key} \rangle\!\rangle$, assume in σ_a a path to an external object which has access to `acc.key`, assume that path is unknown in σ_b: then, the transition from σ_b to σ_c cannot eliminate that path—hence, $\sigma_d \not\models \langle\!\langle \texttt{acc.key} \rangle\!\rangle$. *Restrictive*: Take $A \overset{\text{txt}}{=} \langle\!\langle \texttt{acc.key} \rangle\!\rangle \wedge a.\texttt{blnce} \geq b$; then, requiring A to hold in all states from σ_a until termination would prevent all future withdrawals from a, rendering the account useless.

Example 7.3. $\overline{M}; M \models \{\, A_1 \,\}\, \sigma_4 \,\{\, A_2 \,\} \;\parallel\; \{\, A_3 \,\}$ in Fig. 3, assuming σ_4 satisfies A_1 and σ_{23} has empty continuation, then σ_{23} will satisfy A_2, while σ_6-σ_9, σ_{13}-σ_{17}, σ_{20}-σ_{21} will satisfy A_3.

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

Definition 7.4 (Semantics of Specifications)**.** We define $M \models S$ by cases over$S$:

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

(2) $M \models \{\, A_1 \,\}\, p\ D\!::\!m(\overline{y:D})\, \{\, A_2 \,\} \;\parallel\; \{A_3\} \quad \triangleq \quad \forall \overline{M}, \sigma.[$

$\forall y_0, \overline{y}, \sigma [\quad \sigma.\texttt{cont} \overset{\text{txt}}{=} u := y_0.m(y_1, ..y_n) \quad \Longrightarrow \quad M \models \{\, A_1' \,\}\, \sigma \{\, A_2' \,\} \;\parallel\; \{\, A_3' \,\} \quad]$

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) $M \models S \;\wedge\; S' \quad \triangleq \quad M \models S \;\wedge\; M \models S'$

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

Example 7.5 (Scoped Invariants)**.** S_5 guarantees that non-null keys do not change:

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

Example 7.6 (Method Specifications)**.** A specification for method `buy` appeared in §2.2. Here, S_9 guarantees that `set` preserves the protectedness of any account, and any key.

$S_9 \quad \triangleq \quad \{\, a : \texttt{Account}, a' : \texttt{Account} \wedge \langle\!\langle a \rangle\!\rangle \wedge \langle\!\langle a'.\texttt{key} \rangle\!\rangle \,\}$

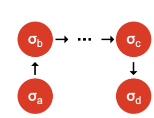
`public Account :: set(key' : Key)`

$\{\, \langle\!\langle a \rangle\!\rangle \wedge \langle\!\langle a'.\texttt{key} \rangle\!\rangle \,\} \;\parallel\; \{\langle\!\langle a \rangle\!\rangle \wedge \langle\!\langle a'.\texttt{key} \rangle\!\rangle\}$

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

Discussion: Comparing with Object and History Invariants. Our scoped invariants y are similar to, but different from, history invariants and object invariants. but neither of these provide what we need. We compare through an example:

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



History invariants [26, 64, 66], instead, consider all future states including any method returns , and therefore would require that $\sigma_d \models A$. Thus, they are, for our purposes, both *unenforceable* and overly *restrictive*. *Unenforceable*: Take $A \overset{\text{txt}}{=} \langle\!\langle \texttt{acc.key} \rangle\!\rangle$, assume in σ_a a path to an external object which has access to `acc.key`, assume that path is unknown in σ_b: then, the transition from σ_b to σ_c cannot eliminate that path—hence, $\sigma_d \not\models \langle\!\langle \texttt{acc.key} \rangle\!\rangle$. *Restrictive*: Take $A \overset{\text{txt}}{=} \langle\!\langle \texttt{acc.key} \rangle\!\rangle \wedge a.\texttt{blnce} \geq b$; then, requiring A to hold in all states from σ_a until termination would prevent all future withdrawals from a, rendering the account useless.

Object invariants [8, 63, 78, 79, 88], on the other hand, expect invariants to hold in all (visible) states, here would require, *e.g.* that $\sigma_a \models A$. Thus, they are *inapplicable* for us: They would require, *e.g.*, that for all `acc`, in all (visible) states, $\langle\!\langle \texttt{acc.key} \rangle\!\rangle$, and thus prevent *any* withdrawals from *any* account in *any* state.

Object invariants [8, 65, 80, 81, 90], on the other hand, expect invariants to hold in all (visible) states, here would require, *e.g.* that $\sigma_a \models A$. Thus, they are *inapplicable* for us: They would require, *e.g.*, that for all `acc`, in all (visible) states, ⟪`acc.key`⟫, and thus prevent *any* withdrawals from *any* account in *any* state.

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

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

Then, the assertion ⟪`this.key`⟫ is broken by the external call `extArg.m(this.key)`, but is established by `his.key:=new Key`. Therefore, ⟪`this.key`⟫ is not an invariant. The specification of `tempLeak` could be

$$S_{\texttt{tempLeak}} \triangleq \{\,\texttt{true}\,\}\ \texttt{public Account :: tempLeak(extArg : external)}\ \{\,\langle\!\langle\texttt{this.key}\rangle\!\rangle\,\}\ \|\ \{\,\texttt{true}\,\}$$

8 HOARE LOGIC

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

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.

Second Phase: We develop a logic of quadruples $M \vdash \{A\}\ stmt\ \{A'\}\ \|\ \{A''\}$. 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.

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.

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

8.1 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'.

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

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

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

Then, the assertion ⟪`this.key`⟫ is broken by the external call `extArg.m(this.key)`, but is established by `his.key:=new Key`. Therefore, ⟪`this.key`⟫ is not an invariant. The specification of `tempLeak` could be

$$S_{\texttt{tempLeak}} \triangleq \{\ \texttt{true}\ \}\ \ \texttt{public Account} :: \texttt{tempLeak}(\texttt{extArg} : \texttt{external})\ \ \{\ \langle\!\langle \texttt{this.key} \rangle\!\rangle\ \}\ \|\ \{\ \texttt{true}\ \}$$

8 Hoare Logic

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

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.

Second Phase: We develop a logic of quadruples $M\ \vdash\ \{\,A\,\}\ stmt\ \{\,A'\,\}\ \ \|\ \ \{\,A''\,\}$. 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.

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.

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

8.1 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'.

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. F.5.1.

Note that the only way that"protection" of an object can decrease is if we call an external method, and pass it an internal object 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.

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

$$\frac{u \overset{\text{txt}}{\neq} x}{M \vdash \{\, true \,\}\ u = \mathtt{new}\ C\ \{\, \langle u \rangle\ \wedge \langle u \rangle \not\leftrightarrow x \,\}}\ [\text{PROT-NEW}]$$

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

$$\frac{\begin{array}{c} stmt \text{ is either } x := y \text{ or } x := y.f \qquad z, z' \overset{\text{txt}}{\neq} x \\ M \vdash \{\, A \wedge z{=}e \wedge z'{=}e' \,\}\ stmt \{\, z{=}e \wedge z'{=}e' \,\} \end{array}}{M \vdash \{\ A \wedge \langle e \rangle \not\leftrightarrow e'\ \}\ stmt \{\ \langle e \rangle \not\leftrightarrow e'\ \}}\ [\text{PROT-2}]$$

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

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

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

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 the only way that"protection" of an object can decrease is if we call an external method, and pass it an internal object 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.

8.2 Second Phase: Quadruples

8.2.1 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)

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

Substructural quadruple rules appear in Fig. 13, and are as expected: Rules sequ and consequ are the usual rules for statement sequences and consequence, adapted to quadruples. Rule combine combines two quadruples for the same statement into one. Rule Absurd allows us to deduce anything our of `false` precondition, and Cases allows for case analysis. These rules apply to *any* statements – even those containing method calls.

8.2.2 Adaptation. As discussed in §2.2, the $-\!\triangledown$ operator adapts an assertion from the view of the callee to that of the caller, and is used in the Hoare logic for method calls. It is defined below.

Definition 8.2. [The $-\!\triangledown$ operator]

$$\begin{array}{rcl@{\qquad}rcl} \langle e \rangle -\!\triangledown \overline{y} & \triangleq & \langle e \rangle \not\leftrightarrow \overline{y} & (A_1 \wedge A_2) -\!\triangledown \overline{y} & \triangleq & (A_1 -\!\triangledown \overline{y}) \wedge (A_2 -\!\triangledown \overline{y}) \\ (\langle e \rangle \not\leftrightarrow \overline{u}) -\!\triangledown \overline{y} & \triangleq & \langle e \rangle \not\leftrightarrow \overline{u} & (\forall x : C.A) -\!\triangledown \overline{y} & \triangleq & \forall x : C.(A -\!\triangledown \overline{y}) \\ (e : \mathtt{extl}) -\!\triangledown \overline{y} & \triangleq & e : \mathtt{extl} & (\neg A) -\!\triangledown \overline{y} & \triangleq & \neg (A -\!\triangledown \overline{y}) \\ e -\!\triangledown \overline{y} & \triangleq & e & (e : C) -\!\triangledown \overline{y} & \triangleq & e : C \end{array}$$

Only the first equation in Def. 8.2 is interesting: for e to be protected at a callee with arguments $\overline{y}$, it should be protected from these arguments – thus $\langle e \rangle -\!\triangledown \overline{y} = \langle e \rangle \not\leftrightarrow \overline{y}$. The notation $\langle e \rangle \not\leftrightarrow \overline{y}$ stands for $\langle e \rangle \not\leftrightarrow y_0\ \wedge\ ... \wedge \langle e \rangle \not\leftrightarrow y_n$, assuming that $\overline{y}{=}y_0, ...y_n$.

Lemma 8.3 states that indeed, $-\!\triangledown$ adapts assertions from the callee to the caller, and is the counterpart to the $\triangledown$. In particular: (1): $-\!\triangledown$ turns an assertion into a stable assertion. (2): If

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

$$\frac{u \overset{\text{txt}}{\neq} x}{M \;\vdash\; \{\, true \,\}\; u = \mathtt{new}\, C \;\{\, \langle u \rangle \;\wedge \langle u \rangle \not\leftrightarrow x \,\}}\;[\text{PROT-NEW}]$$

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

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

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

$$\frac{}{M \;\vdash\; \{\; \langle x \rangle \not\leftrightarrow z \;\wedge\; \langle x \rangle \not\leftrightarrow y' \,\}\; y.f = y' \;\{\; \langle x \rangle \not\leftrightarrow z \;\}}\;[\text{PROT-4}]$$

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

8.2 Second Phase: Quadruples

8.2.1 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)

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

Substructural quadruple rules appear in Fig. 13, and are as expected: Rules SEQU and CONSEQU are the usual rules for statement sequences and consequence, adapted to quadruples. Rule COMBINE combines two quadruples for the same statement into one. Rule ABSURD allows us to deduce anything our of `false` precondition, and CASES allows for case analysis. These rules apply to *any* statements – even those containing method calls.

8.2.2 Adaptation. As discussed in §2.2, the $-\!\triangledown$ operator adapts an assertion from the view of the callee to that of the caller, and is used in the Hoare logic for method calls. It is defined below.

Definition 8.2. [The $-\!\triangledown$ operator]

$$\begin{array}{rcl@{\qquad}rcl}
\langle e \rangle -\!\triangledown \overline{y} & \triangleq & \langle e \rangle \not\leftrightarrow \overline{y} & (A_1 \wedge A_2) -\!\triangledown \overline{y} & \triangleq & (A_1 -\!\triangledown \overline{y}) \wedge (A_2 -\!\triangledown \overline{y}) \\
(\langle e \rangle \not\leftrightarrow \overline{u}) -\!\triangledown \overline{y} & \triangleq & \langle e \rangle \not\leftrightarrow \overline{u} & (\forall x : C.A) -\!\triangledown \overline{y} & \triangleq & \forall x : C.(A -\!\triangledown \overline{y}) \\
(e : \mathtt{extl}) -\!\triangledown \overline{y} & \triangleq & e : \mathtt{extl} & (\neg A) -\!\triangledown \overline{y} & \triangleq & \neg(A -\!\triangledown \overline{y}) \\
e -\!\triangledown \overline{y} & \triangleq & e & (e : C) -\!\triangledown \overline{y} & \triangleq & e : C
\end{array}$$

Only the first equation in Def. 8.2 is interesting: for e to be protected at a callee with arguments $\overline{y}$, it should be protected from these arguments – thus $\langle e \rangle -\!\triangledown \overline{y} = \langle e \rangle \not\leftrightarrow \overline{y}$. The notation $\langle e \rangle \not\leftrightarrow \overline{y}$ stands for $\langle e \rangle \not\leftrightarrow y_0 \;\wedge\; ... \wedge \langle e \rangle \not\leftrightarrow y_n$, assuming that $\overline{y}{=}y_0, ...y_n$.

Lemma 8.3 states that indeed, $-\!\triangledown$ adapts assertions from the callee to the caller, and is the counterpart to the $\triangledown$. In particular: (1): $-\!\triangledown$ turns an assertion into a stable assertion. (2): If the caller, σ, satisfies $A \triangledown Rng(\phi)$, then the callee, $(\sigma -\!\triangledown \phi)$, satisfies A. (3): When returning from external states, an assertion implies its adapted version. (4): When calling from external states, an assertion implies its adapted version.

Lemma 8.3. For states σ, assertions A, so that $Stb^{+}(A)$ and $Fv(A) = \emptyset$, frame ϕ, variables y_0, $\overline{y}$:

(1) $Stbl(\, A -\!\triangledown (y_0, \overline{y})\,)$

the caller, σ, satisfies $A \triangledown Rng(\phi)$, then the callee, $(\sigma \text{-}\!\triangledown \phi)$, satisfies A. (3): When returning from external states, an assertion implies its adapted version. (4): When calling from external states, an assertion implies its adapted version.

Lemma 8.3. For states σ, assertions A, so that $Stb^+(A)$ and $Fv(A) = \emptyset$, frame ϕ, variables y_0, $\overline{y}$:

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

Proofs in Appendix F.4. Example 8.4 demonstrates the need for the `extl` requirement in (3).

Example 8.4 (When returning from internal states, A does not imply $A \text{-}\!\triangledown Rng(\phi)$. In Fig. 2 we have $\sigma_2 = \sigma_1 \triangledown \phi_2$, and $\sigma_2 \models \langle\!\langle o_1 \rangle\!\rangle$, and $o_1 \in Rng(\phi_2)$. But, since $o_1 = o_1$, we also have $\sigma_1 \not\models \langle\!\langle o_1 \rangle\!\rangle \not\leftrightarrow o_1$.

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

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

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

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

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

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

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

Notice that internal calls, Call_Int require the *un-adapted* method precondition (*i.e.* A_1'), while external calls, both Call_Ext_Adapt and Call_Ext_Adapt_Strong, require the *adapted* invariant (*i.e.* $A \text{-}\!\triangledown (y_0, \overline{y})$). This is sound, because internal callees preserve $Stb^+(\_)$-assertions – *c.f.* Lemma 6.2. On the other hand, external callees do not necessarily preserve $Stb^+(\_)$-assertions – *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 8.3.

Remember that popping frames does not necessarily preserve $Stb^+(\_)$ assertions – *c.f.* Ex. 6.3. Nevertheless, Call_Int guarantees the unadapted version, A, upon return from the call. This is sound, because of our *scoped satisfaction* of assertions – more in Sect. 9.

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

Proofs in Appendix E.4. Example 8.4 demonstrates the need for the `extl` requirement in (3).

Example 8.4 (When returning from internal states, A does not imply $A \text{-}\!\triangledown Rng(\phi)$. In Fig. 2 we have $\sigma_2 = \sigma_1 \triangledown \phi_2$, and $\sigma_2 \models \langle\!\langle o_1 \rangle\!\rangle$, and $o_1 \in Rng(\phi_2)$. But, since $o_1 = o_1$, we also have $\sigma_1 \not\models \langle\!\langle o_1 \rangle\!\rangle \not\leftrightarrow o_1$.

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

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

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

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

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

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

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

Notice that internal calls, Call_Int require the *un-adapted* method precondition (*i.e.* A_1'), while external calls, both Call_Ext_Adapt and Call_Ext_Adapt_Strong, require the *adapted* invariant (*i.e.* $A \text{-}\!\triangledown (y_0, \overline{y})$). This is sound, because internal callees preserve $Stb^+(\_)$-assertions – *c.f.* Lemma 6.2. On the other hand, external callees do not necessarily preserve $Stb^+(\_)$-assertions – *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 8.3.

Remember that popping frames does not necessarily preserve $Stb^+(\_)$ assertions – *c.f.* Ex. 6.3. Nevertheless, Call_Int guarantees the unadapted version, A, upon return from the call. This is sound, because of our *scoped satisfaction* of assertions – more in Sect. 9.

Discussion: Polymorphic Calls. Our rules do not directly address the possibility that the receiver might belong to one class or another class, or even be internal or external, and where the choice is made at runtime. However, such scenaria can be supported through the case-split rule and the rule of consequence. More details in h Appendix G.7.

Discussion: Polymorphic Calls. Our rules do not directly address the possibility that the receiver might belong to one class or another class, or even be internal or external, and where the choice is made at runtime. However, such scenaria can be supported through the case-split rule and the rule of consequence. More details in h Appendix H.7.

Example 8.5 (Proving external calls). We continue our discussion from §2.2 on how to establish the Hoare triple **(1)** :

$$\begin{array}{ll} & \{\ \texttt{buyer}:\texttt{extl}\ \wedge\ \langle\!\langle\texttt{this.accnt.key}\rangle\!\rangle\not\leftrightarrow \texttt{buyer}\ \wedge\ \texttt{this.accnt.blnce} = b\ \} \\ \textbf{(1?)} & \qquad \texttt{buyer.pay(this.accnt,price)} \\ & \{\ \texttt{this.accnt.blnce} \geq b\ \}\ \|\ \{\ \langle\!\langle\texttt{a.key}\rangle\!\rangle \wedge \texttt{a.blnce}\geq\texttt{b}\ \} \end{array}$$

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

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

holds. Using Def. 8.2, and type information, we can indeed establish that

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

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

8.3 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.

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

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

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

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. In the postcondition we also ask that $A\text{-}\!\nabla\texttt{res}$, so that res does not leak any of the values that A promises will be protected. Invariant says that a module satisfies a specification $\mathbb{\forall}\overline{x:C}.\{A\}$, if the method body of each public method has A as its pre-, post- and midcondition. Moreover, the precondition is strengthened by $A\text{-}\!\nabla(\texttt{this},\overline{y})$ – this is sound because the caller is external, and by Lemma 8.3, part (4).

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

Example 8.5 (Proving external calls). We continue our discussion from §2.2 on how to establish the Hoare triple **(1)** :

$$\begin{array}{ll} & \{\ \texttt{buyer}:\texttt{extl}\ \wedge\ \langle\!\langle\texttt{this.accnt.key}\rangle\!\rangle \not\leftrightarrow \texttt{buyer}\ \wedge\ \texttt{this.accnt.blnce} = b\ \} \\ \textbf{(1?)} & \qquad \texttt{buyer.pay(this.accnt,price)} \\ & \{\ \texttt{this.accnt.blnce} \geq b\ \}\ \|\ \{\ \langle\!\langle\texttt{a.key}\rangle\!\rangle \wedge \texttt{a.blnce} \geq \texttt{b}\ \} \end{array}$$

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

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

holds. Using Def. 8.2, and type information, we can indeed establish that

$$\textbf{(3)}\quad \langle\!\langle\texttt{this.accnt.key}\rangle\!\rangle \not\leftrightarrow (\texttt{buyer},\texttt{this.accnt},\texttt{price})\ =\ \langle\!\langle\texttt{this.accnt.key}\rangle\!\rangle \not\leftrightarrow \texttt{buyer}$$

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

8.3 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.

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

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

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

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. In the postcondition we also ask that $A -\!\nabla\,\texttt{res}$, so that res does not leak any of the values that A promises will be protected. Invariant says that a module satisfies a specification $\mathbb{\forall}\overline{x:C}.\{A\}$, if the method body of each public method has A as its pre-, post- and midcondition. Moreover, the precondition is strengthened by $A -\!\nabla(\texttt{this},\overline{y})$ – this is sound because the caller is external, and by Lemma 8.3, part (4).

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

Example 8.6 (Proving a public method). Consider the proof that $\texttt{Account::set}$ from M_{fine} satisfies S_2. Applying rule invariant, we need to establish:

$$\begin{array}{ll} & \{\ ...\texttt{a}:\texttt{Account}\ \wedge\ \langle\!\langle\texttt{a.key}\rangle\!\rangle\ \wedge\ \langle\!\langle\texttt{a.key}\rangle\!\rangle \not\leftrightarrow (\texttt{key'},\texttt{key''})\ \} \\ \textbf{(5?)} & \qquad \texttt{body\_of\_set\_in\_Account\_in\_}M_{fine} \\ & \{\ \{\langle\!\langle\texttt{a.key}\rangle\!\rangle\ \wedge\ \langle\!\langle\texttt{a.key}\rangle\!\rangle -\!\nabla\,\texttt{res}\ \}\ \|\ \{\langle\!\langle\texttt{a.key}\rangle\!\rangle\} \end{array}$$

Example 8.6 (Proving a public method). Consider the proof that `Account::set` from M_{fine} satisfies S_2. Applying rule INVARIANT, we need to establish:

$\{$ ...`a:Account` $\wedge$ ⟨`a.key`⟩ $\wedge$ ⟨`a.key`⟩$\not\leftrightarrow$ (`key'`,`key''`) $\}$
(5?) `body_of_set_in_Account_in_`M_{fine}
$\{$ $\{$⟨`a.key`⟩ $\wedge$ ⟨`a.key`⟩$\not\leftrightarrow$`res` $\}$ $||$ $\{$⟨`a.key`⟩$\}$

Given the conditional statement in `set`, and with the obvious treatment of conditionals (*c.f.* Fig. 13), among other things, we need to prove for the `true`-branch that:

$\{$ ...⟨`a.key`⟩ $\wedge$ ⟨`a.key`⟩$\not\leftrightarrow$ (`key'`,`key''`) $\wedge$ `this.key = key'` $\}$
(6?) `this.key := key'`
$\{$ $\{$⟨`a.key`⟩ $\}$ $||$ $\{$⟨`a.key`⟩$\}$

We can apply case-split (*c.f.* Fig. 13) on whether `this=a`, and thus a proof of **(7?)** and **(8?)**, gives a proof of **(6)**:

$\{$ ...⟨`a.key`⟩ $\wedge$ ⟨`a.key`⟩$\not\leftrightarrow$ (`key'`,`key''`) $\wedge$ `this.key=key'` $\wedge$ `this=a`$\}$
(7?) `this.key := key'`
$\{$ $\{$⟨`a.key`⟩ $\}$ $||$ $\{$⟨`a.key`⟩$\}$

and also

$\{$ ...⟨`a.key`⟩ $\wedge$ ⟨`a.key`⟩$\not\leftrightarrow$ (`key'`,`key''`) $\wedge$ `this.key=key'` $\wedge$ `this≠a`$\}$
(8?) `this.key := key'`
$\{$ $\{$⟨`a.key`⟩ $\}$ $||$ $\{$⟨`a.key`⟩$\}$

If `this.key = a'` $\wedge$ `this = a`, then `a.key = key'`, which contradicts that ⟨`a.key`⟩ $\not\leftrightarrow$ `key'`, and so by contradiction (*c.f.* Fig. 13), we have proven **(7?)**. When `this.key` $\neq$ `a`, we would obtain from the underlying Hoare logic that the value of `a.key` did not change. Thus, we can apply rule PROT_1, and obtain **(7?)**. More details in §H.6

This also demonstrates why `set` from M_{bad} cannot be proven to satisfy S_3. Namely, it does not have the condition `this.key=key'`, and would requires us to prove that

$\{$ ...⟨`a.key`⟩ $\wedge$ ⟨`a.key`⟩$\not\leftrightarrow$ (`key'`,`key''`) $\}$
(??) `this.key := key'`
$\{$ $\{$⟨`a.key`⟩ $\}$ $||$ $\{$ ⟨`a.key`⟩ $\}$

and there is no way we can prove **(??)**.

9 SOUNDNESS

We now outline some interesting aspects when proving soundness of the logic from §8.

Scoped Satisfaction. Remember that an assertion which held at the end of a method execution, need not hold upon return from it – *c.f.* Ex. 6.3, and G.5. To address this, we introduce *scoped satisfaction*: $M, \sigma, k \models A$ says that σ satisfies A from k onwards, if it satisfies it in k-th frame, and all the frames above it. *i.e.* $\forall j.[\ k \leq j \leq |\sigma| \implies M, \sigma[j] \models A\ ]$. We also introduce *scoped quadruples*, $M \models_{\circ} \{\, A \,\}\ \sigma\{\, A' \,\}\ \ ||\ \ \{\, A'' \,\}$, which promise for all $k \leq |\sigma|$, if σ satisfies A from k onwards, and executes its continuation to termination, then the final state will satisfy A' from k onwards, and that all intermediate external states will satisfy A'' from k onwards - *c.f.* Def G.6. More in Appendix 9. Scoped satisfaction is stronger than shallow:

Lemma 9.1 (Scoped vs Shallow Satisfaction). For all $M, A, A', A'', \sigma, stmt$:

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

Soundness of the Hoare Triples Logic. We require the assertion logic, $M \vdash A$, and the underlying Hoare logic, $M \vdash_{ul} \{\, A \,\}\ stmt\{\, A' \,\}$, to be be sound. We prove properties of protection, and soundness of the inference system for triples $M \vdash \{\, A \,\}\ stmt\{\, A' \,\}$ – *c.f.* Appendix G.5.

Given the conditional statement in `set`, and with the obvious treatment of conditionals (*c.f.* Fig. 13), among other things, we need to prove for the `true`-branch that:

$$\begin{array}{ll} & \{\ ...\langle\texttt{a.key}\rangle \land \langle\texttt{a.key}\rangle\not\leftrightarrow(\texttt{key}',\texttt{key}'') \land \texttt{this.key} = \texttt{key}'\ \} \\ \textbf{(6?)} & \qquad \texttt{this.key := key}' \\ & \{\ \{\langle\texttt{a.key}\rangle\ \}\ \ ||\ \ \{\langle\texttt{a.key}\rangle\} \end{array}$$

We can apply case-split (*c.f.* Fig. 13) on whether `this=a`, and thus a proof of **(7?)** and **(8?)**, gives a proof of **(6)**:

$$\begin{array}{ll} & \{\ ...\langle\texttt{a.key}\rangle \land \langle\texttt{a.key}\rangle\not\leftrightarrow(\texttt{key}',\texttt{key}'') \land \texttt{this.key=key}' \ \land\ \texttt{this=a}\} \\ \textbf{(7?)} & \qquad \texttt{this.key := key}' \\ & \{\ \{\langle\texttt{a.key}\rangle\}\ \ ||\ \ \{\langle\texttt{a.key}\rangle\} \end{array}$$

and also

$$\begin{array}{ll} & \{\ ...\langle\texttt{a.key}\rangle \land \langle\texttt{a.key}\rangle\not\leftrightarrow(\texttt{key}',\texttt{key}'') \land \texttt{this.key=key}' \ \land\ \texttt{this}\neq\texttt{a}\} \\ \textbf{(8?)} & \qquad \texttt{this.key := key}' \\ & \{\ \{\langle\texttt{a.key}\rangle\}\ \ ||\ \ \{\langle\texttt{a.key}\rangle\} \end{array}$$

If `this.key = a'` $\land$ `this = a`, then `a.key = key'`, which contradicts that $\langle\texttt{a.key}\rangle \not\leftrightarrow$ `key'`, and so by contradiction (*c.f.* Fig. 13), we have proven **(7?)**. When `this.key` $\neq$ `a`, we would obtain from the underlying Hoare logic that the value of `a.key` did not change. Thus, we can apply rule Prot_1, and obtain **(7?)**. More details in §G.6

This also demonstrates why `set` from M_{bad} cannot be proven to satisfy S_3. Namely, it does not have the condition `this.key=key'`, and would requires us to prove that

$$\begin{array}{ll} & \{\ ...\langle\texttt{a.key}\rangle \land \langle\texttt{a.key}\rangle\not\leftrightarrow(\texttt{key}',\texttt{key}'')\ \} \\ \textbf{(??)} & \qquad \texttt{this.key := key}' \\ & \{\ \{\langle\texttt{a.key}\rangle\ \}\ \ ||\ \ \{\ \langle\texttt{a.key}\rangle\ \} \end{array}$$

and there is no way we can prove **(??)**.

9 Soundness

We now outline some interesting aspects when proving soundness of the logic from §8.

Scoped Satisfaction. Remember that an assertion which held at the end of a method execution, need not hold upon return from it – *c.f.* Ex. 6.3, and F.5. To address this, we introduce *scoped satisfaction*: $M, \sigma, k \models A$ says that σ satisfies A from k onwards, if it satisfies it in k-th frame, and all the frames above it. *i.e.* $\forall j.[\ k \leq j \leq |\sigma| \ \Rightarrow\ M, \sigma[j] \models A\ ]$. We also introduce *scoped quadruples*, $M \models_{\circ} \{\ A\ \}\ \sigma\{\ A'\ \}\ \ ||\ \ \{\ A''\ \}$, which promise for all $k \leq |\sigma|$, if σ satisfies A from k onwards, and executes its continuation to termination, then the final state will satisfy A' from k onwards, and that all intermediate external states will satisfy A'' from k onwards - *c.f.* Def F.6. More in Appendix 9. Scoped satisfaction is stronger than shallow:

Lemma 9.1 (Scoped vs Shallow Satisfaction)**.** For all $M, A, A', A'', \sigma, stmt$:

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

Soundness of the Hoare Triples Logic. We require the assertion logic, $M \vdash A$, and the underlying Hoare logic, $M\ \ \vdash_{ul}\ \ \{\ A\ \}\ stmt\{\ A'\ \}$, to be be sound. We prove properties of protection, and soundness of the inference system for triples $M \vdash \{\ A\ \}\ stmt\{\ A'\ \}$ – *c.f.* Appendix F.5.

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

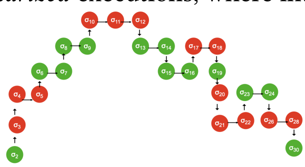
$$M\ \vdash\ \{\ A\ \}\ stmt\{\ A'\ \} \quad\Longrightarrow\quad M \models_{\circ} \{\ A\ \}\ stmt\{\ A'\ \}\ \ ||\ \ \{\ A''\ \}$$

Summarised Execution. Execution of an external call may consist of any number of external transitions, interleaved with calls to public internal methods, which in turn may make any number

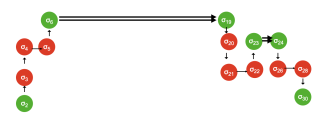
Theorem 9.2. For module M such that $\vdash M$, and for any assertions A, A', A'' and statement $stmt$:

$$M \ \vdash \ \{\, A \,\}\, stmt \{\, A' \,\} \quad \Longrightarrow \quad M \models_{\circ} \{\, A \,\}\, stmt \{\, A' \,\} \ \parallel \ \{\, A'' \,\}$$

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



summarized to



Lemma G.27 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.28 says that such an execution preserves an encapsulated assertion A provided that all these finalising internal executions also preserve A.

Soundness of the Hoare Quadruples Logic. Proving soundness of our quadruples in some cases requires induction on the execution while in other cases requires induction on the derivation of the quadruples. We address this through a well-founded ordering that combines both, *c.f.* Def. G.21 and lemma G.22. Finally, in G.16, we prove soundness:

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

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

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

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

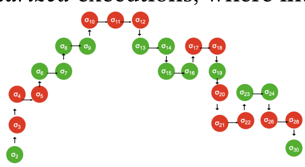
Our motivation comes from the OCAP approach to security, whereby object capabilities guard against un-sanctioned effects. Miller [82, 84] advocates *defensive consistency*: whereby "An object is defensively consistent when it can defend its own invariants and provide correct service to its well behaved 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 [91].

Our Work aims to elucidate such guarantees. We want to formalize and prove that [41]:

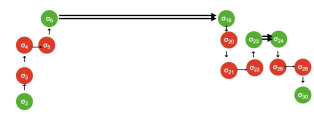
> *Lack of eventual access implies that certain properties will be preserved, even in the presence of external calls.*

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.

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



summarized to



Lemma F.27 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 F.28 says that such an execution preserves an encapsulated assertion A provided that all these finalising internal executions also preserve A.

Soundness of the Hoare Quadruples Logic. Proving soundness of our quadruples in some cases requires induction on the execution while in other cases requires induction on the derivation of the quadruples. We address this through a well-founded ordering that combines both, *c.f.* Def. F.21 and lemma F.22. Finally, in F.16, we prove soundness:

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

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

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

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 G, 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 [80, 82] 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 [89].

Our Work aims to elucidate such guarantees. We want to formalize and prove that [41]:

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

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

For lack of eventual access, we introduced protection, 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

For lack of eventual access, we introduced 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, 93]. Ownership is fundamental to key systems like Rust's memory safety [57, 61], Scala's Concurrency [47, 48], Java heap analyses [51, 87, 97], and plays a critical role in program verification [13, 64] including Spec# [8, 9] and universes [33, 34, 71], Borrowable Fractional Ownership [92], and recently integrated into languages like OCAML [70, 75].

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 [62, 102], Reachability Types [7, 113] and Capturing Types [12, 17, 117] 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 [72] directly prohibits nominated effects, but within a closed, fully-typed world.

Temporal scope of the guarantee Starting with loop invariants[43, 52], property preservation at various granularities and durations has been widely and successfully adapted and adopted [8, 26, 38, 53, 65, 66, 68, 80, 81, 90]. 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 §4.1.

Such guarantees are maintained by the module as a whole. Drossopoulou et al. [40] proposed "holistic specifications" which take an external perspective across the interface of a module. Mackay et al. [73] 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 [40, 73] 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 [52], and have inspired a plethora of influential further developments and tools. We shall discuss a few only.

Separation logics [55, 100] reason about disjoint memory regions. Incorporating Separation Logic's powerful framing mechanisms will pose several challenges: We have no specifications and no footprint for external calls. Because protection is "scope-aware", expressing it as a predicate would require quantification over all possible paths and variables within the current stack frame.

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, 91]. Ownership is fundamental to key systems like Rust's memory safety [56, 59], Scala's Concurrency [46, 47], Java heap analyses [50, 85, 96], and plays a critical role in program verification [13, 62] including Spec# [8, 9] and universes [33, 34, 69], Borrowable Fractional Ownership [90], and recently integrated into languages like OCAML [68, 73].

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

In future work we want to explore how to express protection within Ownership Types, with the primary challenge being how to accommodate capabilities accessible to some external objects while still inaccessible to others. Moreover, tightening some rules in our current Hoare logic (e.g. Def. 5.4) may lead to a native Hoare logic of ownership. Also, recent approaches like the Alias Calculus [60, 101], Reachability Types [7, 112] and Capturing Types [12, 17, 116] 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 [70] directly prohibits nominated effects, but within a closed, fully-typed world.

Temporal scope of the guarantee Starting with loop invariants[43, 51], property preservation at various granularities and durations has been widely and successfully adapted and adopted [8, 26, 38, 52, 63, 64, 66, 78, 79, 88]. 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 §4.1.

Such guarantees are maintained by the module as a whole. Drossopoulou et al. [40] proposed "holistic specifications" which take an external perspective across the interface of a module. Mackay et al. [71] 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 [40, 71] 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 [51], and have inspired a plethora of influential further developments and tools. We shall discuss a few only.

Separation logics [54, 99] reason about disjoint memory regions. Incorporating Separation Logic's powerful framing mechanisms will pose several challenges: We have no specifications and no footprint for external calls. Because protection is "scope-aware", expressing it as a predicate would require quantification over all possible paths and variables within the current stack frame. We may also require a new separating conjunction operator. Hyper-Hoare Logics [28, 37] 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 [92] under-approximates postconditions, and thus reasons about the presence of bugs, rather than their absence. Our work, like classical Hoare Logic, over-approximates

We may also require a new separating conjunction operator. Hyper-Hoare Logics [29, 37] 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 [94] 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 [40, 73] 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 [49, 112] and Deny-Guarantee [36] 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 [49], rather than just at method boundaries as in visible states semantics [38, 90, 107]. 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 [91] made the first attempt to formalise defensive consistency, to tolerate interacting with any untrustworthy object, although without a specification language for describing effects (i.e. when an object is correct).

Cassez et al. [21] propose one approach to reason about external calls. Given that external callbacks are necessarily restricted to the module’s public interface, external callsites are replaced with a generated `externalcall()` method that nondeterministically invokes any method in that interface. Rao et al. [99]’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. [109] 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 [57] developed this approach to prove Rust memory safety using Iris [58], and combined with RustHorn [77] for the safe subset, produced RustHornBelt [76] that verifies both safe and unsafe Rust programs. Similar techniques were extended to C [103]. 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. [32] 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 [69] and Cerise, which uses Iris invariants to support proofs of programs with outgoing calls and callbacks, on capability-safe CPUs [45], 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 [105] is a minimalistic functional 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 [40, 71] 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 [48, 111] and Deny-Guarantee [36] 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 [48], rather than just at method boundaries as in visible states semantics [38, 88, 106]. 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 [89] made the first attempt to formalise defensive consistency, to tolerate interacting with any untrustworthy object, although without a specification language for describing effects (i.e. when an object is correct).

Cassez et al. [21] propose one approach to reason about external calls. Given that external callbacks are necessarily restricted to the module's public interface, external callsites are replaced with a generated `externalcall()` method that nondeterministically invokes any method in that interface. Rao et al. [98]'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. [108] 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 [56] developed this approach to prove Rust memory safety using Iris [57], and combined with RustHorn [75] for the safe subset, produced RustHornBelt [74] that verifies both safe and unsafe Rust programs. Similar techniques were extended to C [102]. 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. [31] 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 [67] and Cerise, which uses Iris invariants to support proofs of programs with outgoing calls and callbacks, on capability-safe CPUs [44], 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 [104] 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 [95]. VerX's specification language is based on temporal logic. It is restricted to "effectively call-back free" programs [2, 45], delaying any callbacks until the incoming call to the internal object has finished.

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 [96]. VerX's specification language is based on temporal logic. It is restricted to "effectively call-back free" programs [2, 46], delaying any callbacks until the incoming call to the internal object has finished.

ConSol [114] provides a specification langauge for smart contracts, checked at runtime [42]. SCIO* [4], implemented in F*, supports both verified and unverified code. Both Consol and SCIO* are similar to gradual verification techniques [27, 118] that insert dynamic checks between verified and unverified code, and contracts for general access control [35, 59, 88].

Programming languages with object capabilities Google's Caja [86] applies (object-)capabilities [31, 82, 89], sandboxes, proxies, and wrappers to limit components' access to *ambient* authority. Sandboxing has been validated formally [74]; Many recent languages [19, 50, 101] including Newspeak [16], Dart [15], Grace [11, 56] and Wyvern [78] have adopted object capabilities. Schaefer et al. [104] 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. [104] 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 [83], rather than public/private annotations, languages that support ownership types such as Rust, that can be leveraged for verification [5, 63, 76], and languages from the functional tradition such as OCAML, which are gaining imperative features such as ownership and uniqueness[70, 75]. 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 [118], thus paving the way towards practical verification for the open world.

ConSol [113] provides a specification langauge for smart contracts, checked at runtime [42]. SCIO* [4], implemented in F*, supports both verified and unverified code. Both Consol and SCIO* are similar to gradual verification techniques [27, 117] that insert dynamic checks between verified and unverified code, and contracts for general access control [35, 58, 86].

Programming languages with object capabilities Google's Caja [84] applies (object-)capabilities [30, 80, 87], sandboxes, proxies, and wrappers to limit components' access to *ambient* authority. Sandboxing has been validated formally [72]; Many recent languages [19, 49, 100] including Newspeak [16], Dart [15], Grace [11, 55] and Wyvern [76] have adopted object capabilities. Schaefer et al. [103] 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. [103] 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 [81], rather than public/private annotations, languages that support ownership types such as Rust, that can be leveraged for verification [5, 61, 74], and languages from the functional tradition such as OCAML, which are gaining imperative features such as ownership and uniqueness[68, 73]. 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 [117], 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 auxiliary information with this submission.

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

REFERENCES

[1] Gul Agha and Carl Hewitt. 1987. Actors: A conceptual foundation for concurrent object-oriented programming. In *Research Directions in Object-Oriented Programming*. 49–74.

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

[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

Data Availability Statement

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

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

References

[1] Gul Agha and Carl Hewitt. 1987. Actors: A conceptual foundation for concurrent object-oriented programming. In *Research Directions in Object-Oriented Programming*. 49–74.

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

[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. doi: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. doi: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. doi: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. doi: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. doi: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] Tom Van Cutsem and Mark S. 2013. Trustworthy Proxies: Virtualizing Objects with Invariants. In *ECOOP*.

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

[30] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. 2006. Ambient-Oriented Programming in AmbientTalk. In *ECOOP*. 230–254.

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

[32] 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

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

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

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

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

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

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

[39] Sophia Drossopoulou and James Noble. 2013. The Need for Capability Policies. In *(FTfJP)*.

[40] 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

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

[42] 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

[43] Robert W. Floyd. 1967. Assigning Meanings to Programs. *Marhematical Aspects of Computer Science* 19 (1967), 19–32.

[44] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. 1994. *Design Patterns*.

[45] 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.

[46] 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

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

[48] 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

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

[50] 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

[51] 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.

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

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

[54] Daniel H. Ingalls. 1981. Design Principles Behind Smalltalk. *BYTE* 6, 8 (August 1981), 286–298.

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

[56] 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

[57] 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.

[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. doi: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. doi:10.1145/3656437
[29] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. 2006. Ambient-Oriented Programming in AmbientTalk. In *ECOOP*. 230–254.
[30] Jack B. Dennis and Earl C. Van Horn. 1966. Programming Semantics for Multiprogrammed Computations. *Comm. ACM* 9, 3 (1966).
[31] Dominique Devriese, Lars Birkedal, and Frank Piessens. 2016. Reasoning about Object Capabilities with Logical Relations and Effect Parametricity. In *IEEE EuroS&P*. 147–162. doi:10.1109/EuroSP.2016.22
[32] W. Dietl, S. Drossopoulou, and P. Müller. 2007. Generic Universe Types. In *ECOOP (LNCS, Vol. 4609)*. Springer, 28–53. http://www.springerlink.com
[33] W. Dietl and P. Müller. 2005. Universes: Lightweight Ownership for JML. *JOT* 4, 8 (October 2005), 5–32.
[34] W. Dietl, Drossopoulou S., and P. Müller. 2007. Generic Universe Types. (Jan 2007). FOOL/WOOD'07.
[35] Christos Dimoulas, Scott Moore, Aslan Askarov, and Stephen Chong. 2014. Declarative Policies for Capability Control. In *Computer Security Foundations Symposium (CSF)*.
[36] Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. 2009. Deny-guarantee reasoning. In *ESOP*. Springer.
[37] Emanuele D'Osualdo, Azadeh Farzan, and Derek Dreyer. 2022. Proving hypersafety compositionally. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 289–314.
[38] S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers. 2008. A Unified Framework for Verification Techniques for Object Invariants. In *ECOOP (LNCS)*. Springer.
[39] Sophia Drossopoulou and James Noble. 2013. The Need for Capability Policies. In *(FTfJP)*.
[40] Sophia Drossopoulou, James Noble, Julian Mackay, and Susan Eisenbach. 2020. Holistic Specifications for Robust Programs. In *FASE*. Cham, 420–440. doi:10.1007/978-3-030-45234-6_21
[41] Sophia Drossopoulou, James Noble, Mark Miller, and Toby Murray. 2016. Permission and Authority revisited – towards a formalization. In *(FTfJP)*.
[42] 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. doi:10.1145/504282.504283
[43] Robert W. Floyd. 1967. Assigning Meanings to Programs. *Marhematical Aspects of Computer Science* 19 (1967), 19–32.
[44] 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.
[45] 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). doi:10.1145/3158136
[46] Philipp Haller. 2024. Lightweight Affine Types for Safe Concurrency in Scala (Keynote). In *Programming*.
[47] Philipp Haller and Alexander Loiko. 2016. LaCasa: lightweight affinity and object capabilities in Scala. In *OOPSLA*. 272–291. doi:10.1145/2983990.2984042
[48] Ian J. Hayes and Cliff B. Jones. 2018. A Guide to Rely/Guarantee Thinking. In *SETSS 2017 (LNCS11174)*. 1–38.
[49] Ian J. Hayes, Xi Wu, and Larissa A. Meinicke. 2017. Capabilities for Java: Secure Access to Resources. In *APLAS*. 67–84. doi:10.1007/978-3-319-71237-6_4
[50] 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.
[51] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Comm. ACM* 12 (1969), 576–580.
[52] C. A. R. Hoare. 1974. Monitors: an operating system structuring concept. *Commun. ACM* 17, 10 (1974), 549–557.
[53] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *ACM ToPLAS* 23, 3 (2001), 396–450. doi:10.1145/503502.503505
[54] S. S. Ishtiaq and P. W. O'Hearn. 2001. BI as an assertion language for mutable data structures. In *POPL*. 14–26.
[55] Timothy Jones, Michael Homer, James Noble, and Kim B. Bruce. 2016. Object Inheritance Without Classes. In *ECOOP*. 13:1–13:26. doi:10.4230/LIPIcs.ECOOP.2016.13

[58] 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.

[59] 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*.

[60] B.W. Kernighan and R. Pike. 1984. *The UNIX Programming Environment.* Prentice-Hall.

[61] Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language* (2nd ed.).

[62] Alexander Kogtenkov, Bertrand Meyer, and Sergey Velder. 2015. Alias calculus, change calculus and frame inference. *Sci. Comp. Prog.* 97 (2015), 163–172.

[63] 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

[64] Dirk Leinenbach and Thomas Santen. 2009. Verifying the Microsoft Hyper-V Hypervisor with VCC. In *Formal Methods*.

[65] K. Rustan M. Leino and Peter Müller. 2004. Object Invariants in Dynamic Contexts. In *ECOOP*.

[66] K. Rustan M. Leino and Wolfram Schulte. 2007. Using History Invariants to Verify Observers. In *ESOP*.

[67] Henry M. Levy. 1984. *Capability-Based Computer Systems*. Butterworth-Heinemann.

[68] B. Liskov and J. Wing. 1994. A Behavioral Notion of Subtyping. *ACM ToPLAS* 16, 6 (1994), 1811–1841.

[69] 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.

[70] Anton Lorenzen, Stephen Dolan, Richard A. Eisenberg, and Sam Lindley. 2024. Oxidizing OCaml with Modal Memory Management. In *ICFP*.

[71] Y. Lu and J. Potter. 2006. Protecting Representation with Effect Encapsulation.. In *POPL*. 359–371.

[72] 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

[73] Julian Mackay, Susan Eisenbach, James Noble, and Sophia Drossopoulou. 2022. Necessity Specifications are Necessary. In *OOPSLA*. ACM.

[74] S. Maffeis, J.C. Mitchell, and A. Taly. 2010. Object Capabilities and Isolation of Untrusted Web Applications. In *Proc of IEEE Security and Privacy*.

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

[76] 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.

[77] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2021. RustHorn: CHC-based Verification for Rust Programs. *TOPLAS* (2021).

[78] 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

[79] Adrian Mettler, David Wagner, and Tyler Close. 2010. Joe-E a Security-Oriented Subset of Java. In *NDSS*.

[80] Bertrand Meyer. 1992. Applying "Design by Contract". *Computer* 25, 10 (1992), 40–51.

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

[82] Mark Samuel Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* Ph. D. Dissertation. Baltimore, Maryland.

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

[84] Mark Samuel Miller, Tom Van Cutsem, and Bill Tulloh. 2013. Distributed Electronic Rights in JavaScript. In *ESOP*.

[85] Mark Samuel Miller, Chip Morningstar, and Bill Frantz. 2000. Capability-based Financial Instruments: From Object to Capabilities. In *Financial Cryptography*. Springer.

[86] 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/.

[87] Nick Mitchell. 2006. The Runtime Structure of Object Ownership. In *ECOOP*. 74–98.

[88] 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.

[89] James H. Morris Jr. 1973. Protection in Programming Languages. *CACM* 16, 1 (1973).

[90] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. 2006. Modular Invariants for Layered Object Structures. *Science of Computer Programming* 62 (2006), 253–286.

[56] 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.
[57] 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.
[58] 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*.
[59] Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language* (2nd ed.).
[60] Alexander Kogtenkov, Bertrand Meyer, and Sergey Velder. 2015. Alias calculus, change calculus and frame inference. *Sci. Comp. Prog.* 97 (2015), 163–172.
[61] 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. doi:10.1145/3586037
[62] Dirk Leinenbach and Thomas Santen. 2009. Verifying the Microsoft Hyper-V Hypervisor with VCC. In *Formal Methods*.
[63] K. Rustan M. Leino and Peter Müller. 2004. Object Invariants in Dynamic Contexts. In *ECOOP*.
[64] K. Rustan M. Leino and Wolfram Schulte. 2007. Using History Invariants to Verify Observers. In *ESOP*.
[65] Henry M. Levy. 1984. *Capability-Based Computer Systems*. Butterworth-Heinemann.
[66] Barbara Liskov and Jeanette Wing. 1994. A Behavioral Notion of Subtyping. *ACM ToPLAS* 16, 6 (1994), 1811–1841.
[67] 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.
[68] Anton Lorenzen, Stephen Dolan, Richard A. Eisenberg, and Sam Lindley. 2024. Oxidizing OCaml with Modal Memory Management. In *ICFP*.
[69] Y. Lu and J. Potter. 2006. Protecting Representation with Effect Encapsulation.. In *POPL*. 359–371.
[70] 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. doi:10.1145/3607846
[71] Julian Mackay, Susan Eisenbach, James Noble, and Sophia Drossopoulou. 2022. Necessity Specifications are Necessary. In *OOPSLA*. ACM.
[72] S. Maffeis, J.C. Mitchell, and A. Taly. 2010. Object Capabilities and Isolation of Untrusted Web Applications. In *Proc of IEEE Security and Privacy*.
[73] Daniel Marshall and Dominic Orchard. 2024. Functional Ownership through Fractional Uniqueness. In *OOPSLA*. doi:10.1145/3649848
[74] 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.
[75] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2021. RustHorn: CHC-based Verification for Rust Programs. *TOPLAS* (2021).
[76] Darya Melicher, Yangqingwei Shi, Alex Potanin, and Jonathan Aldrich. 2017. A Capability-Based Module System for Authority Control. In *ECOOP*. 20:1–20:27. doi:10.4230/LIPIcs.ECOOP.2017.20
[77] Adrian Mettler, David Wagner, and Tyler Close. 2010. Joe-E a Security-Oriented Subset of Java. In *NDSS*.
[78] Bertrand Meyer. 1992. Applying "Design by Contract". *Computer* 25, 10 (1992), 40–51.
[79] B. Meyer. 1992. *Eiffel: The Language*. Prentice Hall.
[80] Mark Samuel Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Ph. D. Dissertation. Baltimore, Maryland.
[81] Mark Samuel Miller. 2011. Secure Distributed Programming with Object-capabilities in JavaScript. (Oct. 2011). Talk at Vrije Universiteit Brussel, mobicrant-talks.eventbrite.com.
[82] Mark Samuel Miller, Tom Van Cutsem, and Bill Tulloh. 2013. Distributed Electronic Rights in JavaScript. In *ESOP*.
[83] Mark Samuel Miller, Chip Morningstar, and Bill Frantz. 2000. Capability-based Financial Instruments: From Object to Capabilities. In *Financial Cryptography*. Springer.
[84] 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/.
[85] Nick Mitchell. 2006. The Runtime Structure of Object Ownership. In *ECOOP*. 74–98.
[86] 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.
[87] James H. Morris Jr. 1973. Protection in Programming Languages. *CACM* 16, 1 (1973).
[88] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. 2006. Modular Invariants for Layered Object Structures. *Science of Computer Programming* 62 (2006), 253–286.

[91] Toby Murray. 2010. *Analysing the Security Properties of Object-Capability Patterns.* Ph. D. Dissertation. University of Oxford.

[92] Takashi Nakayama, Yusuke Matsushita, Ken Sakayori, Ryosuke Sato, and Naoki Kobayashi. 2024. Borrowable Fractional Ownership Types for Verification. In *VMCAI*, Vol. LNCS14500. 224–246.

[93] James Noble, John Potter, and Jan Vitek. 1998. Flexible Alias Protection. In *ECOOP*.

[94] 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

[95] 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.

[96] 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*.

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

[98] 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

[99] 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.

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

[101] 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

[102] 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.

[103] 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.

[104] 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

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

[106] Randall B. Smith and David Ungar. 1995. Programming as an Experience: The Inspiration for Self. In *ECOOP*.

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

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

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

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

[111] David Ungar and Randall B. Smith. 1987. SELF: The Power of Simplicity. In *OOPSLA*.

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

[113] 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.

[114] 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

[115] M. V. Wilkes and R. M. Needham. 1979. The Cambridge CAP computer and its operating system.

[116] 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.

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

[118] 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

[89] Toby Murray. 2010. *Analysing the Security Properties of Object-Capability Patterns.* Ph. D. Dissertation. University of Oxford.
[90] Takashi Nakayama, Yusuke Matsushita, Ken Sakayori, Ryosuke Sato, and Naoki Kobayashi. 2024. Borrowable Fractional Ownership Types for Verification. In *VMCAI*, Vol. LNCS14500. 224–246.
[91] James Noble, John Potter, and Jan Vitek. 1998. Flexible Alias Protection. In *ECOOP*.
[92] Peter W. O'Hearn. 2019. Incorrectness Logic. *Proc. ACM Program. Lang.* 4, POPL, Article 10 (Dec. 2019), 32 pages. doi:10.1145/3371078
[93] 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.
[94] M. Parkinson and G. Bierman. 2005. Separation logic and abstraction. In *POPL*. ACM Press, 247–258.
[95] 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*.
[96] John Potter, James Noble, and David G. Clarke. 1998. The Ins and Outs of Objects. In *Australian Software Engineering Conference*. 80–89.
[97] 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. doi:10.1145/3591265
[98] 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.
[99] J. C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. IEEE Computer Society, 55–74.
[100] Dustin Rhodes, Tim Disney, and Cormac Flanagan. 2014. Dynamic Detection of Object Capability Violations Through Model Checking. In *DLS*. 103–112. doi:10.1145/2661088.2661099
[101] 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.
[102] 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.
[103] 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. doi:10.1007/978-3-030-03418-4_30
[104] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan. 2019. Safer Smart Contract Programming with Scilla. In *OOPSLA*.
[105] Randall B. Smith and David Ungar. 1995. Programming as an Experience: The Inspiration for Self. In *ECOOP*.
[106] Alexander J. Summers and Sophia Drossopoulou. 2010. Considerate Reasoning and the Composite Pattern. In *VMCAI*.
[107] A. J. Summers, S. Drossopoulou, and P. Müller. 2009. A Universe-Type-Based Verification Technique for Mutable Static Fields and Methods. *JOT* (2009).
[108] David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and Compositional Verification of Object Capability Patterns. In *OOPSLA*.
[109] The Ethereum Wiki. 2018. ERC20 Token Standard. (Dec. 2018). https://theethereum.wiki/w/index.php/ERC20_Token_Standard
[110] David Ungar and Randall B. Smith. 1987. SELF: The Power of Simplicity. In *OOPSLA*.
[111] Stephan van Staden. 2015. On Rely-Guarantee Reasoning. In *MPC*, Vol. LNCS9129. 30–49.
[112] 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.
[113] Guannan Wei, Danning Xie, Wuqi Zhang, Yongwei Yuan, and Zhuo Zhang. 2024. Consolidating Smart Contracts with Behavioral Contracts. In *PLDI*. doi:10.1145/3656416
[114] M. V. Wilkes and R. M. Needham. 1979. The Cambridge CAP computer and its operating system.
[115] 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.
[116] Yichen Xu and Martin Odersky. 2024. A Formal Foundation of Reach Capabilities. In *Programming*.
[117] 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. doi:10.1145/3632927

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

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

A.1 Syntax

The syntax of $\mathscr{L}_{ul}$ is given in Fig. 4[11]. To reduce the complexity of our formal models, as is usually done, CITE - CITE, $\mathscr{L}_{ul}$ lacks many common languages features, omitting static fields and methods, interfaces, inheritance, subsumption, exceptions, and control flow. $\mathscr{L}_{ul}$ and which may be defined recursively.

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

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

A.2 Semantics

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

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

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

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

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

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

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

A Appendix to Section 3 – The programming language $\mathscr{L}_{ul}$

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

A.1 Syntax

The syntax of $\mathscr{L}_{ul}$ is given in Fig. 4[11]. To reduce the complexity of our formal models, as is usually done, CITE - CITE, $\mathscr{L}_{ul}$ lacks many common languages features, omitting static fields and methods, interfaces, inheritance, subsumption, exceptions, and control flow. $\mathscr{L}_{ul}$ and which may be defined recursively.

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

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

A.2 Semantics

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

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

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

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

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

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

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

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

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

Module linking is defined for modules with disjoint definitions:

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

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

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

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

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

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

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

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

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

As we already said in §4.2, we forbid assignments to a method's parameters. To do that, the following function returns the identifiers of the formal parameters of the currently active method.

Definition A.6. For program state σ:

$$\begin{array}{lll} Prms(\sigma, \overline{M}) & \triangleq & \overline{x} \;\; \text{such that} \;\; \exists \overline{\phi},\, \phi_k,\, \phi_{k+1},\, C,\, p. \\ & & [\;\; \sigma = (\overline{\phi} \cdot \phi_k \cdot \phi_{k+1}, \chi) \;\;\wedge\;\; \phi_k.\texttt{cont} = \_ := y_0.m(\_);\_ \;\;\wedge \\ & & classOf((\phi_{k+1}, \chi), \texttt{this}) \;\;\wedge\;\; Meth(\overline{M}, C, m) = p\; C :: m(\overline{x : \_})\colon \_\{\_\} \end{array}$$

$$M, \sigma, v \hookrightarrow v \quad \text{(E-Val)} \qquad\qquad M, \sigma, x \hookrightarrow \lfloor x \rfloor_\sigma \quad \text{(E-Var)} \qquad\qquad \frac{M, \sigma, gt \hookrightarrow \alpha}{M, \sigma, gt.f \hookrightarrow \lfloor \alpha.f \rfloor_\sigma} \quad \text{(E-Field)}$$

$$\frac{\begin{array}{c} M, \sigma, gt_0 \hookrightarrow \alpha \\ \overline{M, \sigma, gt \hookrightarrow v} \qquad \texttt{ghost}\; gf(\overline{x : T})\{gt\} : T' \quad ! \in M(classOf(\sigma, \alpha))(\texttt{gflds}) \qquad M, \sigma, [\overline{v/x}]gt \hookrightarrow v \end{array}}{M, \sigma, gt_0.gf(\overline{gt}) \hookrightarrow v} \quad \text{(E-Ghost)}$$

Fig. 9. $\mathscr{L}_{ul}$ ghost term evaluation

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

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

Definition A.7 (Initial Program State)**.** A program state σ is said to be an initial state (*Initial*(σ)) if and only if

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

for some address α and some statement s.

We provide a semantics for expression evaluation is given in Fig. 9. That is, given a module M and a program state σ, expression e evaluates to v if $M, \sigma, e \hookrightarrow v$. Note, the evaluation of expressions is separate from the operational semantics of $\mathscr{L}_{ul}$, and thus there is no restriction on field access.

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

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

Module linking is defined for modules with disjoint definitions:

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

That is, their linking is the union of the two if their domains are disjoint.
Definition A.4 defines the method lookup function for a method call m on an object of class C.

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

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

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

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

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

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

As we already said in §4.2, we forbid assignments to a method's parameters. To do that, the following function returns the identifiers of the formal parameters of the currently active method.

Definition A.6. For program state σ:

$$\begin{array}{lll} Prms(\sigma, \overline{M}) & \triangleq & \overline{x} \quad \text{such that} \quad \exists \overline{\phi},\ \phi_k,\ \phi_{k+1},\ C,\ p. \\ & & [\ \ \sigma = (\overline{\phi} \cdot \phi_k \cdot \phi_{k+1}, \chi) \ \ \wedge\ \ \phi_k.\texttt{cont} = \_ := y_0.m(\_); \_ \ \ \wedge \\ & & classOf((\phi_{k+1}, \chi), \texttt{this}) \ \ \wedge\ \ Meth(\overline{M}, C, m) = p\ C :: m(\overline{x : \_})\colon \_\{\_\} \end{array}$$

$$M, \sigma, v \hookrightarrow v \quad (\text{E-Val}) \qquad\qquad M, \sigma, x \hookrightarrow \lfloor x \rfloor_\sigma \quad (\text{E-Var}) \qquad\qquad \frac{M, \sigma, e \hookrightarrow \alpha}{M, \sigma, e.f \hookrightarrow \lfloor \alpha.f \rfloor_\sigma} \quad (\text{E-Field})$$

$$\frac{M, \sigma, e_0 \hookrightarrow \alpha \qquad \overline{M, \sigma, e \hookrightarrow v} \qquad M(classOf(\sigma, \alpha))\ \text{contains}\ \texttt{ghost}\ gf(\overline{x : T})\{e\} : T' \qquad M, \sigma, [\overline{v/x}]e \hookrightarrow v}{M, \sigma, e_0.gf(\overline{e}) \hookrightarrow v} \quad (\text{E-Ghost})$$

Fig. 9. $\mathscr{L}_{ul}$ ghost term evaluation

While the small-step operational semantics of $\mathscr{L}_{ul}$ is given in Fig. 5, specification satisfaction is defined over an abstracted notion of the operational semantics that models the open world.
An *Initial* program state contains a single frame with a single local variable `this` pointing to a single object in the heap of class `Object`, and a continuation.

Definition A.7 (Initial Program State)**.** A program state σ is said to be an initial state (*Initial*(σ)) if and only if

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

for some address α and some statement s.

We provide a semantics for expression evaluation is given in Fig. 9. That is, given a module M and a program state σ, expression e evaluates to v if $M, \sigma, e \hookrightarrow v$. Note, the evaluation of expressions is separate from the operational semantics of $\mathscr{L}_{ul}$, and thus there is no restriction on field access.

Proof of lemma 4.6 The first assertion is proven by unfolding the definition of $\_ \models \_$.

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

End Proof

We now prove lemma B.2:

Proof of lemma B.2

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

End Proof

Lemma A.8 states that initila states are well-formed, and that (2) a pre-existing object, locally reachable after any number of scoped execution steps, was locally reachable at the first step.

Lemma A.8. For all modules $\overline{M}$, states σ, σ', and frame ϕ:

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

Consider Fig. 3 . Lemma A.8, 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}.

Proof of lemma 4.6 The first assertion is proven by unfolding the definition of $\_ \models \_$.

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

End Proof

We now prove lemma B.2:

Proof of lemma B.2

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

End Proof

Lemma A.8 states that initila states are well-formed, and that (2) a pre-existing object, locally reachable after any number of scoped execution steps, was locally reachable at the first step.

Lemma A.8. For all modules $\overline{M}$, states σ, σ', and frame ϕ:

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

Consider Fig. 3 . Lemma A.8, 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}.

B APPENDIX TO SECTION 4 – FUNDAMENTAL CONCEPTS

Lemma B.1 says, essentially, that scoped executions describe the same set of executions as those starting at an initial state[12]. For instance, revisit Fig. 3 , 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 $\overline{M};\ \sigma_6 \rightsquigarrow^* \sigma_{14}$.

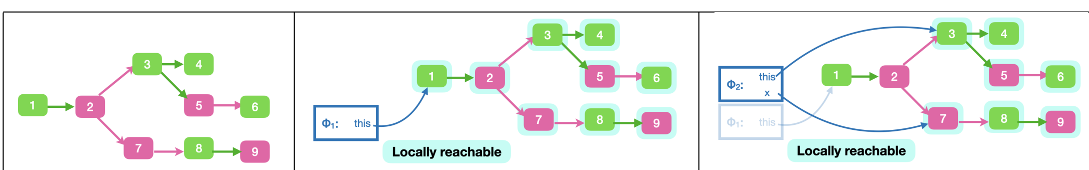
Lemma B.1. For all modules $\overline{M}$, state σ_{init}, σ, σ', where σ_{init} is initial:

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

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

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

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



a heap	Locally Reachable from ϕ_1	Locally Reachable from ϕ_2

Fig. 10. -Locally Reachable Objects

Fig. 10 illustrates local reachability: In the middle pane the top frame is ϕ_1 which maps `this` to o_1; all objects are locally reachable. In the right pane the top frame is ϕ_2, which maps `this` to o_3, and x to o_7; now o_1 and o_2 are no longer locally reachable.

Proof of lemma B.1

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

End Proof

Proof of lemma B.2

- We unfolding the definition of $\overline{M};\ \sigma \rightsquigarrow \sigma'$ $\overline{M};\ \sigma \rightsquigarrow \sigma'$ and the rules of the operational semantics.
- Take $k = |\sigma|$. We unfold the definition from 4.2, and obtain that $\sigma = \sigma'$ or, $\exists \sigma_1, ...\sigma_{n1}.\forall i \in [1..n)[\ \overline{M}; \sigma_i \dashrightarrow \sigma_{i+1}\ \wedge\ |\sigma_1| \leq |\sigma_{i+1}|\ \wedge \sigma = \sigma_1\ \wedge\ \sigma' = \sigma_n\ ]$
  Consider the second case. Take any $i \in [1..n)$. Then, by Definition, $k \leq |\sigma|$. If $k = |\sigma_i|$, then we are executing part of $\sigma.prgcont$, and because $y \notin Vs(\sigma.\texttt{cont})$, we get $\lfloor y \rfloor_{\sigma[i]} = \lfloor y \rfloor_{\sigma_{i+1}[k]}$. If $k = |\sigma_i|$, then we apply the bullet from above, and also obtain $\lfloor y \rfloor_{\sigma[i]} = \lfloor y \rfloor_{\sigma_{i+1}[k]}$
  This gives that $\lfloor y \rfloor_{\sigma[k]} = \lfloor y \rfloor_{\sigma'[k]}$. Moreover, because $\overline{M};\ \sigma \rightsquigarrow^*_{fin} \sigma'$ we obtain that $|\sigma| = |\sigma'| = k$. Therefore, we have that $\lfloor y \rfloor_{\sigma} = \lfloor y \rfloor_{\sigma'}$.

[12] 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)

B Appendix to Section 4 – Fundamental Concepts

Lemma B.1 says, essentially, that scoped executions describe the same set of executions as those starting at an initial state[12]. For instance, revisit Fig. 3 , and assume that σ_6 is an initial state. We have $\overline{M}; \sigma_{10} \dashrightarrow^* \sigma_{14}$ and $\overline{M};\ \sigma_{10} \not\leadsto^* \sigma_{14}$, but also $\overline{M};\ \sigma_6 \leadsto^* \sigma_{14}$.

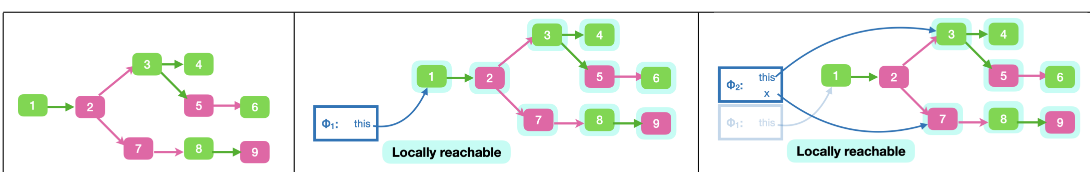
Lemma B.1. For all modules $\overline{M}$, state σ_{init}, σ, σ', where σ_{init} is initial:

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

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

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

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



a heap	Locally Reachable from ϕ_1	Locally Reachable from ϕ_2

Fig. 10. -Locally Reachable Objects

Fig. 10 illustrates local reachability: In the middle pane the top frame is ϕ_1 which maps `this` to o_1; all objects are locally reachable. In the right pane the top frame is ϕ_2, which maps `this` to o_3, and x to o_7; now o_1 and o_2 are no longer locally reachable.

Proof of lemma B.1

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

End Proof

Proof of lemma B.2

- We unfolding the definition of $\overline{M};\ \sigma \leadsto \sigma'\ \overline{M};\ \sigma \leadsto \sigma'$ and the rules of the operational semantics.
- Take $k = |\sigma|$. We unfold the definition from 4.2, and obtain that $\sigma = \sigma'$ or, $\exists \sigma_1, ...\sigma_{n1}.\forall i \in [1..n)[\ \overline{M}; \sigma_i \dashrightarrow \sigma_{i+1} \ \wedge\ |\sigma_1| \leq |\sigma_{i+1}| \ \wedge \sigma = \sigma_1 \ \wedge\ \sigma' = \sigma_n\ ]$
  Consider the second case. Take any $i \in [1..n)$. Then, by Definition, $k \leq |\sigma|$. If $k = |\sigma_i|$, then we are executing part of $\sigma.prgcont$, and because $y \notin Vs(\sigma.\texttt{cont})$, we get $\lfloor y \rfloor_{\sigma[i]} = \lfloor y \rfloor_{\sigma_{i+1}[k]}$. If $k = |\sigma_i|$, then we apply the bullet from above, and also obtain $\lfloor y \rfloor_{\sigma[i]} = \lfloor y \rfloor_{\sigma_{i+1}[k]}$
  This gives that $\lfloor y \rfloor_{\sigma[k]} = \lfloor y \rfloor_{\sigma'[k]}$. Moreover, because $\overline{M};\ \sigma \leadsto^*_{fin} \sigma'$ we obtain that $|\sigma| = |\sigma'| = k$. Therefore, we have that $\lfloor y \rfloor_{\sigma} = \lfloor y \rfloor_{\sigma'}$.

[12]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)

End Proof

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

Lemma B.3. $\models \sigma \quad \land k < |\sigma| \quad \Longrightarrow \quad LocRchbl(\sigma[k+1]) \subseteq LocRchbl(\sigma[k])$

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

□

Proof of lemma 4.6

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

End Proof

End Proof

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

Lemma B.3. $\models \sigma \quad \wedge k < |\sigma| \quad \Longrightarrow \quad LocRchbl(\sigma[k+1]) \subseteq LocRchbl(\sigma[k])$

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

□

Proof of lemma 4.6

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

End Proof

C APPENDIX TO SECTION 6 – PRESERVATION OF SATISFACTION

Proof of lemma 6.1

Take any M A, σ

(1) To show that $M, \sigma \models A \quad \Longleftrightarrow \quad M, \sigma \models A[\lfloor x \rfloor_\sigma / x]$
The proof goes by induction on the structure of A, application of Defs. 5.3, 5.4, and 5.4, and auxiliary lemma **??**.

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

End Proof

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

Lemma C.1. $M, \sigma, e \hookrightarrow \alpha \quad \Longrightarrow \quad [\ M, \sigma \models A \quad \Longleftrightarrow \quad M, \sigma \models A[\alpha/e]\ \ ]$

Proof. by induction on the structure of A, application of Defs. 5.3, 5.4, and 5.4. □

C.1 Stability

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

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

$$Stbl(\langle\!\langle e \rangle\!\rangle) \triangleq false \qquad Stbl(\langle\!\langle e \rangle\!\rangle \leftrightarrow\!\!\!\!\!\times\ \overline{u}) = Stbl(e : \texttt{intl}) = Stbl(e) = Stbl(e : C) \ \triangleq\ true$$
$$Stbl(A_1 \wedge A_2) \ \triangleq\ Stbl(A_1) \wedge Stbl(A_2) \qquad Stbl(\forall x : C.A) = Stbl(\neg A) \ \triangleq\ Stbl(A)$$

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

$$Stb^+(\langle\!\langle e \rangle\!\rangle) = Stb^+(\langle\!\langle e \rangle\!\rangle \leftrightarrow\!\!\!\!\!\times\ \overline{u}) = Stb^+(e : \texttt{intl}) = Stb^+(e) = Stb^+(e : C) \ \triangleq\ true$$
$$Stb^+(A_1 \wedge A_2) \ \triangleq\ Stb^+(A_1) \wedge Stb^+(A_2) \qquad Stb^+(\forall x : C.A) \triangleq Stb^+(A) \qquad Stb^+(\neg A) \triangleq Stbl(A)$$

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

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

- To show
  $Stbl(A) \ \wedge\ Fv(A) = \emptyset \quad \Longrightarrow \quad [\ \ M, \sigma \models A \quad \Longleftrightarrow \quad M, \sigma \triangledown \phi \models A\ \ ]$
  By induction on the structure of the definition of $Stbl(A)$.
- To show
  $Stb^+(A) \wedge Fv(A) = \emptyset \wedge M{\cdot}\overline{M} \models \sigma \triangledown \phi \wedge M, \sigma \models A \wedge M, \sigma \triangledown \phi \models \texttt{intl} \quad \Longrightarrow \quad M, \sigma \triangledown \phi \models A\ ]$
  By induction on the structure of the definition of $Stb^+(A)$. The only interesting case is when A has the form $\langle\!\langle e \rangle\!\rangle$. Because $fv(A) = \emptyset$, we know that $\lfloor e \rfloor_\sigma = \lfloor e \rfloor_{\sigma \triangledown \phi}$. Therefore, we assume that $\lfloor e \rfloor_\sigma = \alpha$ for some α, assume that $M, \sigma \models \langle\!\langle \alpha \rangle\!\rangle$, and want to show that $M, \sigma \triangledown \phi \models \langle\!\langle \alpha \rangle\!\rangle$. From $M \cdot \overline{M} \models \sigma \triangledown \phi$ we obtain that $Rng(\phi) \subseteq Rng(\sigma)$. From this, we obtain that $LocRchbl(\sigma \triangledown \phi) \subseteq LocRchbl(\sigma)$. The rest follows by unfolding and folding Def. 5.4.

End Proof

C.2 Encapsulation

Proofs of adherence to $\mathscr{L}^{spec}$ specifications hinge on the expectation that some, specific, assertions cannot be invalidated unless some internal (and thus known) computation took place. We call such assertions *encapsulated*. We define the judgement, $M \ \vdash Enc(A)$, in terms of the judgment $M; \Gamma \vdash Enc(A); \Gamma'$ which checks that any objets read in the validation of A are internaml. We assume a judgment $M; \Gamma \vdash e : \texttt{intl}$ which says that in the context of Γ, the expression e belongs to a class

C Appendix to Section 6 – Preservation of Satisfaction

Proof of lemma 6.1

Take any $M\ A, \sigma$

(1) To show that $M, \sigma \models A \quad \Longleftrightarrow \quad M, \sigma \models A[\lfloor x \rfloor_\sigma / x]$
The proof goes by induction on the structure of A, application of Defs. 5.3, 5.4, and 5.4, and auxiliary lemma ??.

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

End Proof

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

Lemma C.1. $M, \sigma, e \hookrightarrow \alpha \quad \Longrightarrow \quad [\ M, \sigma \models A \ \Longleftrightarrow \ M, \sigma \models A[\alpha/e]\ ]$

Proof. by induction on the structure of A, application of Defs. 5.3, 5.4, and 5.4. □

C.1 Stability

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

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

$$Stbl(\langle\!\langle e \rangle\!\rangle) \triangleq false \qquad Stbl(\langle\!\langle e \rangle\!\rangle \nleftrightarrow \overline{u}) = Stbl(e : \texttt{intl}) = Stbl(e) = Stbl(e : C) \ \triangleq \ true$$
$$Stbl(A_1 \wedge A_2) \ \triangleq \ Stbl(A_1) \wedge Stbl(A_2) \qquad Stbl(\forall x : C.A) = Stbl(\neg A) \ \triangleq \ Stbl(A)$$

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

$$Stb^+(\langle\!\langle e \rangle\!\rangle) = Stb^+(\langle\!\langle e \rangle\!\rangle \nleftrightarrow \overline{u}) = Stb^+(e : \texttt{intl}) = Stb^+(e) = Stb^+(e : C) \ \triangleq \ true$$
$$Stb^+(A_1 \wedge A_2) \ \triangleq \ Stb^+(A_1) \wedge Stb^+(A_2) \qquad Stb^+(\forall x : C.A) \triangleq \ Stb^+(A) \qquad Stb^+(\neg A) \triangleq Stbl(A)$$

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

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

- To show
$Stbl(A) \ \wedge \ Fv(A) = \emptyset \quad \Longrightarrow \quad [\ \ M, \sigma \models A \ \ \Longleftrightarrow \ \ M, \sigma \triangledown \phi \models A \ \ ]$
By induction on the structure of the definition of $Stbl(A)$.
- To show
$Stb^+(A) \wedge Fv(A) = \emptyset \wedge M \cdot \overline{M} \models \sigma \triangledown \phi \ \wedge \ M, \sigma \models A \wedge M, \sigma \triangledown \phi \models \texttt{intl} \ \Longrightarrow \ M, \sigma \triangledown \phi \models A\ ]$
By induction on the structure of the definition of $Stb^+(A)$. The only interesting case is when A has the form $\langle\!\langle e \rangle\!\rangle$. Because $fv(A) = \emptyset$, we know that $\lfloor e \rfloor_\sigma = \lfloor e \rfloor_{\sigma \triangledown \phi}$. Therefore, we assume that $\lfloor e \rfloor_\sigma = \alpha$ for some α, assume that $M, \sigma \models \langle\!\langle \alpha \rangle\!\rangle$, and want to show that $M, \sigma \triangledown \phi \models \langle\!\langle \alpha \rangle\!\rangle$. From $M \cdot \overline{M} \models \sigma \triangledown \phi$ we obtain that $Rng(\phi) \subseteq Rng(\sigma)$. From this, we obtain that $LocRchbl(\sigma \triangledown \phi) \subseteq LocRchbl(\sigma)$. The rest follows by unfolding and folding Def. 5.4.

End Proof

C.2 Encapsulation

Proofs of adherence to $\mathscr{L}^{spec}$ specifications hinge on the expectation that some, specific, assertions cannot be invalidated unless some internal (and thus known) computation took place. We call such assertions *encapsulated*. We define the judgement, $M \ \vdash Enc(A)$, in terms of the judgment $M; \Gamma \vdash Enc(A); \Gamma'$ which checks that any objets read in the validation of A are internaml. We assume from M. We also assume that the judgement $M;\Gamma \vdash e : \texttt{intl}$ can deal with ghistfields – eg through appropriate annotations of the ghost methods. Note that it is possible for $M;\Gamma \vdash Enc(e)$ to hold and $M;\Gamma \vdash e : \texttt{intl}$ not to hold.

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

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

Fig. 11. The judgment $M;\Gamma \vdash Enc(A);\Gamma'$

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

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

- $M \vdash Enc(A) \quad \triangleq \quad \exists\Gamma.[\ M;\emptyset \vdash Enc(A);\Gamma\ ]$ as defined in Fig. 11.

More on Def. 6.4 If the definition 6.4 used the more general execution, $M\cdot\overline{M};\sigma \dashrightarrow \sigma'$, rather than the scoped execution, $M\cdot\overline{M};\ \sigma \rightsquigarrow \sigma'$, then fewer assertions would have been encapsulated. Namely, assertions like $\langle\!\langle x.f \rangle\!\rangle$ would not be encapsulated. Consider, *e.g.*, a heap χ, with objects 1, 2, 3 and 4, where 1, 2 are external, and 3, 4 are internal, and 1 has fields pointing to 2 and 4, and 2 has a field pointing to 3, and 3 has a field f pointing to 4. Take state σ=$(\phi_1\cdot\phi_2, \chi)$, where ϕ_1's receiver is 1, ϕ_2's receiver is 2, and there are no local variables. We have ...$\sigma \models \texttt{extl} \wedge \langle\!\langle 3.f \rangle\!\rangle$. We return from the most recent all, getting ...; $\sigma \dashrightarrow \sigma'$ where $\sigma' = (\phi_1, \chi)$; and have ..., $\sigma' \not\models \langle\!\langle 3.f \rangle\!\rangle$.

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

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

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

Proof of lemma 6.5 By induction on the definition of the judgment $\_ \vdash Enc(\_)$, and then case analysis on program execution **End Proof**

a judgment $M; \Gamma \vdash e : \texttt{intl}$ which says that in the context of Γ, the expression e belongs to a class from M. We also assume that the judgement $M; \Gamma \vdash e : \texttt{intl}$ can deal with ghistfields – eg through appropriate annotations of the ghost methods. Note that it is possible for $M; \Gamma \vdash Enc(e)$ to hold and $M; \Gamma \vdash e : \texttt{intl}$ not to hold.

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

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

Fig. 11. The judgment $M; \Gamma \vdash Enc(A); \Gamma'$

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

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

- $M \vdash Enc(A) \;\triangleq\; \exists \Gamma.[\, M; \emptyset \vdash Enc(A); \Gamma \,]$ as defined in Fig. 11.

More on Def. 6.4 If the definition 6.4 used the more general execution, $M \cdot \overline{M}; \sigma \dashrightarrow \sigma'$, rather than the scoped execution, $M \cdot \overline{M};\ \sigma \rightsquigarrow \sigma'$, then fewer assertions would have been encapsulated. Namely, assertions like $\langle\!\langle x.f \rangle\!\rangle$ would not be encapsulated. Consider, *e.g.*, a heap χ, with objects 1, 2, 3 and 4, where 1, 2 are external, and 3, 4 are internal, and 1 has fields pointing to 2 and 4, and 2 has a field pointing to 3, and 3 has a field f pointing to 4. Take state σ=$(\phi_1 \cdot \phi_2, \chi)$, where ϕ_1's receiver is 1, ϕ_2's receiver is 2, and there are no local variables. We have ...$\sigma \models \texttt{extl} \wedge \langle\!\langle 3.f \rangle\!\rangle$. We return from the most recent all, getting ...; $\sigma \dashrightarrow \sigma'$ where $\sigma' = (\phi_1, \chi)$; and have ..., $\sigma' \not\models \langle\!\langle 3.f \rangle\!\rangle$.

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

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

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

Proof of lemma 6.5 By induction on the definition of the judgment $\_ \vdash Enc(\_)$, and then case analysis on program execution **End Proof**

D APPENDIX TO SECTION 7 – SPECIFICATIONS

Definition D.1 (Specifications Well-formed). *Well-formedness*, $\vdash S$, is defined by cases on S:

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

Example D.2 (Badly Formed Method Specifications). S_{9,bad_1} is not a well-formed specification, because A' is not a formal parameter, nor free in the precondition.

$$
\begin{array}{lll}
S_{9,bad\_1} & \triangleq & \{\, a:\mathtt{Account} \wedge \langle\!\langle a \rangle\!\rangle \,\} \\
& & \quad \mathtt{public\ Account::set(key'}:\mathtt{Key)} \\
& & \{\, \langle\!\langle a \rangle\!\rangle \wedge \langle\!\langle a'.\mathtt{key} \rangle\!\rangle \,\} \ \parallel\ \{true\} \\
S_{9,bad\_2} & \triangleq & \{\, a:\mathtt{Account} \wedge \langle\!\langle a \rangle\!\rangle \,\} \\
& & \quad \mathtt{public\ Account::set(key'}:\mathtt{Key)} \\
& & \{\, \langle\!\langle a \rangle\!\rangle \wedge \langle\!\langle a'.\mathtt{key} \rangle\!\rangle \,\} \ \parallel\ \{\mathtt{this.blnce}\,\}
\end{array}
$$

Example D.3 (More Method Specifications). S_7 below guarantees that `transfer` does not affect the balance of accounts different from the receiver or argument, and if the key supplied is not that of the receiver, then no account's balance is affected. S_8 guarantees that if the key supplied is that of the receiver, the correct amount is transferred from the receiver to the destination. S_9 guarantees that `set` preserves the protectedness of a key.

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

D.1 Examples of Semantics of our Specifications

Example D.4. We revisit the specifications given in Sect. 2.1, the three modules from Sect. 2.1.3, and Example D.3

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

Example D.5. For Example 7.6, we have $\mathtt{M}_{good} \models S_7$ and $\mathtt{M}_{bad} \models S_7$ and $\mathtt{M}_{fine} \models S_7$. Also, $\mathtt{M}_{good} \models S_8$ and $\mathtt{M}_{bad} \models S_8$ and $\mathtt{M}_{fine} \models S_8$. However, $\mathtt{M}_{good} \models S_9$, while $\mathtt{M}_{bad} \not\models S_9$.

Example D.6. For any specification $S \triangleq \{\, A \,\}\, p\ C\!::\!m(\overline{x:C})\,\{\, A' \,\}$ and any module M which does not have a class C with a method m with formal parameter types $\overline{C}$, we have that $M \models S$. Namely, if a method were to be called with that signature on a C from M, then execution would be stuck, and the requirements from Def. 7.4(3) would be trivially satisfied. Thus, $\mathtt{M}_{fine} \models S_8$.

D Appendix to Section 7 – Specifications

Definition D.1 (Specifications Well-formed). *Well-formedness*, $\vdash S$, is defined by cases on S:

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

Example D.2 (Badly Formed Method Specifications). S_{9,bad_1} is not a well-formed specification, because A' is not a formal parameter, nor free in the precondition.

$$
\begin{aligned}
S_{9,bad\_1} \quad \triangleq \quad & \{\, a : \texttt{Account} \land \langle\!\langle a \rangle\!\rangle \,\} \\
& \quad \texttt{public Account} :: \texttt{set}(\texttt{key}' : \texttt{Key}) \\
& \{\, \langle\!\langle a \rangle\!\rangle \land \langle\!\langle a'.\texttt{key} \rangle\!\rangle \,\} \ \parallel \ \{true\} \\
S_{9,bad\_2} \quad \triangleq \quad & \{\, a : \texttt{Account} \land \langle\!\langle a \rangle\!\rangle \,\} \\
& \quad \texttt{public Account} :: \texttt{set}(\texttt{key}' : \texttt{Key}) \\
& \{\, \langle\!\langle a \rangle\!\rangle \land \langle\!\langle a'.\texttt{key} \rangle\!\rangle \,\} \ \parallel \ \{\texttt{this.blnce}\,\}
\end{aligned}
$$

Example D.3 (More Method Specifications). S_7 below guarantees that `transfer` does not affect the balance of accounts different from the receiver or argument, and if the key supplied is not that of the receiver, then no account's balance is affected. S_8 guarantees that if the key supplied is that of the receiver, the correct amount is transferred from the receiver to the destination. S_9 guarantees that `set` preserves the protectedness of a key.

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

D.1 Examples of Semantics of our Specifications

Example D.4. We revisit the specifications given in Sect. 2.1, the three modules from Sect. 2.1.3, and Example D.3

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

Example D.5. For Example 7.6, we have $\mathbb{M}_{good} \models S_7$ and $\mathbb{M}_{bad} \models S_7$ and $\mathbb{M}_{fine} \models S_7$. Also, $\mathbb{M}_{good} \models S_8$ and $\mathbb{M}_{bad} \models S_8$ and $\mathbb{M}_{fine} \models S_8$. However, $\mathbb{M}_{good} \models S_9$, while $\mathbb{M}_{bad} \not\models S_9$.

Example D.6. For any specification $S \triangleq \{\, A \,\}\, p\ C :: m(\overline{x : C})\, \{\, A' \,\}$ and any module M which does not have a class C with a method m with formal parameter types $\overline{C}$, we have that $M \models S$. Namely, if a method were to be called with that signature on a C from M, then execution would be stuck, and the requirements from Def. 7.4(3) would be trivially satisfied. Thus, $\mathbb{M}_{fine} \models S_8$.

E EXPRESSIVENESS

We argue the expressiveness of our approach by comparing with example specifications proposed in [73].

E.1 The DOM

This is the motivating example in [32], dealing with a tree of DOM nodes: Access to a DOM node gives access to all its `parent` and `children` nodes, with the ability to modify the node's `property` – where `parent`, `children` and `property` are fields in class `Node`. Since the top nodes of the tree usually contain privileged information, while the lower nodes contain less crucial third-party information, we must be able to limit access given to third parties to only the lower part of the DOM tree. We do this through a `Proxy` class, which has a field `node` pointing to a `Node`, and a field `height`, which restricts the range of `Node`s which may be modified through the use of the particular `Proxy`. Namely, when you hold a `Proxy` you can modify the `property` of all the descendants of the `height`-th ancestors of the `node` of that particular `Proxy`. We say that `pr` has *modification-capabilities* on `nd`, where `pr` is a `Proxy` and `nd` is a `Node`, if the `pr.height`-th `parent` of the node at `pr.node` is an ancestor of `nd`.

We specify this property as follows:

$$
\begin{aligned}
S_{dom\_1} \;&\triangleq\; \forall nd : \texttt{DomNode}.\{\ \forall pr : \texttt{Proxy}.[\ may\_modify(pr, nd) \longrightarrow \langle\!\langle pr \rangle\!\rangle\ ]\ \}\\
S_{dom\_2} \;&\triangleq\; \forall nd : \texttt{DomNode}, val : \texttt{PropertyValue}.\\
&\qquad\qquad \{\ \forall pr : \texttt{Proxy}.[\ may\_modify(pr, nd) \longrightarrow \langle\!\langle pr \rangle\!\rangle\ ] \wedge nd.property = val\ \}
\end{aligned}
$$

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

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

In [73] this was specified as follows:

```
DOMSpec ≜ from nd : Node ∧ nd.property = p  to nd.property != p
          onlyIf ∃ o.[ o:extl ∧
                        ( ∃ nd':Node.[ o access nd'⟩ ]  ∨
                          ∃ pr:Proxy,k:ℕ.[o access pr⟩ ∧ nd.parent^k=pr.node.parent^pr.height
```

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

E.2 DAO

The Decentralized Autonomous Organization (DAO) [23] is a well-known Ethereum contract allowing participants to invest funds. The DAO famously was exploited with a re-entrancy bug in 2016, and lost $50M. Here we provide specifications that would have secured the DAO against such a bug.

$$
\begin{aligned}
S_{dao\_1} \;&\triangleq\; \forall d : \texttt{DAO}.\{\ \forall p : \texttt{Participant}.[\ d.ether \geq d.balance(p)\ ]\ \}\\
S_{dao\_2} \;&\triangleq\; \forall d : \texttt{DAO}.\{\ \ d.ether \geq \textstyle\sum_{p \in d.particiants} d.balance(p)\ \ \}
\end{aligned}
$$

The specifications above say the following:

- S_{edao_1} guarantees that the DAO holds more ether than the balance of any of its participant's.
- S_{dao_2} guarantees that that the DAO holds more ether than the sum of the balances held by DAO's participants.

S_{dao_2} is stronger than S_{dao_1}. They would both have precluded the DAO bug. Note that these specifications do not mention capabilities. They are, essentially, simple class invariants and could

D.2 Expressiveness

We argue the expressiveness of our approach by comparing with example specifications proposed in [71].

D.3 The DOM

This is the motivating example in [31], dealing with a tree of DOM nodes: Access to a DOM node gives access to all its `parent` and `children` nodes, with the ability to modify the node's `property` – where `parent`, `children` and `property` are fields in class `Node`. Since the top nodes of the tree usually contain privileged information, while the lower nodes contain less crucial third-party information, we must be able to limit access given to third parties to only the lower part of the DOM tree. We do this through a `Proxy` class, which has a field `node` pointing to a `Node`, and a field `height`, which restricts the range of `Node`s which may be modified through the use of the particular `Proxy`. Namely, when you hold a `Proxy` you can modify the `property` of all the descendants of the `height`-th ancestors of the `node` of that particular `Proxy`. We say that `pr` has *modification-capabilities* on `nd`, where `pr` is a `Proxy` and `nd` is a `Node`, if the `pr.height`-th `parent` of the node at `pr.node` is an ancestor of `nd`.

We specify this property as follows:

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

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

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

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

In [71] this was specified as follows:

```
DOMSpec ≜ from nd : Node ∧ nd.property = p  to nd.property != p
          onlyIf ∃ o.[ o:extl ∧
                      ( ∃ nd':Node.[ o access nd'⟩ ]  ∨
                        ∃ pr:Proxy,k:ℕ.[o access pr⟩ ∧ nd.parent^k=pr.node.parent^pr.height
```

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

D.4 DAO

The Decentralized Autonomous Organization (DAO) [23] is a well-known Ethereum contract allowing participants to invest funds. The DAO famously was exploited with a re-entrancy bug in 2016, and lost \$50M. Here we provide specifications that would have secured the DAO against such a bug.

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

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

The specifications above say the following:

S_{edao_1} guarantees that the DAO holds more ether than the balance of any of its participant's.

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

S_{dao_2} is stronger than S_{dao_1}. They would both have precluded the DAO bug. Note that these specifications do not mention capabilities. They are, essentially, simple class invariants and could have been expressed with the techniques proposed already by [80]. The only difference is that S_{dao_1} and S_{dao_2} are two-state invariants, which means that we require that they are *preserved*, *i.e.* if they hold in one (observable) state they have to hold in all successor states, while class invariants are one-state, which means they are required to hold in all (observable) states. [13]

We now compare with the specification given in [73]. `DAOSpec1` in similar to S_{dao_1}: iy says that no participant's balance may ever exceed the ether remaining in DAO. It is, essentially, a one-state invariant.

```
DAOSpec1 ≜ from d : DAO ∧ p : Object
            to d.balance(p) > d.ether
            onlyIf false
```

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

[73] gives one more specification:

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

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

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

E.3 ERC20

The ERC20 [110] is a widely used token standard describing the basic functionality of any Ethereum-based token contract. This functionality includes issuing tokens, keeping track of tokens belonging to participants, and the transfer of tokens between participants. Tokens may only be transferred if there are sufficient tokens in the participant's account, and if either they (using the `transfer` method) or someone authorised by the participant (using the `transferFrom` method) initiated the transfer.

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

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

We specify the guarantees of ERC20 as follows:

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

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

[13]This should have been explained somewhere earlier.

have been expressed with the techniques proposed already by [78]. The only difference is that S_{dao_1} and S_{dao_2} are two-state invariants, which means that we require that they are *preserved*, *i.e.* if they hold in one (observable) state they have to hold in all successor states, while class invariants are one-state, which means they are required to hold in all (observable) states. [13]

We now compare with the specification given in [71]. `DAOSpec1` in similar to S_{dao_1}: iy says that no participant's balance may ever exceed the ether remaining in DAO. It is, essentially, a one-state invariant.

```
DAOSpec1 ≜ from d : DAO ∧ p : Object
            to d.balance(p) > d.ether
            onlyIf false
```

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

[71] gives one more specification:

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

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

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

D.5 ERC20

The ERC20 [109] is a widely used token standard describing the basic functionality of any Ethereum-based token contract. This functionality includes issuing tokens, keeping track of tokens belonging to participants, and the transfer of tokens between participants. Tokens may only be transferred if there are sufficient tokens in the participant's account, and if either they (using the `transfer` method) or someone authorised by the participant (using the `transferFrom` method) initiated the transfer.

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

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

We specify the guarantees of ERC20 as follows:

$$S_{erc\_1} \triangleq \mathbb{\forall} e : \texttt{ERC20}, p : \texttt{Participant}.\{\ e.allowed(p, p)\ \}$$
$$S_{erc\_2} \triangleq \mathbb{\forall} e : \texttt{ERC20}, p, p' : \texttt{Participant}, n : \mathbb{N}. \quad \{\ \forall p'.[\ (e.allowed(p', p) \longrightarrow \langle\!\langle p' \rangle\!\rangle\ ]\ \}.\{\ e.balance(b) = n\ \}$$

[13]This should have been explained somewhere earlier.

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

The specifications above say the following:

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

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

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

We compare with the specifications given in [73]: Firstly, `ERC20Spec1` says that if the balance of a participant's account is ever reduced by some amount m, then that must have occurred as a result of a call to the `transfer` method with amount m by the participant, or the `transferFrom` method with the amount m by some other participant.

```
ERC20Spec1 ≜ from e : ERC20 ∧ e.balance(p) = m + m' ∧ m > 0
              next e.balance(p) = m'
              onlyIf ∃ p' p''.[⟨p' calls e.transfer(p, m)⟩ ∨
                     e.allowed(p, p'') ≥ m ∧ ⟨p″ calls e.transferFrom(p', m)⟩]
```

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

```
ERC20Spec2 ≜ from e : ERC20 ∧ p : Object ∧ p' : Object ∧ m : Nat
              next e.allowed(p, p') = m
              onlyIf ⟨p calls e.approve(p', m)⟩ ∨
                     (e.allowed(p, p') = m ∧
                      ¬ (⟨p' calls e.transferFrom(p, _)⟩ ∨
                             ⟨p calls e.allowed(p, _)⟩)) ∨
                     ∃ p''. [e.allowed(p, p') = m + m' ∧ ⟨p' calls e.transferFrom(p″, m')⟩]
```

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

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

The specifications above say the following:

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

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

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

We compare with the specifications given in [71]: Firstly, `ERC20Spec1` says that if the balance of a participant's account is ever reduced by some amount m, then that must have occurred as a result of a call to the `transfer` method with amount m by the participant, or the `transferFrom` method with the amount m by some other participant.

```
ERC20Spec1 ≜ from e : ERC20 ∧ e.balance(p) = m + m' ∧ m > 0
              next e.balance(p) = m'
              onlyIf ∃ p' p''.[⟨p' calls e.transfer(p, m)⟩ ∨
                     e.allowed(p, p'') ≥ m ∧ ⟨p″ calls e.transferFrom(p′, m)⟩]
```

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

```
ERC20Spec2 ≜ from e : ERC20 ∧ p : Object ∧ p' : Object ∧ m : Nat
              next e.allowed(p, p') = m
              onlyIf ⟨p calls e.approve(p′, m)⟩ ∨
                     (e.allowed(p, p') = m ∧
                      ¬ (⟨p′ calls e.transferFrom(p, _)⟩ ∨
                            ⟨p calls e.allowed(p, _)⟩)) ∨
                     ∃ p''. [e.allowed(p, p') = m + m' ∧ ⟨p′ calls e.transferFrom(p″, m′)⟩]
```

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

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

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

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

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

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

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

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

Example F.2. The specification from Example 7.6 can be renamed as

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

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

F.2 Types

The rules in Fig. 12 allow triples to talk about the types Rule TYPES-1 promises that types of local variables do not change. Rule TYPES-2 generalizes TYPES-1 to any statement, provided that there already exists a triple for that statement.

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

F.3 Second Phase - more

in Fig. 13, we extend the Hoare Quadruples Logic with substructural rules, rules for conditionals, case analysis, and a contradiction rule. For the conditionals we assume the obvious operational. semantics, but do not define it in this paper

E Appendix to Section 8 – Proving Open Calls and Adherence to $\mathscr{L}^{spec}$ Specifications

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

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

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

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

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

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

Example E.2. The specification from Example 7.6 can be renamed as

$$
\begin{array}{rll}
S_{9r} & \triangleq & \{\, a1 : \texttt{Account}, a2 : \texttt{Account} \wedge \langle\!\langle a1 \rangle\!\rangle \wedge \langle\!\langle a2.\texttt{key} \rangle\!\rangle \,\} \\
& & \quad \texttt{public Account :: set(nKey : Key)} \\
& & \{\, \langle\!\langle a1 \rangle\!\rangle \wedge \langle\!\langle a2.\texttt{key} \rangle\!\rangle \,\} \ \parallel\ \{\, \langle\!\langle a1 \rangle\!\rangle \wedge \langle\!\langle a2.\texttt{key} \rangle\!\rangle \,\}
\end{array}
$$

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

E.2 Types

The rules in Fig. 12 allow triples to talk about the types Rule TYPES-1 promises that types of local variables do not change. Rule TYPES-2 generalizes TYPES-1 to any statement, provided that there already exists a triple for that statement.

TYPES-1

$$
\frac{stmt \text{ contains no method call} \qquad stmt \text{ contains no assignment to } x}{M \vdash \{\, x : C \,\}\ stmt\ \{\, x : C \,\}}
$$

TYPES-2

$$
\frac{M \vdash \{\, A \,\}\ s\ \{\, A' \,\} \ \parallel\ \{\, A'' \,\}}{M \vdash \{\, x : C \wedge A \,\}\ s\ \{\, x : C \wedge A' \,\} \ \parallel\ \{\, A'' \,\}}
$$

Fig. 12. Types

TYPES-1

$$\frac{stmt \text{ contains no method call} \qquad stmt \text{ contains no assignment to } x}{M \vdash \{\, x : C \,\}\ stmt\ \{\, x : C \,\}}$$

TYPES-2

$$\frac{M \vdash \{\, A \,\}\ s\ \{\, A' \,\}\ \parallel\ \{\, A'' \,\}}{M \vdash \{\, x : C \wedge A \,\}\ s\ \{\, x : C \wedge A' \,\}\ \parallel\ \{\, A'' \,\}}$$

Fig. 12. Types

[COMBINE]

$$\frac{\begin{array}{c} M \vdash \{\, A_1 \,\}\ s\ \{\, A_2 \,\}\ \parallel\ \{\, A \,\} \\ M \vdash \{\, A_3 \,\}\ s\ \{\, A_4 \,\}\ \parallel\ \{\, A \,\} \end{array}}{M \vdash \{\, A_1 \wedge A_3 \,\}\ s\ \{\, A_2 \wedge A_4 \,\}\ \parallel\ \{\, A \,\}}$$

[SEQU]

$$\frac{\begin{array}{c} M \vdash \{\, A_1 \,\}\ s_1\ \{\, A_2 \,\}\ \parallel\ \{\, A \,\} \\ M \vdash \{\, A_2 \,\}\ s_2\ \{\, A_3 \,\}\ \parallel\ \{\, A \,\} \end{array}}{M \vdash \{\, A_1 \,\}\ s_1;\ s_2\ \{\, A_3 \,\}\ \parallel\ \{\, A \,\}}$$

[CONSEQU]

$$\frac{M \vdash \{\, A_4 \,\}\ s\ \{\, A_5 \,\}\ \parallel\ \{\, A_6 \,\} \qquad M \vdash A_1 \longrightarrow A_4 \qquad M \vdash A_5 \longrightarrow A_2 \qquad M \vdash A_6 \longrightarrow A_3}{M \vdash \{\, A_1 \,\}\ s\ \{\, A_2 \,\}\ \parallel\ \{\, A_3 \,\}}$$

[IF_RULE]

$$\frac{\begin{array}{c} M \vdash \{\ A \wedge Cond\ \}\ stmt_1\ \{\ A'\ \}\ \parallel\ \{\ A''\ \} \\ M \vdash \{\ A \wedge \neg Cond\ \}\ stmt_2\ \{\ A'\ \}\ \parallel\ \{\ A''\ \} \end{array}}{M \vdash \{\ A\ \}\ \texttt{if}\ Cond\ \texttt{then}\ stmt_1\ \texttt{else}\ stmt_2\ \{\ A'\ \}\ \parallel\ \{\ A''\ \}}$$

[ABSURD]

$$\frac{}{M \vdash \{\ false\ \}\ stmt\ \{\ A'\ \}\ \parallel\ \{\ A''\ \}}$$

[CASES]

$$\frac{\begin{array}{c} M \vdash \{\ A \wedge A_1\ \}\ stmt\ \{\ A'\ \}\ \parallel\ \{\ A''\ \} \\ M \vdash \{\ A \wedge A_2\ \}\ stmt\ \{\ A'\ \}\ \parallel\ \{\ A''\ \} \end{array}}{M \vdash \{\ A \wedge (A_1 \vee A_2)\ \}\ stmt\ \{\ A'\ \}\ \parallel\ \{\ A''\ \}}$$

Fig. 13. Hoare Quadruples - substructural rules, and conditionals

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

E.3 Second Phase - more

in Fig. 13, we extend the Hoare Quadruples Logic with substructural rules, rules for conditionals, case analysis, and a contradiction rule. For the conditionals we assume the obvious operational. semantics, but do not define it in this paper

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

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

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

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

$$\frac{}{M \vdash \{\ false\ \}\ stmt\ \{\ A'\ \}\ \parallel\ \{\ A''\ \}}\ [\textsc{Absurd}]$$

$$\frac{M \vdash \{\ A \wedge A_1\ \}\ stmt\ \{\ A'\ \}\ \parallel\ \{\ A''\ \} \qquad M \vdash \{\ A \wedge A_2\ \}\ stmt\ \{\ A'\ \}\ \parallel\ \{\ A''\ \}}{M \vdash \{\ A \wedge (A_1 \vee A_2)\ \}\ stmt\ \{\ A'\ \}\ \parallel\ \{\ A''\ \}}\ [\textsc{Cases}]$$

Fig. 13. Hoare Quadruples - substructural rules, and conditionals

Finally, we discuss the proof

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

End Proof

F.4 Adaptation

We now discuss the proof of Lemma 8.3.

Proof of lemma 8.3, part 1

To Show: $Stbl(\ A \neg\nabla(y_0, \overline{y})\ )$

By structural induction on A.

End Proof

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

Auxiliary Lemma F.4. For all α, $\overline{\phi_1}$, $\overline{\phi_2}$, $\overline{\phi_2}$, ϕ and χ

$(L1)$ $M, (\overline{\phi_1}, \chi) \models \langle\!\langle \alpha \rangle\!\rangle \leftrightarrow\!\!\!\!\!\times\, Rng(\phi) \implies M, (\overline{\phi_2} \cdot \phi, \chi) \models \langle\!\langle \alpha \rangle\!\rangle$

$(L2)$ $M, (\overline{\phi_1} \cdot \phi, \chi) \models \langle\!\langle \alpha \rangle\!\rangle \wedge \texttt{extl} \implies M, (\overline{\phi_2}, \chi) \models \langle\!\langle \alpha \rangle\!\rangle \leftrightarrow\!\!\!\!\!\times\, Rng(\phi)$

$(L3)$ $M, (\overline{\phi_1} \cdot \phi_1, \chi) \models \langle\!\langle \alpha \rangle\!\rangle \wedge \texttt{extl} \wedge Rng(\phi) \subseteq Rng(\phi_1) \implies M, (\overline{\phi_2}, \chi) \models \langle\!\langle \alpha \rangle\!\rangle \leftrightarrow\!\!\!\!\!\times\, Rng(\phi)$

Proof.

We first prove (L1):

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

The above definitions imply that:

(1) $\forall \alpha', \forall \overline{f}.[\ \lfloor \alpha'.\overline{f} \rfloor_{\sigma_1} = \lfloor \alpha'.\overline{f} \rfloor_{\sigma_2}\ ]$

(2) $\forall \alpha'.[\ Rchbl(\alpha', \sigma_1) = Rchbl(\alpha', \sigma_2)\ ]$

(3) $LocRchbl(\sigma_2) = \bigcup_{\alpha' \in Rng(\phi)} Rchbl(\alpha', \sigma_2)$.

We now assume that

(4) $M, \sigma_1 \models \langle\!\langle \alpha \rangle\!\rangle \leftrightarrow\!\!\!\!\!\times\, Rng(\phi)$.

and want to show that

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

From (4) and by definitions, we obtain that

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

(6) $\alpha \notin Rng(\phi)$

From (5) and (3) we obtain:

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

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

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

From (8), by definitions, we obtain

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

This completes the proof of (L1).

We now prove (L2):

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

Finally, we discuss the proof

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

End Proof

E.4 Adaptation

We now discuss the proof of Lemma 8.3.

Proof of lemma 8.3, part 1

To Show: $\quad Stbl(\, A \mathrel{-\!\!\triangledown} (y_0, \overline{y})\,)$

By structural induction on A.

End Proof

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

Auxiliary Lemma E.4. For all α, $\overline{\phi_1}$, $\overline{\phi_2}$, $\overline{\phi_2}$, ϕ and χ

$(L1)\quad M, (\overline{\phi_1}, \chi) \models \langle\alpha\rangle \not\leftrightarrow Rng(\phi) \implies M, (\overline{\phi_2} \cdot \phi, \chi) \models \langle\alpha\rangle$

$(L2)\quad M, (\overline{\phi_1} \cdot \phi, \chi) \models \langle\alpha\rangle \wedge \texttt{extl} \implies M, (\overline{\phi_2}, \chi) \models \langle\alpha\rangle \not\leftrightarrow Rng(\phi)$

$(L3)\quad M, (\overline{\phi_1} \cdot \phi_1, \chi) \models \langle\alpha\rangle \wedge \texttt{extl} \;\wedge Rng(\phi) \subseteq Rng(\phi_1) \implies M, (\overline{\phi_2}, \chi) \models \langle\alpha\rangle \not\leftrightarrow Rng(\phi)$

Proof.

We first prove (L1):

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

The above definitions imply that:

(1) $\forall \alpha', \forall \overline{f}.[\; \lfloor \alpha'.\overline{f} \rfloor_{\sigma_1} = \lfloor \alpha'.\overline{f} \rfloor_{\sigma_2} \;]$

(2) $\forall \alpha'.[\; Rchbl(\alpha', \sigma_1) = Rchbl(\alpha', \sigma_2) \;]$

(3) $LocRchbl(\sigma_2) = \bigcup_{\alpha' \in Rng(\phi)} Rchbl(\alpha', \sigma_2)$.

We now assume that

(4) $M, \sigma_1 \models \langle\alpha\rangle \not\leftrightarrow Rng(\phi)$.

and want to show that

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

From (4) and by definitions, we obtain that

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

(6) $\alpha \notin Rng(\phi)$

From (5) and (3) we obtain:

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

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

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

From (8), by definitions, we obtain

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

This completes the proof of (L1).

We now prove (L2):

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

The above definitions imply that:
(1) $\forall \alpha', \forall \overline{f}.[\ \lfloor \alpha'.\overline{f} \rfloor_{\sigma_1} = \lfloor \alpha'.\overline{f} \rfloor_{\sigma_2}\ ]$
(2) $\forall \alpha'.[\ Rchbl(\alpha', \sigma_1) = Rchbl(\alpha', \sigma_2)\ ]$
(3) $LocRchbl(\sigma_1) = \bigcup_{\alpha' \in Rng(\phi)} Rchbl(\alpha', \sigma_1)$.
We assume that
(4) $M, \sigma_1 \models \langle\!\langle \alpha \rangle\!\rangle \wedge \texttt{extl}$.
and want to show that
(A?) $M, \sigma_2 \models A \mathrel{-\!\nabla} Rng(\phi)$.
From (4), and unfolding the definitions, we obtain:
(5) $\forall \alpha' \in LocRchbl(\sigma_1).\forall f.[\ M, \sigma_1 \models \alpha' : \texttt{extl} \ \rightarrow \alpha'.f \neq \alpha\ ]$, and
(6) $\forall \alpha' \in Rng(\phi).[\ \alpha' \neq \alpha\ ]$.
From(5), and using (3) and (2) we obtain:
(7) $\forall \alpha' \in Rng(\phi).\forall \alpha'' \in Rchbl(\alpha', \sigma_2).\forall f.[\ M, \sigma_2 \models \alpha'' : \texttt{extl} \ \rightarrow \alpha''.f \neq \alpha\ ]$
From (5) and (7) and by definitions, we obtain
(8) $\forall \alpha' \in Rng(\phi).[\ \models \alpha \langle\!\langle \alpha \rangle\!\rangle \not\leftrightarrow \alpha'\ ]$.
From (8) and definitions we obtain (A?).
This completes the proof of (L2).

We now prove (L3):

We define $\sigma_1 \triangleq (\overline{\phi_1} \cdot \phi_1, \chi)$, and $\sigma_2 \triangleq (\overline{\phi_2}, \chi)$.
The above definitions imply that:
(1) $\forall \alpha', \forall \overline{f}.[\ \lfloor \alpha'.\overline{f} \rfloor_{\sigma_1} = \lfloor \alpha'.\overline{f} \rfloor_{\sigma_2}\ ]$
(2) $\forall \alpha'.[\ Rchbl(\alpha', \sigma_1) = Rchbl(\alpha', \sigma_2)\ ]$
(3) $LocRchbl(\sigma_1) = \bigcup_{\alpha' \in Rng(\phi_1)} Rchbl(\alpha', \sigma_1)$.
We assume that
(4a) $M, \sigma_1 \models \langle\!\langle \alpha \rangle\!\rangle \wedge \texttt{extl}$, and (4b) $Rng(\phi) \subseteq Rng(\phi_1)$
We want to show that
(A?) $M, \sigma_2 \models A \mathrel{-\!\nabla} Rng(\phi)$.
From (4a), and unfolding the definitions, we obtain:
(5) $\forall \alpha' \in LocRchbl(\sigma_1).\forall f.[\ M, \sigma_1 \models \alpha' : \texttt{extl} \ \rightarrow \alpha'.f \neq \alpha\ ]$, and
(6) $\forall \alpha' \in Rng(\phi_1).[\ \alpha' \neq \alpha\ ]$.
From(5), and (3) and (2) and (4b) we obtain:
(7) $\forall \alpha' \in Rng(\phi).\forall \alpha'' \in Rchbl(\alpha', \sigma_2).\forall f.[\ M, \sigma_2 \models \alpha'' : \texttt{extl} \ \rightarrow \alpha''.f \neq \alpha\ ]$
From(6), and (4b) we obtain:
(8) $\forall \alpha' \in Rng(\phi_1).[\ \alpha' \neq \alpha\ ]$.
From (8) and definitions we obtain (A?).
This completes the proof of (L3).

□

Proof of lemma 8.3, part 2
To Show: (∗) $M, \sigma \models A \mathrel{-\!\nabla} Rng(\phi) \quad \Longrightarrow \quad M, \sigma \triangledown \phi \models A$

By induction on the structure of A. For the case where A has the form $\langle\!\langle \alpha.\overline{f} \rangle\!\rangle$, we use lemma F.4,(L1), taking $\overline{\phi_1} = \overline{\phi_2}$, and $\sigma \triangleq (\overline{\phi_1}, \chi)$.
End Proof

The above definitions imply that:
(1) $\forall \alpha', \forall \overline{f}.[\ \lfloor \alpha'.\overline{f} \rfloor_{\sigma_1} = \lfloor \alpha'.\overline{f} \rfloor_{\sigma_2}\ ]$
(2) $\forall \alpha'.[\ Rchbl(\alpha', \sigma_1) = Rchbl(\alpha', \sigma_2)\ ]$
(3) $LocRchbl(\sigma_1) = \bigcup_{\alpha' \in Rng(\phi)} Rchbl(\alpha', \sigma_1)$.
We assume that
(4) $M, \sigma_1 \models \langle\!\langle \alpha \rangle\!\rangle \wedge \texttt{extl}$.
and want to show that
(A?) $M, \sigma_2 \models A \multimap\!\!\!\triangledown Rng(\phi)$.
From (4), and unfolding the definitions, we obtain:
(5) $\forall \alpha' \in LocRchbl(\sigma_1).\forall f.[\ M, \sigma_1 \models \alpha' : \texttt{extl} \ \rightarrow \alpha'.f \neq \alpha\ ]$, and
(6) $\forall \alpha' \in Rng(\phi).[\ \alpha' \neq \alpha\ ]$.
From(5), and using (3) and (2) we obtain:
(7) $\forall \alpha' \in Rng(\phi).\forall \alpha'' \in Rchbl(\alpha', \sigma_2).\forall f.[\ M, \sigma_2 \models \alpha'' : \texttt{extl} \ \rightarrow \alpha''.f \neq \alpha\ ]$
From (5) and (7) and by definitions, we obtain
(8) $\forall \alpha' \in Rng(\phi).[\ \models \alpha \langle\!\langle \alpha \rangle\!\rangle \not\leftrightarrow \alpha'\ ]$.
From (8) and definitions we obtain (A?).
This completes the proof of (L2).

We now prove (L3):

We define $\sigma_1 \triangleq (\overline{\phi_1} \cdot \phi_1, \chi)$, and $\sigma_2 \triangleq (\overline{\phi_2}, \chi)$.
The above definitions imply that:
(1) $\forall \alpha', \forall \overline{f}.[\ \lfloor \alpha'.\overline{f} \rfloor_{\sigma_1} = \lfloor \alpha'.\overline{f} \rfloor_{\sigma_2}\ ]$
(2) $\forall \alpha'.[\ Rchbl(\alpha', \sigma_1) = Rchbl(\alpha', \sigma_2)\ ]$
(3) $LocRchbl(\sigma_1) = \bigcup_{\alpha' \in Rng(\phi_1)} Rchbl(\alpha', \sigma_1)$.
We assume that
(4a) $M, \sigma_1 \models \langle\!\langle \alpha \rangle\!\rangle \wedge \texttt{extl}$, and (4b) $Rng(\phi) \subseteq Rng(\phi_1)$
We want to show that
(A?) $M, \sigma_2 \models A \multimap\!\!\!\triangledown Rng(\phi)$.
From (4a), and unfolding the definitions, we obtain:
(5) $\forall \alpha' \in LocRchbl(\sigma_1).\forall f.[\ M, \sigma_1 \models \alpha' : \texttt{extl} \ \rightarrow \alpha'.f \neq \alpha\ ]$, and
(6) $\forall \alpha' \in Rng(\phi_1).[\ \alpha' \neq \alpha\ ]$.
From(5), and (3) and (2) and (4b) we obtain:
(7) $\forall \alpha' \in Rng(\phi).\forall \alpha'' \in Rchbl(\alpha', \sigma_2).\forall f.[\ M, \sigma_2 \models \alpha'' : \texttt{extl} \ \rightarrow \alpha''.f \neq \alpha\ ]$
From(6), and (4b) we obtain:
(8) $\forall \alpha' \in Rng(\phi_1).[\ \alpha' \neq \alpha\ ]$.
From (8) and definitions we obtain (A?).
This completes the proof of (L3).

$\square$

Proof of lemma 8.3, part 2
To Show: $(*)\quad M, \sigma \models A \multimap\!\!\!\triangledown Rng(\phi) \qquad \Longrightarrow \qquad M, \sigma \triangledown \phi \models A$

By induction on the structure of A. For the case where A has the form $\langle\!\langle \alpha.\overline{f} \rangle\!\rangle$, we use lemma E.4,(L1), taking $\overline{\phi_1} = \overline{\phi_2}$, and $\sigma \triangleq (\overline{\phi_1}, \chi)$.
End Proof

Proof of lemma 8.3, part 3
To Show $(*)$ $M, \sigma \triangledown \phi \models A \wedge \texttt{extl} \implies M, \sigma \models A \neg\triangledown Rng(\phi)$

We apply induction on the structure of A. For the case where A has the form $\langle\!\langle \alpha.\overline{f} \rangle\!\rangle$, we apply lemma F.4,(L2), using $\overline{\phi_1} = \overline{\phi_2}$, and $\sigma \triangleq (\overline{\phi_1}, \chi)$.
End Proof

Proof of lemma 8.3, part 4
To Show: (*) $M, \sigma \models A \wedge \texttt{extl} \ \wedge \ M \cdot \overline{M} \models \sigma \triangledown \phi \implies M, \sigma \triangledown \phi \models A \neg\triangledown Rng(\phi)$

By induction on the structure of A. For the case where A has the form $\langle\!\langle \alpha.\overline{f} \rangle\!\rangle$, we want to apply lemma F.4,(L3). We take σ to be $(\overline{\phi_1} \cdot \phi_1, \chi)$, and $\overline{\phi_2} = \overline{\phi_1} \cdot \phi_1 \cdot \phi$. Moreover, $M \cdot \overline{M} \models \sigma \triangledown \phi$ gives that $Rng(\phi) \subseteq LocRchbl(\sigma_2)$. Therefore, (*) follows by application of lemma F.4,(L3).
End Proof

Proof of lemma 8.3, part 3

To Show $(*)$ $M, \sigma \triangledown \phi \models A \wedge \texttt{extl} \quad \Longrightarrow \quad M, \sigma \models A \triangledown\!\!\!- Rng(\phi)$

We apply induction on the structure of A. For the case where A has the form $\langle\!\langle \alpha.\overline{f} \rangle\!\rangle$, we apply lemma E.4,(L2), using $\overline{\phi_1} = \overline{\phi_2}$, and $\sigma \triangleq (\overline{\phi_1}, \chi)$.

End Proof

Proof of lemma 8.3, part 4

To Show: (*) $M, \sigma \models A \wedge \texttt{extl} \ \wedge \ M \cdot \overline{M} \models \sigma \triangledown \phi \quad \Longrightarrow \quad M, \sigma \triangledown \phi \models A \triangledown\!\!\!- Rng(\phi)$

By induction on the structure of A. For the case where A has the form $\langle\!\langle \alpha.\overline{f} \rangle\!\rangle$, we want to apply lemma E.4,(L3). We take σ to be $(\overline{\phi_1} \cdot \phi_1, \chi)$, and $\overline{\phi_2} = \overline{\phi_1} \cdot \phi_1 \cdot \phi$. Moreover, $M \cdot \overline{M} \models \sigma \triangledown \phi$ gives that $Rng(\phi) \subseteq LocRchbl(\sigma_2)$. Therefore, (*) follows by application of lemma E.4,(L3).

End Proof

G APPENDIX TO SECTION 9 – SOUNDNESS OF THE HOARE LOGICS

G.1 Expectations

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

$$M \vdash A \quad \Longrightarrow \quad \forall \sigma.[\ M, \sigma \models A\ ].$$
$$M\ \vdash_{ul}\ \{\, A \,\}\, stmt \{\, A' \,\} \quad \Longrightarrow \quad M \models \{\, A \,\}\, stmt \{\, A' \,\}$$

G.2 Scoped satisfaction of assertions

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

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

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

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

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

Proof Sketch

(1) By unfolding and folding the definitions.
(2) By unfolding and folding the definitions.
(3) By induction on the definition of $Stbl(\_)$.
(4) By contradiction: Find a σ, a k and such that $\forall i \geq k.[M, \sigma[i] \models A[\overline{\lfloor z \rfloor_\sigma / z}]$, and $\exists j \geq k.[M, \sigma[j] \not\models A'[[\overline{\lfloor z \rfloor_\sigma / z}]$ such that $\overline{z} = Fv(A)$. Take $\sigma'' \triangleq \sigma[j]$, and then we have that $M, \sigma'' \models A[[\overline{\lfloor z \rfloor_\sigma / z}]$ and $M, \sigma'' \not\models A'[[\overline{\lfloor z \rfloor_\sigma / z}]$. This contradicts $M \models A \rightarrow A'$. Here we are also using the property that $M \models A$ and $u \notin Fv(A)$ implies $M \models A[u/z]$ – this is needed because we have free variables in A which are not free in $A[...]$

End Proof Sketch

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

Lemma G.4. For states σ, σ', frame ϕ such that $\sigma' = \sigma \triangledown \phi$, and for assertion A, such that $fv(A) = \emptyset$:

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

Proof. By structural induction on A, and unfolding/folding the definitions. □

G.3 Shallow and Scoped Semantics of Hoare tuples

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

Example G.5 (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 .., $\sigma_a \models \langle\!\langle o_4 \rangle\!\rangle$. 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 .., $\sigma_b \models \langle\!\langle o_4 \rangle\!\rangle$. 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

F Appendix to Section 9 – Soundness of the Hoare Logics

F.1 Expectations

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

$$M \vdash A \quad \Longrightarrow \quad \forall \sigma.[\ M, \sigma \models A\ ].$$
$$M\ \vdash_{ul}\ \{\, A\,\}\ stmt\{\, A'\,\} \quad \Longrightarrow \quad M \models \{\, A\,\}\ stmt\{\, A'\,\}$$

F.2 Scoped satisfaction of assertions

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

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

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

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

(1) $M, \sigma, |\sigma| \models A \quad \Longleftrightarrow \quad M, \sigma \models A$
(2) $M, \sigma, k \models A \ \land\ k \leq k' \quad \Longrightarrow \quad M, \sigma, k' \models A$
(3) $M, \sigma \models A \ \land\ Stbl(A) \quad \Longrightarrow \quad \forall k \leq |\sigma|.[\ M, \sigma, k \models A\ ]$
(4) $M \models A \to A' \quad \Longrightarrow \quad \forall \sigma. \forall k \leq |\sigma|.[\ M, \sigma, k \models A \ \Longrightarrow\ M, \sigma, k \models A'\ ]$

Proof Sketch

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

End Proof Sketch

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

Lemma F.4. For states σ, σ', frame ϕ such that $\sigma' = \sigma \triangledown \phi$, and for assertion A, such that $fv(A) = \emptyset$:

- $M, \sigma, k \models A \ \land\ M, \sigma' \models A \quad \Longleftrightarrow \quad M, \sigma', k \models A$

Proof. By structural induction on A, and unfolding/folding the definitions. □

F.3 Shallow and Scoped Semantics of Hoare tuples

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

Example F.5 (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 .., $\sigma_a \models \langle\!\langle o_4 \rangle\!\rangle$. 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 .., $\sigma_b \models \langle\!\langle o_4 \rangle\!\rangle$. 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 $.., \sigma_c \models \langle\!\langle o_4 \rangle\!\rangle$. However, after we pop the stack, we obtain σ_d, for which $.., \sigma_d \not\models \langle\!\langle o_4 \rangle\!\rangle$.

Definition G.6 (Scoped Satisfaction of Quadruples by States)**.** For modules $\overline{M}$, M, state σ, and assertions A, A' and A''

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

Lemma G.7. For all M, $\overline{M}$ A, A', A'' and σ:

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

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

Definition G.8 (Scoped Semantics of Hoare triples)**.** For modules M, and assertions A, A' we define:

- $M \models \{\, A \,\}\; stmt \;\{\, A' \,\} \quad \triangleq$
  $\forall \overline{M}. \forall \sigma.[\;\; \sigma.\texttt{cont} \stackrel{\text{txt}}{=} stmt \implies \overline{M}; M \models \{\;\, A \;\,\} \sigma \{\, A' \,\} \;\|\; \{\, true \,\} \;\;]$
- $M \models \{\, A \,\}\; stmt \{\, A' \,\} \;\;\|\;\; \{\, A'' \,\} \quad \triangleq$
  $\forall \overline{M}. \forall \sigma.[\;\; \sigma.\texttt{cont} \stackrel{\text{txt}}{=} stmt \implies \overline{M}; M \models \{\;\, A \;\,\} \sigma \{\, A' \,\} \;\|\; \{\, A'' \,\} \;\;]$
- $M \models_{\circ} \{\, A \,\}\; stmt \;\{\, A' \,\} \quad \triangleq$
  $\forall \overline{M}. \forall \sigma.[\;\; \sigma.\texttt{cont} \stackrel{\text{txt}}{=} stmt \implies \overline{M}; M \models_{\circ} \{\;\, A \;\,\} \sigma \{\, A' \,\} \;\|\; \{\, true \,\} \;\;]$
- $M \models_{\circ} \{\, A \,\}\; stmt \{\, A' \,\} \;\;\|\;\; \{\, A'' \,\} \quad \triangleq$
  $\forall \overline{M}. \forall \sigma.[\;\; \sigma.\texttt{cont} \stackrel{\text{txt}}{=} stmt \implies \overline{M}; M \models_{\circ} \{\;\, A \;\,\} \sigma \{\, A' \,\} \;\|\; \{\, A'' \,\} \;\;]$

Lemma G.9 (Scoped vs Shallow Semantics of Quadruples)**.** For all M, A, A', and *stmt*:

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

Proof. By unfolding and folding the definitions □

G.4 Scoped satisfaction of specifications

We now give a scoped meaning to specifications:

Definition G.10 (Scoped Semantics of Specifications)**.** We define $M \models_{\circ} S$ by cases:

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

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

(3) $M \models_{\circ} S \;\wedge\; S' \quad \triangleq \quad M \models_{\circ} S \;\wedge\; M \models_{\circ} S'$

Lemma G.11 (Scoped vs Shallow Semantics of Quadruples)**.** For all M, S:

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

the end of method execution, and before popping the stack, we have state σ_c, which also satisfies $.., \sigma_c \models \langle\!\langle o_4 \rangle\!\rangle$. However, after we pop the stack, we obtain σ_d, for which $.., \sigma_d \not\models \langle\!\langle o_4 \rangle\!\rangle$.

Definition F.6 (Scoped Satisfaction of Quadruples by States). For modules $\overline{M}$, M, state σ, and assertions A, A' and A''

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

Lemma F.7. For all M, $\overline{M}$ A, A', A'' and σ:

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

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

Definition F.8 (Scoped Semantics of Hoare triples). For modules M, and assertions A, A' we define:

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

Lemma F.9 (Scoped vs Shallow Semantics of Quadruples). For all M, A, A', and *stmt*:

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

Proof. By unfolding and folding the definitions □

F.4 Scoped satisfaction of specifications

We now give a scoped meaning to specifications:

Definition F.10 (Scoped Semantics of Specifications). We define $M \models_{\circ} S$ by cases:

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

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

(3) $M \models_{\circ} S \;\wedge\; S' \quad \triangleq \quad M \models_{\circ} S \;\wedge\; M \models_{\circ} S'$

Lemma F.11 (Scoped vs Shallow Semantics of Quadruples). For all M, S:

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

G.5 Soundness of the Hoare Triples Logic

Auxiliary Lemma G.12. For any module M, assertions A, A' and A'', such that $Stb^+(A)$, and $Stb^+(A')$, and a statement $stmt$ which does not contain any method calls:

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

Proof. □

G.5.1 Lemmas about protection.

Definition G.13. $LocRchbl(\sigma, k) \triangleq \{\alpha \mid \exists i.[\ k \le i \le |\sigma| \ \wedge\ \alpha \in LocRchbl(\sigma[i])\ ]$

Lemma G.14 guarantees that program execution reduces the locally reachable objects, unless it allocates new ones. That is, any objects locally reachable in the k-th frame of the new state (σ'), are either new, or were locally reachable in the k-th frame of the previous state (σ).

Lemma G.14. For all σ, σ', and α, where $\models \sigma$, and where $k \le |\sigma|$:

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

Proof.

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

□

Lemma G.15 guarantees that any change to the contents of an external object can only happen during execution of an external method.

Lemma G.15. For all σ, σ':

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

Proof. Through inspection of the operational semantics in Fig. 5, and in particular rule Write. □

Lemma G.16 guarantees that internal code which does not include method calls preserves absolute protection. It is used in the proof of soundness of the inference rule Prot-1.

Lemma G.16. For all σ, σ', and α:

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

Proof.

F.5 Soundness of the Hoare Triples Logic

Auxiliary Lemma F.12. For any module M, assertions A, A' and A'', such that $Stb^+(A)$, and $Stb^+(A')$, and a statement $stmt$ which does not contain any method calls:

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

Proof. □

F.5.1 Lemmas about protection.

Definition F.13. $LocRchbl(\sigma, k) \quad \triangleq \quad \{\alpha \mid \exists i.[\; k \leq i \leq |\sigma| \;\wedge\; \alpha \in LocRchbl(\sigma[i]) \;]$

Lemma F.14 guarantees that program execution reduces the locally reachable objects, unless it allocates new ones. That is, any objects locally reachable in the k-th frame of the new state (σ'), are either new, or were locally reachable in the k-th frame of the previous state (σ).

Lemma F.14. For all σ, σ', and α, where $\models \sigma$, and where $k \leq |\sigma|$:

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

Proof.

- If the step is a method call, then the assertion follows by construction. If the steps is a local execution in a method, we proceed by case analysis. If it is an assignment to a local variable, then $\forall k.[\; LocRchbl(\sigma', k) = LocRchbl(\sigma, k) \;]$. If the step is the creation of a new object, then the assertion holds by construction. If it it is a field assignment, say, $\sigma' = \sigma[\alpha_1, f \mapsto \alpha_2]$, then we have that α_1, α_2 $LocRchbl(\sigma, |\sigma|)$. And therefore, by Lemma B.3, we also have that α_1, α_2 $LocRchbl(\sigma, k)$ All locally reachable objects in σ' were either already reachable in σ or reachable through α_2, Therefore, we also have that $LocRchbl(\sigma', k) \subseteq LocRchbl(\sigma, k)$ And by definition of $\_;\, \_ \rightsquigarrow \_$, it is not a method return.
- By induction on the number of steps in $\overline{M} \cdot M;\; \sigma \rightsquigarrow^* \sigma'$. For the steps that correspond to method calls, the assertion follows by construction. For the steps that correspond to local execution in a method, the assertion follows from the bullet above. For the steps that correspond to method returns, the assertion follows by lemma B.3.

□

Lemma F.15 guarantees that any change to the contents of an external object can only happen during execution of an external method.

Lemma F.15. For all σ, σ':

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

Proof. Through inspection of the operational semantics in Fig. 5, and in particular rule Write. □

Lemma F.16 guarantees that internal code which does not include method calls preserves absolute protection. It is used in the proof of soundness of the inference rule Prot-1.

Lemma F.16. For all σ, σ', and α:

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

Proof.

- Because σ.cont contains no method calls, we also have that $|\sigma'| = |\sigma|$. Let us take $m = |\sigma|$. We continue by contradiction. Assume that $M, \sigma, k \models \langle\!\langle \alpha \rangle\!\rangle$ and $M, \sigma, k \not\models \langle\!\langle \alpha \rangle\!\rangle$
  Then:
  (*) $\forall f.\forall i \in [k..m].\forall \alpha_o \in LocRchbl(\sigma, i).[\ M, \sigma \models \alpha_o : \texttt{extl} \Rightarrow \lfloor \alpha_o.f \rfloor_\sigma \neq \alpha\ \wedge \alpha_o \neq \alpha\ ]$.
  (**) $\exists f.\exists j \in [k..m].\exists \alpha_o \in LocRchbl(\sigma', j).[\ M, \sigma' \models \alpha_o : \texttt{extl} \wedge \lfloor \alpha_o.f \rfloor_{\sigma'} = \alpha\ \vee \alpha_o = \alpha\ ]$
  We proceed by cases
  1st Case $\alpha_o \notin \sigma$, *i.e.* α_o is a new object. Then, by our operational semantics, it cannot have a field pointing to an already existing object (α), nor can it be equal with α. Contradiction.
  2nd Case $\alpha_o \in \sigma$. Then, by Lemma G.14, we obtain that $\alpha_o \in LocRchbl(\sigma, j)$. Therefore, using (*), we obtain that $\lfloor \alpha_o.f \rfloor_\sigma \neq \alpha$, and therefore $\lfloor \alpha_o.f \rfloor_\sigma \neq \lfloor \alpha_o.f \rfloor_{\sigma'}$. By lemma G.15, we obtain $M, \sigma \models \texttt{extl}$. Contradiction!
- By induction on the number of steps, and using the bullet above.

□

Lemma G.17. For all σ, σ', and α:

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

Proof. By unfolding and folding the definitions. □

Lemma G.18. For all σ, and α, α_o, α_1, α_2:

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

Definition G.19. • $M, \sigma \models e : \texttt{intl*} \triangleq \forall \overline{f}.[\ M, \sigma \models e.\overline{f} : \texttt{intl}\ ]$

Lemma G.20. For all σ, and α_o and α:

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

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

EXTEND By soundness of the underlying Hoare logic (axiom G.1), we obtain that $M \models \{\ A\ \}\ stmt\{\ A'\ \}$. By axiom F.3 we also obtain that $Stbl(A)$ and $Stbl(A')$. This, together with Lemma G.3, part 3, gives us that $M \models_{\circ} \{\ A\ \}\ stmt\{\ A'\ \}$. By the assumption of EXTEND, $stmt$ does not contain any method call. Rest follows by lemma G.12.

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

Prot-1 by Lemma G.16. The rule premise $M \vdash \{\ z = e\ \}\ stmt\{\ z = e\ \}$ allows us to consider addresses, α, rather than expressions, e.

Prot-2 by Lemma G.17. The rule premise $M \vdash \{\ z = e \wedge z = e'\ \}\ stmt\{\ z = e \wedge z = e'\ \}$ allows us to consider addresses α, α' rather than expressions e, e'.

Prot-3 also by Lemma G.17. Namely, the rule does not change, and $y.f$ in the old state has the same value as x in the new state.

Prot-4 by Lemma G.18.

TYPES-1 Follows from type system, the assumption of TYPES-1 and lemma G.12.

End Proof Sketch

G.6 Well-founded ordering

Definition G.21. For a module M, and modules $\overline{M}$, we define a measure, $[\![\ A, \sigma, A', A''\ ]\!]_{M,\overline{M}}$, and based on it, a well founded ordering $(A_1, \sigma_1, A_2, A_3) \ll_{M,\overline{M}} (A_4, \sigma_2, A_5, A_6)$ as follows:

- $[\![\ A, \sigma, A', A''\ ]\!]_{M,\overline{M}} \triangleq (m, n)$, where

- Because σ.cont contains no method calls, we also have that $|\sigma'| = |\sigma|$. Let us take $m = |\sigma|$. We continue by contradiction. Assume that $M, \sigma, k \models \langle\!\langle \alpha \rangle\!\rangle$ and $M, \sigma, k \not\models \langle\!\langle \alpha \rangle\!\rangle$
Then:
(*) $\forall f.\forall i \in [k..m].\forall \alpha_o \in LocRchbl(\sigma, i).[\ M, \sigma \models \alpha_o : \texttt{extl} \Rightarrow \lfloor \alpha_o.f \rfloor_\sigma \neq \alpha\ \wedge \alpha_o \neq \alpha\ ]$.
(**) $\exists f.\exists j \in [k..m].\exists \alpha_o \in LocRchbl(\sigma', j).[\ M, \sigma' \models \alpha_o : \texttt{extl} \wedge \lfloor \alpha_o.f \rfloor_{\sigma'} = \alpha\ \vee \alpha_o = \alpha\ ]$
We proceed by cases
1st Case $\alpha_o \notin \sigma$, *i.e.* α_o is a new object. Then, by our operational semantics, it cannot have a field pointing to an already existing object (α), nor can it be equal with α. Contradiction.
2nd Case $\alpha_o \in \sigma$. Then, by Lemma F.14, we obtain that $\alpha_o \in LocRchbl(\sigma, j)$. Therefore, using (*), we obtain that $\lfloor \alpha_o.f \rfloor_\sigma \neq \alpha$, and therefore $\lfloor \alpha_o.f \rfloor_\sigma \neq \lfloor \alpha_o.f \rfloor_{\sigma'}$. By lemma F.15, we obtain $M, \sigma \models \texttt{extl}$. Contradiction!
- By induction on the number of steps, and using the bullet above.

□

Lemma F.17. For all σ, σ', and α:

- $M, \sigma \models \langle\!\langle \alpha \rangle\!\rangle \not\leftrightarrow \alpha_o\ \wedge\ \sigma.\texttt{heap} = \sigma'.\texttt{heap} \implies M, \sigma' \models \langle\!\langle \alpha \rangle\!\rangle \not\leftrightarrow \alpha_o$

Proof. By unfolding and folding the definitions. □

Lemma F.18. For all σ, and α, α_o, α_1, α_2:

- $M, \sigma \models \langle\!\langle \alpha \rangle\!\rangle \not\leftrightarrow \alpha_o\ \wedge\ M, \sigma \models \langle\!\langle \alpha \rangle\!\rangle \not\leftrightarrow \alpha_1 \implies M, \sigma[\alpha_2, f \mapsto \alpha_1] \models \langle\!\langle \alpha \rangle\!\rangle \not\leftrightarrow \alpha_o$

Definition F.19.
- $M, \sigma \models e : \texttt{intl*} \triangleq \forall \overline{f}.[\ M, \sigma \models e.\overline{f} : \texttt{intl}\ ]$

Lemma F.20. For all σ, and α_o and α:

- $M, \sigma \models \alpha_o : \texttt{intl*} \implies M, \sigma \models \langle\!\langle \alpha \rangle\!\rangle \not\leftrightarrow \alpha_o$

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

EXTEND By soundness of the underlying Hoare logic (axiom F.1), we obtain that $M \models \{\ A\ \}\ stmt\{\ A'\ \}$. By axiom E.3 we also obtain that $Stbl(A)$ and $Stbl(A')$. This, together with Lemma F.3, part 3, gives us that $M \models_{\circ} \{\ A\ \}\ stmt\{\ A'\ \}$. By the assumption of EXTEND, *stmt* does not contain any method call. Rest follows by lemma F.12.

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

Prot-1 by Lemma F.16. The rule premise $M \vdash \{\ z = e\ \}\ stmt\{\ z = e\ \}$ allows us to consider addresses, α, rather than expressions, e.

Prot-2 by Lemma F.17. The rule premise $M \vdash \{\ z = e \wedge z = e'\ \}\ stmt\{\ z = e \wedge z = e'\ \}$ allows us to consider addresses α, α' rather than expressions e, e'.

Prot-3 also by Lemma F.17. Namely, the rule does not change, and $y.f$ in the old state has the same value as x in the new state.

Prot-4 by Lemma F.18.

TYPES-1 Follows from type system, the assumption of TYPES-1 and lemma F.12.

End Proof Sketch

F.6 Well-founded ordering

Definition F.21. For a module M, and modules $\overline{M}$, we define a measure, $[\![\ A, \sigma, A', A''\ ]\!]_{M,\overline{M}}$, and based on it, a well founded ordering $(A_1, \sigma_1, A_2, A_3) \ll_{M,\overline{M}} (A_4, \sigma_2, A_5, A_6)$ as follows:

- $[\![\ A, \sigma, A', A''\ ]\!]_{M,\overline{M}} \triangleq (m, n)$, where

- m is the minimal number of execution steps so that $M \cdot \overline{M};\ \sigma \rightsquigarrow^*_{fin} \sigma'$ for some σ', and ∞ otherwise.
- n is minimal depth of all proofs of $M \vdash \{\, A \,\}\ \sigma.\texttt{cont}\{\, A' \,\}\ \ \|\ \ \{\, A'' \,\}$.

- $(m, n) \ll (m', n') \quad \triangleq \quad m < m' \vee (m = m' \wedge n < n')$.
- $(A_1, \sigma_1, A_2, A_3) \ll_{M,\overline{M}} (A_4, \sigma_2, A_5, A_6) \quad \triangleq \quad [\![\, A_1, \sigma_1, A_2, A_3 \,]\!]_{M,\overline{M}} \ll [\![\, A_4, \sigma_2, A_5 . A_6 \,]\!]_{M,\overline{M}}$

Lemma G.22. For any modules M and $\overline{M}$, the relation $\_ \ll_{M,\overline{M}} \_$ is well-founded.

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

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

Definition G.23. We use the form $M, \sigma \models \texttt{pub}$ to express that the currently executing method is public.[14] Note that `pub` is not part of the assertion language.

Auxiliary Lemma G.24 (Enclosed Terminating Executions)**.** For modules $\overline{M}$, states σ, σ', σ_1:

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

Auxiliary Lemma G.25 (Executing sequences)**.** For modules $\overline{M}$, statements s_1, s_2, states σ, σ', σ''':

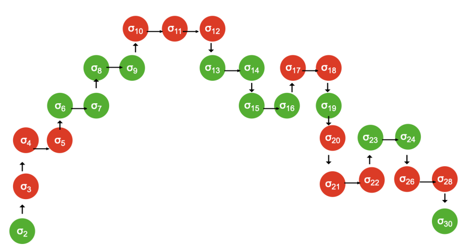
- $\sigma.\texttt{cont} = s_1; s_2 \ \wedge\ \overline{M};\ \sigma \rightsquigarrow^*_{fin} \sigma' \ \wedge\ \overline{M};\ \sigma \rightsquigarrow^* \sigma''$
  $\Longrightarrow$
  $\exists \sigma''.[\quad \overline{M};\ \sigma[\texttt{cont} \mapsto s_1] \rightsquigarrow^*_{fin} \sigma'' \ \wedge\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

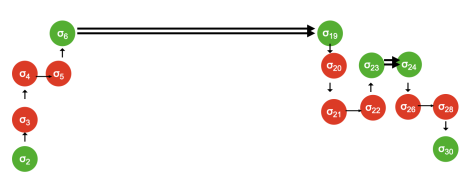
G.8 Summarised Executions

We repeat the two diagrams given in §9.

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

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





In order to express such summaries, Def. G.26 introduces the following concepts:

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

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

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

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

Lemma F.22. For any modules M and $\overline{M}$, the relation $\_ \ll_{M,\overline{M}} \_$ is well-founded.

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

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

Definition F.23. We use the form $M, \sigma \models \texttt{pub}$ to express that the currently executing method is public.[14] Note that pub is not part of the assertion language.

Auxiliary Lemma F.24 (Enclosed Terminating Executions)**.** For modules $\overline{M}$, states σ, σ', σ_1:

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

Auxiliary Lemma F.25 (Executing sequences)**.** For modules $\overline{M}$, statements s_1, s_2, states σ, σ', σ''':

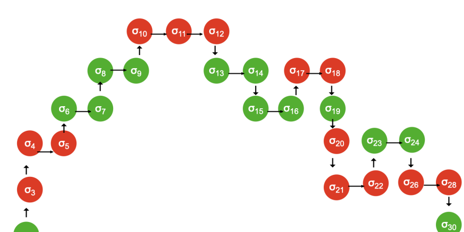
- $\sigma.\texttt{cont} = s_1; s_2 \ \ \land\ \ \overline{M};\ \sigma \rightsquigarrow^*_{fin} \sigma' \ \ \land\ \ \overline{M};\ \sigma \rightsquigarrow^* \sigma''$
  $\Longrightarrow$
  $\exists \sigma''.[\quad \overline{M};\ \sigma[\texttt{cont} \mapsto s_1] \rightsquigarrow^*_{fin} \sigma'' \ \land\ \overline{M};\ \sigma''[\texttt{cont} \mapsto s_2] \rightsquigarrow^*_{fin} \sigma' \ \land$
  $[\ \ \overline{M};\ \sigma[\texttt{cont} \mapsto s_1] \rightsquigarrow^* \sigma'' \ \lor\ \overline{M};\ \sigma''[\texttt{cont} \mapsto s_2] \rightsquigarrow^*_{fin} \sigma'''\ ] \qquad]$

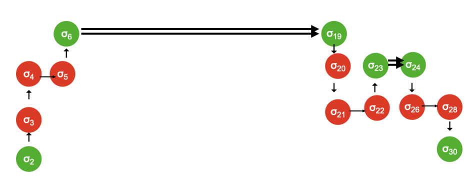
F.8 Summarised Executions

We repeat the two diagrams given in §9.

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

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





In order to express such summaries, Def. F.26 introduces the following concepts:

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

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

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

(1) $(\overline{M}\cdot M,\ \sigma_3); \sigma_3 \rightsquigarrow^*_e \sigma_5$ Purely external execution from σ_3 to σ_5, scoped by σ_3.
(2) $(\overline{M}\cdot M); \sigma_5 \rightsquigarrow^*_p \sigma_{20}\,\mathbf{pb}\,\sigma_6$. Public method call from external state σ_5 into nternal state σ_6 returning to σ_{20}. Note that this summarises two internal method executions ($\sigma_6 - \sigma_{19}$, and $\sigma_8 - \sigma_{14}$), and two external method executions ($\sigma_6 - \sigma_{19}$, and $\sigma_8 - \sigma_{14}$).
(3) $(\overline{M}\cdot M,\ \sigma_3); \sigma_{20} \rightsquigarrow^*_e \sigma_{21}$.
(4) $(\overline{M}\cdot M); \sigma_{21} \rightsquigarrow^*_p \sigma_{25}\,\mathbf{pb}\,\sigma_{23}$. Public method call from external state σ_{21} into internal state σ_{23}, and returning to external state σ_{25}.
(5) $(\overline{M}\cdot M,\ \sigma_3); \sigma_{25} \rightsquigarrow^*_e \sigma_{28}$. Purely external execution from σ_{25} to σ_{28}, scoped by σ_3.

Definition G.26. For any module M where M is the internal module, external modules $\overline{M}$, and states σ_{sc}, σ, σ_1, ... σ_n, and σ', we define:

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

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

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

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

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

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

Note that $(\overline{M}\cdot M,\ \sigma_{sc}); \sigma \rightsquigarrow^*_e \sigma'$ implies that σ is external, but does not imply that σ' is external. $(\overline{M}\cdot M,\ \sigma); \sigma \rightsquigarrow^*_e \sigma'$. On the other hand, $(\overline{M}\cdot M,\ \sigma_{sc}); \sigma \rightsquigarrow^*_{e,p} \sigma'\ \mathbf{pb}\,\sigma_1 ... \sigma_n$ implies that σ and σ' are external, and σ_1, ... σ_1 are internal and public. Finally, note that in part (6) above it is possible that $n = 0$, and so $\overline{M}\cdot M; \sigma \rightsquigarrow^*_{e,p} \sigma'$ also holds when Finally, note that the decomposition used in (5) is not unique, but since we only care for the public states this is of no importance.

Lemma G.27 says that

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

Auxiliary Lemma G.27. [Summarised Executions] For module M, modules $\overline{M}$, and states σ, σ':

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

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

(1) $(\overline{M}\cdot M,\ \sigma_3); \sigma_3 \rightsquigarrow^*_e \sigma_5$ Purely external execution from σ_3 to σ_5, scoped by σ_3.
(2) $(\overline{M}\cdot M); \sigma_5 \rightsquigarrow^*_p \sigma_{20}\,\mathbf{pb}\,\sigma_6$. Public method call from external state σ_5 into nternal state σ_6 returning to σ_{20}. Note that this summarises two internal method executions ($\sigma_6 - \sigma_{19}$, and $\sigma_8 - \sigma_{14}$), and two external method executions ($\sigma_6 - \sigma_{19}$, and $\sigma_8 - \sigma_{14}$).
(3) $(\overline{M}\cdot M,\ \sigma_3); \sigma_{20} \rightsquigarrow^*_e \sigma_{21}$.
(4) $(\overline{M}\cdot M); \sigma_{21} \rightsquigarrow^*_p \sigma_{25}\,\mathbf{pb}\,\sigma_{23}$. Public method call from external state σ_{21} into internal state σ_{23}, and returning to external state σ_{25}.
(5) $(\overline{M}\cdot M,\ \sigma_3); \sigma_{25} \rightsquigarrow^*_e \sigma_{28}$. Purely external execution from σ_{25} to σ_{28}, scoped by σ_3.

Definition F.26. For any module M where M is the internal module, external modules $\overline{M}$, and states σ_{sc}, σ, σ_1, ... σ_n, and σ', we define:

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

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

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

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

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

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

Note that $(\overline{M}\cdot M,\ \sigma_{sc}); \sigma \rightsquigarrow^*_e \sigma'$ implies that σ is external, but does not imply that σ' is external. $(\overline{M}\cdot M,\ \sigma); \sigma \rightsquigarrow^*_e \sigma'$. On the other hand, $(\overline{M}\cdot M,\ \sigma_{sc}); \sigma \rightsquigarrow^*_{e,p} \sigma'\ \mathbf{pb}\ \sigma_1 ... \sigma_n$ implies that σ and σ' are external, and σ_1, ... σ_1 are internal and public. Finally, note that in part (6) above it is possible that $n = 0$, and so $\overline{M}\cdot M; \sigma \rightsquigarrow^*_{e,p} \sigma'$ also holds when Finally, note that the decomposition used in (5) is not unique, but since we only care for the public states this is of no importance.

Lemma F.27 says that

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

Auxiliary Lemma F.27. [Summarised Executions] For module M, modules $\overline{M}$, and states σ, σ':

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

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

Auxiliary Lemma G.28. [Preservation of Encapsulated Assertions] For any module M, modules $\overline{M}$, assertion A, and states σ_{sc}, σ, σ_1 ... σ_n, σ_a, σ_b and σ':
If

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

Then

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

Proof Sketch

(1) is proven by Def. of $Enc(\_)$ and the fact $|\sigma'| \geq |\sigma_{sc}|$ and therefore $k \leq |\sigma'|$. In particular, the step $(\overline{M} \cdot M, \sigma_{sc});\ \sigma \rightsquigarrow \sigma'$ may push or pop a frame onto σ. If it pops a frame, then $M, \sigma', k \models A$ holds by definition. If is pushes a frame, then $M, \sigma' \models A$, by lemma 6.5; this gives that $M, \sigma', k \models A$.
(2) by induction on the number of steps in $(\overline{M} \cdot M,\ \sigma_{sc}); \sigma \rightsquigarrow_e^* \sigma'$, and using (1).
(3) by induction on the number of states appearing in $\sigma_1 ... \sigma_n$, and using (2).

End Proof Sketch

G.9 Sequences, Sets, Substitutions and Free Variables

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

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

Definition G.29 (Sequences, Disjointness, and Disjoint Concatenation)**.** For any variables v, w, and sequences of variables $\overline{v}$, $\overline{w}$ we define:

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

Lemma G.30 (Substitutions and Free Variables)**.** For any sequences of variables $\overline{x}$, $\overline{y}$, $\overline{z}$, $\overline{v}$, $\overline{w}$, a variable w, any assertion A, we have

(1) $\overline{x}[\overline{y/x}] = \overline{y}$
(2) $\overline{x} \# \overline{y} \quad \Rightarrow \quad \overline{y}[\overline{z/x}] = \overline{y}$
(3) $\overline{z} \subseteq \overline{y} \quad \Rightarrow \quad \overline{y}[\overline{z/x}] \subseteq \overline{y}$

Auxiliary Lemma F.28. [Preservation of Encapsulated Assertions] For any module M, modules $\overline{M}$, assertion A, and states σ_{sc}, σ, σ_1 ... σ_n, σ_a, σ_b and σ':
If

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

Then

(1) $M, \sigma \models \texttt{extl} \ \wedge \ (\overline{M} \cdot M, \sigma_{sc});\ \sigma \rightsquigarrow \sigma' \quad \Longrightarrow \quad M, \sigma', k \models A$
(2) $(\overline{M} \cdot M,\ \sigma_{sc}); \sigma \rightsquigarrow^*_e \sigma' \quad \Longrightarrow \quad M, \sigma', k \models A$
(3) $(\overline{M} \cdot M,\ \sigma_{sc}); \sigma \rightsquigarrow^*_{e,p} \sigma' \ \mathbf{pb}\ \sigma_1 ... \sigma_n \ \wedge$

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

$\Longrightarrow$

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

Proof Sketch

(1) is proven by Def. of $Enc(\_)$ and the fact $|\sigma'| \geq |\sigma_{sc}|$ and therefore $k \leq |\sigma'|$. In particular, the step $(\overline{M} \cdot M, \sigma_{sc});\ \sigma \rightsquigarrow \sigma'$ may push or pop a frame onto σ. If it pops a frame, then $M, \sigma', k \models A$ holds by definition. If is pushes a frame, then $M, \sigma' \models A$, by lemma 6.5; this gives that $M, \sigma', k \models A$.
(2) by induction on the number of steps in $(\overline{M} \cdot M,\ \sigma_{sc}); \sigma \rightsquigarrow^*_e \sigma'$, and using (1).
(3) by induction on the number of states appearing in $\sigma_1 ... \sigma_n$, and using (2).

End Proof Sketch

F.9 Sequences, Sets, Substitutions and Free Variables

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

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

Definition F.29 (Sequences, Disjointness, and Disjoint Concatenation)**.** For any variables v, w, and sequences of variables $\overline{v}$, $\overline{w}$ we define:

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

Lemma F.30 (Substitutions and Free Variables)**.** For any sequences of variables $\overline{x}$, $\overline{y}$, $\overline{z}$, $\overline{v}$, $\overline{w}$, a variable w, any assertion A, we have

(1) $\overline{x}[\overline{y/x}] = \overline{y}$
(2) $\overline{x} \# \overline{y} \ \Rightarrow \ \overline{y}[\overline{z/x}] = \overline{y}$
(3) $\overline{z} \subseteq \overline{y} \ \Rightarrow \ \overline{y}[\overline{z/x}] \subseteq \overline{y}$

(4) $\overline{y} \subseteq \overline{z} \;\;\Rightarrow\;\; \overline{y}[\overline{z/x}] \subseteq \overline{z}$

(5) $\overline{x}\#\overline{y} \;\;\Rightarrow\;\; \overline{z}[\overline{y/x}]\#\overline{x}$

(6) $Fv(A[\overline{y/x}]) \;=\; Fv(A)[\overline{y/x}]$

(7) $Fv(A) \;=\; \overline{x};\overline{v},\quad Fv(A[\overline{y/x}]) \;=\; \overline{y};\overline{w} \;\;\Longrightarrow\;\; \overline{v} \;=\; (\overline{y} \cap \overline{v});\overline{w}$

(8) $\overline{v}\#\overline{x}\#\overline{y}\#\overline{u} \;\;\Longrightarrow\;\; w[\overline{u/x}][\overline{v/y}] \stackrel{\text{txt}}{=} w[\overline{v/y}][\overline{u/x}]$

(9) $\overline{v}\#\overline{x}\#\overline{y}\#\overline{u} \;\;\Longrightarrow\;\; A[\overline{u/x}][\overline{v/y}] \stackrel{\text{txt}}{=} A[\overline{v/y}][\overline{u/x}]$

(10) $(fv(A[\overline{y/x}]) \setminus \overline{y})\,\#\,\overline{x}$

(11)

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

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

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

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

(5) by induction on the structure of A

(6) by induction on the structure of A

(7) Assume that

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

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

We define:

$(a)\quad \overline{y_0} \triangleq \overline{v} \cap \overline{y},\quad \overline{v_2} \triangleq \overline{v} \setminus \overline{y},\quad \overline{y_1} = \overline{y_0}[\overline{x/y}]$

This gives:

$(b)\quad \overline{y_0}\#\overline{v_2}$

$(c)\quad \overline{v} = \overline{y_0};\overline{v_2}$

$(d)\quad \overline{y_1} \subseteq \overline{y}$

$(e)\quad \overline{v_2}[\overline{y/x}] = \overline{v_2},$ from assumption and (a) we have $\overline{x}\#\overline{v_2}$ and by Lemma G.30) part (2)

We now calculate

$Fv(A[\overline{y/x}])$	$=$	$(\overline{x};\overline{v})[\overline{y/x}]$	by (ass1), and Lemma G.30 part (5).
	$=$	$(\overline{x};\overline{y_0};\overline{v_2})[\overline{y/x}]$	by (c) above
	$=$	$\overline{x}[\overline{y/x}],\ \overline{y_0}[\overline{y/x}], \overline{v_2}[\overline{y/x}]$	by distributivity of [../..]
	$=$	$\overline{y},\ \overline{y_1}, \overline{v_2}$	by Lemma G.30 part (1), (a), and (e).
	$=$	$\overline{y};\overline{v_2}$	because (d), and $\overline{y}\#\overline{v_2}$
$Fv(A[\overline{y/x}])$	$=$	$\overline{y};\overline{w}$	by (ass2)

The above gives that $\overline{v_2} = \overline{w}$. This, together with (a) and (c) give that $\overline{v} = (\overline{y} \cap \overline{v});\overline{w}$

(8) By case analysis on whether $w \in \overline{x}$ ... etc

(9) By induction on the structure of A, and the guarantee from (8).

(10) We take a variable sequence $\overline{z}$ such that

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

This gives that

$(b)\quad \overline{x}\#\overline{z}$

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

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

Therefore

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

The above, together with (b) conclude the proof

□

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

Proof. (1) by induction on the number of elements in $\overline{x}$
(2) by induction on the number of elements in $\overline{y}$
(3) by induction on the number of elements in $\overline{y}$
(4) by induction on the number of elements in $\overline{y}$
(5) by induction on the structure of A
(6) by induction on the structure of A
(7) Assume that
$(ass1)$ $Fv(A) \; = \; \overline{x};\overline{v},$
$(ass2)$ $Fv(A[\overline{y/x}]) \; = \; \overline{y};\overline{w}$
We define:
(a) $\overline{y_0} \triangleq \overline{v} \cap \overline{y}, \quad \overline{v_2} \triangleq \overline{v} \setminus \overline{y}, \quad \overline{y_1} = \overline{y_0}[\overline{x/y}]$
This gives:
(b) $\overline{y_0}\#\overline{v_2}$
(c) $\overline{v} = \overline{y_0};\overline{v_2}$
(d) $\overline{y_1} \subseteq \overline{y}$
(e) $\overline{v_2}[\overline{y/x}] = \overline{v_2},$ from assumption and (a) we have $\overline{x}\#\overline{v_2}$ and by Lemma F.30) part (2)
We now calculate

$$
\begin{array}{rcll}
Fv(A[\overline{y/x}]) & = & (\overline{x};\overline{v})[\overline{y/x}] & \text{by (ass1), and Lemma F.30 part (5).} \\
 & = & (\overline{x};\overline{y_0};\overline{v_2})[\overline{y/x}] & \text{by (c) above} \\
 & = & \overline{x}[\overline{y/x}],\ \overline{y_0}[\overline{y/x}], \overline{v_2}[\overline{y/x}] & \text{by distributivity of [../..]} \\
 & = & \overline{y},\ \overline{y_1}, \overline{v_2} & \text{by Lemma F.30 part (1), (a), and (e).} \\
 & = & \overline{y};\overline{v_2} & \text{because (d), and } \overline{y}\#\overline{v_2} \\
Fv(A[\overline{y/x}]) & = & \overline{y};\overline{w} & \text{by (ass2)}
\end{array}
$$

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

□

Lemma G.31 (Substitutions and Adaptations)**.** For any sequences of variables $\overline{x}$, $\overline{y}$, sequences of expressions $\overline{e}$, and any assertion A, we have

- $\overline{x}\#\overline{y} \quad\Longrightarrow\quad (A[\overline{e/x}])\nabla\overline{y} \stackrel{\text{txt}}{=} (A\nabla\overline{y})[\overline{e/x}]$

Proof. We first consider A to be $\langle\!\langle e\rangle\!\rangle_0$, and just take one variable. Then,
$(\langle\!\langle e_0\rangle\!\rangle[e/x])\nabla\overline{y} \stackrel{\text{txt}}{=} \langle\!\langle e_0[e/x]\rangle\!\rangle \not\leftrightarrow y$,
and
$(\langle\!\langle e_0\rangle\!\rangle\nabla\overline{y})[e/x] \stackrel{\text{txt}}{=} \langle\!\langle e_0[e/x]\rangle\!\rangle \not\leftrightarrow y[e/x]$.
When $x\#y$ then the two assertions from above are textually equal. The rest follows by induction on the length of $\overline{x}$ and the structure of A. □

Lemma G.32. For assertion A, variables $\overline{x}$, $\overline{v}$, $\overline{y}$, $\overline{w}$, $\overline{v_1}$, addresses $\overline{\alpha_x}$, $\overline{\alpha_y}$, $\overline{\alpha_v}$, and $\overline{\alpha_{v_1}}$
If

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

then

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

Proof.
From Lemma G.30, part 7, we obtain $\quad (*)\ \ \overline{v} \ = \ (\overline{y}\cap\overline{v});\overline{w}$
We first prove that

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

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

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

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

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

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

G.10 Reachability, Heap Identity, and their properties

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

Definition G.33. For any state σ, addresses α, α', we define

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

Lemma G.34. For any module M, state σ, addresses α, α', α''

(1) $M,\sigma \models \langle\!\langle\alpha\rangle\!\rangle \not\leftrightarrow \alpha' \ \wedge\ Reach(\alpha',\alpha'')_\sigma \quad\Longrightarrow\quad M,\sigma \models \langle\!\langle\alpha\rangle\!\rangle \not\leftrightarrow \alpha''$

Lemma F.31 (Substitutions and Adaptations)**.** For any sequences of variables $\overline{x}$, $\overline{y}$, sequences of expressions $\overline{e}$, and any assertion A, we have

- $\overline{x}\#\overline{y} \quad\Longrightarrow\quad (A[\overline{e/x}]) -\!\nabla\, \overline{y} \stackrel{\text{txt}}{=} (A -\!\nabla\, \overline{y})[\overline{e/x}]$

Proof. We first consider A to be $\langle\!\langle e \rangle\!\rangle_0$, and just take one variable. Then,
$(\langle\!\langle e_0 \rangle\!\rangle [e/x]) -\!\nabla\, \overline{y} \ \stackrel{\text{txt}}{=}\ \langle\!\langle e_0[e/x] \rangle\!\rangle \not\leftrightarrow y$,
and
$(\langle\!\langle e_0 \rangle\!\rangle -\!\nabla\, \overline{y})[e/x] \ \stackrel{\text{txt}}{=}\ \langle\!\langle e_0[e/x] \rangle\!\rangle \not\leftrightarrow y[e/x]$.
When $x\#y$ then the two assertions from above are textually equal. The rest follows by induction on the length of $\overline{x}$ and the structure of A. □

Lemma F.32. For assertion A, variables $\overline{x}$, $\overline{v}$, $\overline{y}$, $\overline{w}$, $\overline{v_1}$, addresses $\overline{\alpha_x}$, $\overline{\alpha_y}$, $\overline{\alpha_v}$, and $\overline{\alpha_{v_1}}$
If

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

then

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

Proof.
From Lemma F.30, part 7, we obtain $\quad (*)\ \ \overline{v} \ = \ (\overline{y} \cap \overline{v});\overline{w}$
We first prove that

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

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

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

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

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

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

F.10 Reachability, Heap Identity, and their properties

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

Definition F.33. For any state σ, addresses α, α', we define

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

Lemma F.34. For any module M, state σ, addresses α, α', α''

(1) $M, \sigma \models \langle\!\langle \alpha \rangle\!\rangle \not\leftrightarrow \alpha' \ \wedge\ Reach(\alpha', \alpha'')_\sigma \quad\Longrightarrow\quad M, \sigma \models \langle\!\langle \alpha \rangle\!\rangle \not\leftrightarrow \alpha''$

(2) $\sigma \sim \sigma' \implies [\ Reach(\alpha, \alpha')_{\sigma} \iff Reach(\alpha, \alpha')_{\sigma'}\ ]$
(3) $\sigma \sim \sigma' \implies [\ M, \sigma \models \langle\!\langle \alpha \rangle\!\rangle \not\leftrightarrow \alpha'' \iff M, \sigma' \models \langle\!\langle \alpha \rangle\!\rangle \not\leftrightarrow \alpha''\ ]$
(4) $\sigma \sim \sigma' \wedge Fv(A) = \emptyset \wedge Stbl(A) \implies [\ M, \sigma \models A \iff M, \sigma' \models A\ ]$

Proof.

(1) By unfolding/folding the definitions
(2) By unfolding/folding the definitions
(3) By unfolding/folding definitions.
(4) By structural induction on A, and Lemma G.34 part 3.

□

G.11 Preservation of assertions when pushing or popping frames

In this section we discuss the preservation of satisfaction of assertions when calling methods or returning from methods – *i.e.* when pushing or popping frames. Namely, since pushing/popping frames does not modify the heap, these operations should preserve satisfaction of some assertion A, up to the fact that a) passing an object as a parameter of a a result might break its protection, and b) the bindings of variables change with pushing/popping frames. To deal with a) upon method call, we require that the fame being pushed or the frame to which we return is internal ($M, \sigma' \models$ `intl`), or require the adapted version of an assertion (*i.e.* $A \neg\!\nabla \overline{v}$ rather than A). To deal with b) we either require that there are no free variables in A, or we break the free variables of A into two parts, *i.e.* $Fv(A_{in}) = \overline{v_1}; \overline{v_2}$, where the value of $\overline{v_3}$ in the caller is the same as that of $\overline{v_1}$ in the called frame. This is described in lemmas G.39 - G.41.

We have four lemmas: Lemma G.39 describes preservation from a caller to an internal called, lemma G.40 describes preservation from a caller to any called, Lemma G.41 describes preservation from an internal called to a caller, and Lemma G.42 describes preservation from an any called to a caller, These four lemmas are used in the soundness proof for the four Hoare rules about method calls, as given in Fig. 7.

In the rest of this section we will first introduce some further auxiliary concepts and lemmas, and then state, discuss and prove Lemmas G.39 - G.41.

Plans for next three subsections Lemmas G.39-G.40 are quite complex, because they deal with substitution of some of the assertions' free variables. We therefore approach the proofs gradually: We first state and prove a very simplified version of Lemmas G.39-G.40, where the assertion (A_{in} or A_{out}) is only about protection and only contains addresses; this is the only basic assertion which is not *Stbl*. We then state a slightly more general version of Lemmas G.39-G.40, where the assertion (A_{in} or A_{out}) is variable-free.

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

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

Lemma G.35 (From caller to called - protected, and variable-free)**.** For any address α, addresses $\overline{\alpha}$, states σ, σ', and frame ϕ.
If $\sigma' = \sigma \triangledown \phi$ then

a. $M, \sigma, k \models \langle\!\langle \alpha \rangle\!\rangle \ \wedge\ M, \sigma' \models$ `intl` $\wedge\ Rng(\phi) \subseteq LocRchbl(\sigma) \implies M, \sigma', k \models \langle\!\langle \alpha \rangle\!\rangle$
b. $M, \sigma, k \models \langle\!\langle \alpha \rangle\!\rangle \not\leftrightarrow \overline{\alpha} \ \wedge\ Rng(\phi) \subseteq \overline{\alpha} \implies M, \sigma' \models \langle\!\langle \alpha \rangle\!\rangle$
c. $M, \sigma, k \models \langle\!\langle \alpha \rangle\!\rangle \wedge \langle\!\langle \alpha \rangle\!\rangle \not\leftrightarrow \overline{\alpha} \ \wedge\ Rng(\phi) \subseteq \overline{\alpha} \implies M, \sigma', k \models \langle\!\langle \alpha \rangle\!\rangle$

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

Proof.

(1) By unfolding/folding the definitions
(2) By unfolding/folding the definitions
(3) By unfolding/folding definitions.
(4) By structural induction on A, and Lemma F.34 part 3.

□

F.11 Preservation of assertions when pushing or popping frames

In this section we discuss the preservation of satisfaction of assertions when calling methods or returning from methods – *i.e.* when pushing or popping frames. Namely, since pushing/popping frames does not modify the heap, these operations should preserve satisfaction of some assertion A, up to the fact that a) passing an object as a parameter of a a result might break its protection, and b) the bindings of variables change with pushing/popping frames. To deal with a) upon method call, we require that the fame being pushed or the frame to which we return is internal ($M, \sigma' \models \texttt{intl}$), or require the adapted version of an assertion (*i.e.* $A \neg\!\nabla \overline{v}$ rather than A). To deal with b) we either require that there are no free variables in A, or we break the free variables of A into two parts, *i.e.* $Fv(A_{in}) = \overline{v_1}; \overline{v_2}$, where the value of $\overline{v_3}$ in the caller is the same as that of $\overline{v_1}$ in the called frame. This is described in lemmas F.39 - F.41.

We have four lemmas: Lemma F.39 describes preservation from a caller to an internal called, lemma F.40 describes preservation from a caller to any called, Lemma F.41 describes preservation from an internal called to a caller, and Lemma F.42 describes preservation from an any called to a caller, These four lemmas are used in the soundness proof for the four Hoare rules about method calls, as given in Fig. 7.

In the rest of this section we will first introduce some further auxiliary concepts and lemmas, and then state, discuss and prove Lemmas F.39 - F.41.

Plans for next three subsections Lemmas F.39-F.40 are quite complex, because they deal with substitution of some of the assertions' free variables. We therefore approach the proofs gradually: We first state and prove a very simplified version of Lemmas F.39-F.40, where the assertion (A_{in} or A_{out}) is only about protection and only contains addresses; this is the only basic assertion which is not *Stbl*. We then state a slightly more general version of Lemmas F.39-F.40, where the assertion (A_{in} or A_{out}) is variable-free.

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

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

Lemma F.35 (From caller to called - protected, and variable-free)**.** For any address α, addresses $\overline{\alpha}$, states σ, σ', and frame ϕ.
If $\sigma' = \sigma \triangledown \phi$ then

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

Proof.

a. (1) Take any $\alpha' \in LocRchbl(\sigma')$. Then, by assumptions, we have $\alpha' \in LocRchbl(\sigma)$. This gives, again by assumptions, that $M, \sigma \models \langle\alpha\rangle \leftrightarrow\!\!\!\times\ \alpha'$. By the construction of σ, and lemma G.34 part 1, we obtain that (2) $M, \sigma' \models \langle\alpha\rangle \leftrightarrow\!\!\!\times\ \alpha'$. From (1) and (2) and because $M, \sigma' \models \texttt{intl}$ we obtain that $M, \sigma' \models \langle\alpha\rangle$. Then apply lemma G.34 part G.4, and we are done.

b. By unfolding and folding the definitions, and application of Lemma G.34 part 1.

c. By part G.34 part b. and G.4.

Notice that part G.35 requires that the called (σ') is internal, but parts b. and c. do not.

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

□

Lemma G.36 (From called to caller – protected, and variable-free)**.** For any states σ, σ', variable v, address α_v, addresses $\overline{\alpha}$, and statement *stmt*.
If $\sigma' = (\sigma \vartriangle)[v \mapsto \alpha_v][\texttt{cont} \mapsto stmt]$,
then

a. $M, \sigma, k \models \langle\alpha\rangle \ \wedge\ k < |\sigma| \ \wedge\ M, \sigma \models \langle\alpha\rangle \leftrightarrow\!\!\!\times\ \alpha_v \implies M, \sigma', k \models \langle\alpha\rangle$.

b. $M, \sigma \models \langle\alpha\rangle \ \wedge\ \overline{\alpha} \subseteq LocRchbl(\sigma) \implies M, \sigma', k \models \langle\alpha\rangle \leftrightarrow\!\!\!\times\ \overline{\alpha}$.

Proof.

a. (1) Take any $i \in [k..|\sigma'|)$. Then, by definitions and assumption, we have $M, \sigma[i] \models \langle\alpha\rangle$. Take any $\alpha' \in LocRchbl(\sigma[i])$. We obtain that $M, \sigma[i] \models \langle\alpha\rangle \leftrightarrow\!\!\!\times\ \alpha'$. Therefore, $M, \sigma[i] \models \langle\alpha\rangle$. Moreover, $\sigma[i]=\sigma'[i]$, and we therefore obtain (2) $M, \sigma'[i] \models \langle\alpha\rangle$.
(3) Now take a $\alpha' \in LocRchbl(\sigma')$.
Then, we have that either (A): $\alpha' \in LocRchbl(\sigma[|\sigma'|])$, or (B): $Reach(\alpha_r, \alpha')_{\sigma'}$.
In the case of (A): Because $k, |\sigma| = |\sigma'| + 1$, and because $M, \sigma, k \models \langle\alpha\rangle$ we have $M, \sigma \models \langle\alpha\rangle \leftrightarrow\!\!\!\times\ \alpha'$. Because $\sigma \sim \sigma'$ and Lemma G.34 part 3, we obtain (A') $M, \sigma' \models \langle\alpha\rangle \leftrightarrow\!\!\!\times\ \alpha'$
In the case of (B): Because $\sigma \sim \sigma'$ and lemma G.34 part 2, we obtain $Reach(\alpha_r, \alpha')_{\sigma}$. Then, applying Lemma G.34 part 3 and assumptions, we obtain (B') $M, \sigma' \models \langle\alpha\rangle \leftrightarrow\!\!\!\times\ \alpha'$.
From (3), (A), (A'), (B) and (B') we obtain: (4) $M, \sigma' \models \langle\alpha\rangle$.
With (1), (2), (4) and Lemma G.34 part 4 we are done.

b. From the definitions we obtain that $M, \sigma \models \langle\alpha\rangle \leftrightarrow\!\!\!\times\ \overline{\alpha}$. Because $\sigma \sim \sigma'$ and lemma G.34 part 3, we obtain $M, \sigma' \models \langle\alpha\rangle \leftrightarrow\!\!\!\times\ \overline{\alpha}$. And because of lemma G.3, part 3, we obtain $M, \sigma', k \models \langle\alpha\rangle \leftrightarrow\!\!\!\times\ \overline{\alpha}$.

□

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

We now move consider preservation of variable-free assertions when pushing/popping frames, and generalize the lemmas G.35 and G.36

Lemma G.37 (From caller to called - variable-free, and *Stbl*$^+$)**.** For any assertion A, addresses $\overline{\alpha}$, states σ, σ', and frame ϕ.
If $\sigma' = \sigma \triangledown \phi$ and $Stb^+(A)$, and $Fv(A) = \emptyset$,
then

a. $M, \sigma, k \models A \ \wedge\ M, \sigma' \models \texttt{intl} \ \wedge\ Rng(\phi) \subseteq LocRchbl(\sigma) \implies M, \sigma', k \models A$

b. $M, \sigma, k \models A \neg\!\triangledown(\overline{\alpha}) \ \wedge\ Rng(\phi) \subseteq \overline{\alpha} \implies M, \sigma' \models A$

c. $M, \sigma, k \models A \wedge A \neg\!\triangledown(\overline{\alpha}) \ \wedge\ Rng(\phi) \subseteq \overline{\alpha} \implies M, \sigma', k \models A$

Proof.

PROOF.

a. (1) Take any $\alpha' \in LocRchbl(\sigma')$. Then, by assumptions, we have $\alpha' \in LocRchbl(\sigma)$. This gives, again by assumptions, that $M, \sigma \models \langle\!\langle \alpha \rangle\!\rangle \leftrightarrow\!\!\!\!\times\ \alpha'$. By the construction of σ, and lemma F.34 part 1, we obtain that (2) $M, \sigma' \models \langle\!\langle \alpha \rangle\!\rangle \leftrightarrow\!\!\!\!\times\ \alpha'$. From (1) and (2) and because $M, \sigma' \models \texttt{intl}$ we obtain that $M, \sigma' \models \langle\!\langle \alpha \rangle\!\rangle$. Then apply lemma F.34 part F.4, and we are done.

b. By unfolding and folding the definitions, and application of Lemma F.34 part 1.

c. By part F.34 part b. and F.4.

Notice that part F.35 requires that the called (σ') is internal, but parts b. and c. do not.

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

□

Lemma F.36 (From called to caller – protected, and variable-free)**.** For any states σ, σ', variable v, address α_v, addresses $\overline{\alpha}$, and statement $stmt$.
If $\sigma' = (\sigma \vartriangle)[v \mapsto \alpha_v][\texttt{cont} \mapsto stmt]$,
then

a. $M, \sigma, k \models \langle\!\langle \alpha \rangle\!\rangle \ \wedge\ k < |\sigma| \ \wedge\ M, \sigma \models \langle\!\langle \alpha \rangle\!\rangle \leftrightarrow\!\!\!\!\times\ \alpha_v \quad\Longrightarrow\quad M, \sigma', k \models \langle\!\langle \alpha \rangle\!\rangle$.

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

PROOF.

a. (1) Take any $i \in [k..|\sigma'|)$. Then, by definitions and assumption, we have $M, \sigma[i] \models \langle\!\langle \alpha \rangle\!\rangle$. Take any $\alpha' \in LocRchbl(\sigma[i])$. We obtain that $M, \sigma[i] \models \langle\!\langle \alpha \rangle\!\rangle \leftrightarrow\!\!\!\!\times\ \alpha'$. Therefore, $M, \sigma[i] \models \langle\!\langle \alpha \rangle\!\rangle$. Moreover, $\sigma[i] = \sigma'[i]$, and we therefore obtain (2) $M, \sigma'[i] \models \langle\!\langle \alpha \rangle\!\rangle$.
(3) Now take a $\alpha' \in LocRchbl(\sigma')$.
Then, we have that either (A): $\alpha' \in LocRchbl(\sigma[|\sigma'|])$, or (B): $Reach(\alpha_r, \alpha')_{\sigma'}$.
In the case of (A): Because $k, |\sigma| = |\sigma'| + 1$, and because $M, \sigma, k \models \langle\!\langle \alpha \rangle\!\rangle$ we have $M, \sigma \models \langle\!\langle \alpha \rangle\!\rangle \leftrightarrow\!\!\!\!\times\ \alpha'$. Because $\sigma \sim \sigma'$ and Lemma F.34 part 3, we obtain (A') $M, \sigma' \models \langle\!\langle \alpha \rangle\!\rangle \leftrightarrow\!\!\!\!\times\ \alpha'$
In the case of (B): Because $\sigma \sim \sigma'$ and lemma F.34 part 2, we obtain $Reach(\alpha_r, \alpha')_\sigma$. Then, applying Lemma F.34 part 3 and assumptions, we obtain (B') $M, \sigma' \models \langle\!\langle \alpha \rangle\!\rangle \leftrightarrow\!\!\!\!\times\ \alpha'$.
From (3), (A), (A'), (B) and (B') we obtain: (4) $M, \sigma' \models \langle\!\langle \alpha \rangle\!\rangle$.
With (1), (2), (4) and Lemma F.34 part 4 we are done.

b. From the definitions we obtain that $M, \sigma \models \langle\!\langle \alpha \rangle\!\rangle \leftrightarrow\!\!\!\!\times\ \overline{\alpha}$. Because $\sigma \sim \sigma'$ and lemma F.34 part 3, we obtain $M, \sigma' \models \langle\!\langle \alpha \rangle\!\rangle \leftrightarrow\!\!\!\!\times\ \overline{\alpha}$. And because of lemma F.3, part 3, we obtain $M, \sigma', k \models \langle\!\langle \alpha \rangle\!\rangle \leftrightarrow\!\!\!\!\times\ \overline{\alpha}$.

□

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

We now move consider preservation of variable-free assertions when pushing/popping frames, and generalize the lemmas F.35 and F.36

Lemma F.37 (From caller to called - variable-free, and $Stbl^+$)**.** For any assertion A, addresses $\overline{\alpha}$, states σ, σ', and frame ϕ.
If $\sigma' = \sigma \triangledown \phi$ and $Stb^+(A)$, and $Fv(A) = \emptyset$,
then

a. $M, \sigma, k \models A \ \wedge\ M, \sigma' \models \texttt{intl} \ \wedge\ Rng(\phi) \subseteq LocRchbl(\sigma) \quad\Longrightarrow\quad M, \sigma', k \models A$

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

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

PROOF.

a. By Lemma G.35, part G.35 and structural induction on the definition of $Stb^{+}(\_)$.
b. By Lemma G.35, part G.35 and structural induction on the definition of $Stb^{+}(\_)$.
c. By part b. and Lemma G.3.

□

Lemma G.38 (From called to caller – protected, and variable-free)**.** For any states σ, σ', variable v, address α_v, addresses $\overline{\alpha}$, and statement $stmt$.
If $\sigma' = (\sigma \vartriangle)[v \mapsto \alpha_v][\texttt{cont} \mapsto stmt]$, and $Stb^{+}(A)$, and $Fv(A) = \emptyset$
then

a. $M, \sigma, k \models A \quad \wedge \quad k < |\sigma| \quad \wedge \quad M, \sigma \models A \text{-}\!\!\triangledown \alpha_v \qquad \Longrightarrow \qquad M, \sigma', k \models A$.
b. $M, \sigma \models A \quad \wedge \quad \overline{\alpha} \subseteq LocRchbl(\sigma) \qquad \Longrightarrow \qquad M, \sigma', k \models A \text{-}\!\!\triangledown (\overline{\alpha})$.

Proof.

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

□

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

Lemma G.39 (From caller to internal called)**.** For any assertion A_{in}, states σ, σ', variables $\overline{v_1}$, $\overline{v_2}$, $\overline{v_3}$, $\overline{v_4}$, $\overline{v_6}$, and frame ϕ.
If

(i) $Stb^{+}(A_{in})$,
(ii) $Fv(A_{in}) = \overline{v_1}; \overline{v_2}$[15], $\quad Fv(A_{in}[\overline{v_3/v_1}]) = \overline{v_3}; \overline{v_4}, \quad \overline{v_6} \triangleq \overline{v_2} \cap \overline{v_3}; \overline{v_4}$,
(iii) $\sigma' = \sigma \triangledown \phi \quad \wedge \quad Rng(\phi) = \overline{\lfloor v_3 \rfloor_\sigma}$
(iv) $\overline{\lfloor v_1 \rfloor_{\sigma'}} = \overline{\lfloor v_3 \rfloor_\sigma}$,

then

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

Discussion of Lemma. In lemma G.39, state σ is the state right before pushing the new frame on the stack, while state σ' is the state right after pushing the frame on the stack. That is, σ is the last state before entering the method body, and σ' is the first state after entering the method body. A_{in} stands for the method's precondition, while the variables $\overline{v_1}$ stand for the formal parameters of the method, and $\overline{v_3}$ stand for the actual parameters of the call. Therefore, $\overline{v_1}$ is the domain of the new frame, and $\overline{\sigma}v_3$ is its range. The variables $\overline{v_6}$ are the free variables of A_{in} which are not in $\overline{v_1}$ – *c.f.* Lemma G.30 part (7). Therefore if (a.) the callee is internal, and $A_{in}[\overline{v_3/v_1}]$ holds at the call point, or if (b.) $(A_{in}[\overline{v_3/v_1}]) \text{-}\!\!\triangledown (\overline{v_3})$ holds at the call point, then $A_{in}[\overline{.../v_6}1]$ holds right after pushing ϕ onto the stack. Notice the difference in the conclusion in (a.) and (b.): in the first case we have scoped satisfaction, while in the second case we only have shallow satisfaction.

Proof.
We will use $\overline{\alpha_1}$ as short for $\{\overline{\lfloor v_1 \rfloor_{\sigma'}}$, and $\overline{\alpha_3}$ as short for $\overline{\lfloor v_3 \rfloor_\sigma}$.
We aslo define $\overline{v_{6,1}} \triangleq \overline{v_2} \cap \overline{v_3}, \quad \overline{\alpha_{6,1}} \triangleq \overline{v_{6,1}}[\overline{\lfloor v_6 \rfloor_\sigma / v_6}]$
We establish that

$(*) \quad A_{in}[\overline{v_3/v_1}][\overline{\lfloor v_3 \rfloor_\sigma / v_3}] \stackrel{\text{txt}}{=} A_{in}[\overline{\alpha_1/v_1}][\overline{\alpha_{6,1}/v_{6,1}}]$

This holds by By Lemma G.32 and assumption (iv) of the current lemma.
And we define $\overline{v_{6,2}} \triangleq \overline{v_2} \setminus \overline{v_3}, \quad \overline{\alpha_{6,2}} \triangleq \overline{v_{6_2}}[\overline{\lfloor v_6 \rfloor_\sigma / v_6}]$.

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

a. By Lemma F.35, part F.35 and structural induction on the definition of $Stb^+(\_)$.
b. By Lemma F.35, part F.35 and structural induction on the definition of $Stb^+(\_)$.
c. By part b. and Lemma F.3.

□

Lemma F.38 (From called to caller – protected, and variable-free)**.** For any states σ, σ', variable v, address α_v, addresses $\overline{\alpha}$, and statement $stmt$.
If $\sigma' = (\sigma \vartriangle)[v \mapsto \alpha_v][\mathtt{cont} \mapsto stmt]$, and $Stb^+(A)$, and $Fv(A) = \emptyset$
then

a. $M, \sigma, k \models A \ \wedge \ k < |\sigma| \ \wedge \ M, \sigma \models A \neg\!\triangledown \alpha_v \quad \Longrightarrow \quad M, \sigma', k \models A$.
b. $M, \sigma \models A \ \wedge \ \overline{\alpha} \subseteq LocRchbl(\sigma) \quad \Longrightarrow \quad M, \sigma', k \models A \neg\!\triangledown (\overline{\alpha})$.

Proof.

a. By Lemma F.36, part a. and structural induction on the definition of $Stb^+(\_)$.
b. By Lemma F.36, part b. and structural induction on the definition of $Stb^+(\_)$.

□

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

Lemma F.39 (From caller to internal called)**.** For any assertion A_{in}, states σ, σ', variables $\overline{v_1}$, $\overline{v_2}$, $\overline{v_3}$, $\overline{v_4}$, $\overline{v_6}$, and frame ϕ.
If

(i) $Stb^+(A_{in})$,
(ii) $Fv(A_{in}) = \overline{v_1};\overline{v_2}$[15], $\quad Fv(A_{in}[\overline{v_3/v_1}]) = \overline{v_3};\overline{v_4}, \quad \overline{v_6} \triangleq \overline{v_2} \cap \overline{v_3};\overline{v_4}$,
(iii) $\sigma' = \sigma \triangledown \phi \quad \wedge \quad Rng(\phi) = \overline{\lfloor v_3 \rfloor_\sigma}$
(iv) $\overline{\lfloor v_1 \rfloor_{\sigma'}} = \overline{\lfloor v_3 \rfloor_\sigma}$,

then

a. $M, \sigma, k \models A_{in}[\overline{v_3/v_1}] \ \wedge \ M, \sigma' \models \mathtt{intl} \quad \Longrightarrow \quad M, \sigma', k \models A_{in}[\overline{\lfloor v_6 \rfloor_\sigma / v_6}]$
b. $M, \sigma, k \models (A_{in}[\overline{v_3/v_1}]) \neg\!\triangledown (\overline{v_3}) \quad \Longrightarrow \quad M, \sigma' \models A_{in}[\overline{v_6}]_\sigma/\overline{v_6}]$.

Discussion of Lemma. In lemma F.39, state σ is the state right before pushing the new frame on the stack, while state σ' is the state right after pushing the frame on the stack. That is, σ is the last state before entering the method body, and σ' is the first state after entering the method body. A_{in} stands for the method's precondition, while the variables $\overline{v_1}$ stand for the formal parameters of the method, and $\overline{v_3}$ stand for the actual parameters of the call. Therefore, $\overline{v_1}$ is the domain of the new frame, and $\overline{\sigma}v_3$ is its range. The variables $\overline{v_6}$ are the free variables of A_{in} which are not in $\overline{v_1}$ – *c.f.* Lemma F.30 part (7). Therefore if (a.) the callee is internal, and $A_{in}[\overline{v_3/v_1}]$ holds at the call point, or if (b.) $(A_{in}[\overline{v_3/v_1}]) \neg\!\triangledown (\overline{v_3})$ holds at the call point, then $A_{in}[.../v_61]$ holds right after pushing ϕ onto the stack. Notice the difference in the conclusion in (a.) and (b.): in the first case we have scoped satisfaction, while in the second case we only have shallow satisfaction.

Proof.
We will use $\overline{\alpha_1}$ as short for $\{\overline{\lfloor v_1 \rfloor_{\sigma'}}$, and $\overline{\alpha_3}$ as short for $\overline{\lfloor v_3 \rfloor_\sigma}$.
We aslo define $\overline{v_{6,1}} \triangleq \overline{v_2} \cap \overline{v_3}, \quad \overline{\alpha_{6,1}} \triangleq \overline{v_{6,1}}[\overline{\lfloor v_6 \rfloor_\sigma / v_6}]$
We establish that
$(*) \quad A_{in}[\overline{v_3/v_1}][\overline{\lfloor v_3 \rfloor_\sigma / v_3}] \stackrel{\text{txt}}{=} A_{in}[\overline{\alpha_1/v_1}][\overline{\alpha_{6,1}/v_{6,1}}]$
This holds by By Lemma F.32 and assumption (iv) of the current lemma.
And we define $\overline{v_{6,2}} \triangleq \overline{v_2} \setminus \overline{v_3}, \quad \overline{\alpha_{6,2}} \triangleq \overline{v_{6_2}}[\overline{\lfloor v_6 \rfloor_\sigma / v_6}]$.

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

a. Assume
$M, \sigma, k \models A_{in}[\overline{v_3/v_1}]$. By Lemma 6.1 part 1 this implies that
$M, \sigma, k \models A_{in}[\overline{v_3/v_1}][\overline{\alpha_3/v_3}]$ By (*) from above we have
$M, \sigma, k \models A_{in}[\overline{\alpha_1/v_1}[\overline{\alpha_{6,1}/v_{6,1}}]$
The above, and Lemma 6.1 part 1 give that
$M, \sigma, k \models A_{in}[\overline{\alpha_1/v_1}][\overline{\alpha_{6,1}/v_{6,1}}][\overline{\alpha_{6,2}/v_{6,2}}]$
The assertion above is variable-free.Therefore, by Lemma G.37 part a. we also obtain
$M, \sigma', k \models A_{in}[\overline{\alpha_1/v_1}][\overline{\alpha_{6,1}/v_{6,1}}][\overline{\alpha_{6,2}/v_{6,2}}]$
With 6.1 part 1 the above gives
$M, \sigma', k \models A_{in}[\overline{\lfloor v_1 \rfloor_{\sigma'}/v_1}][\overline{\lfloor v_6 \rfloor_{\sigma}/v_6}]$
By Lemma 6.1 part 1, we obtain
$M, \sigma', k \models A_{in}[\overline{\lfloor v_6 \rfloor_{\sigma}/v_6}]$

b. Assume
$M, \sigma, k \models (A_{in}[\overline{v_3/v_1}]) \text{-}\!\nabla(\overline{v_3})$. By Lemma 6.1 part 1 this implies that
$M, \sigma, k \models ((A_{in}[\overline{v_3/v_1}]) \text{-}\!\nabla(\overline{v_3}))[\overline{\alpha_3/v_3}]$ which implies that
$M, \sigma, k \models (A_{in}[\overline{v_3/v_1}][\overline{\alpha_3/v_3}]) \text{-}\!\nabla(\overline{\alpha_3})$ By (*) from above we have
$M, \sigma, k \models (A_{in}[\overline{\alpha_1/v_1}][\overline{\alpha_{6,1}/v_{6,1}}]) \text{-}\!\nabla(\overline{\alpha_3})$
The above, and Lemma 6.1 part 1 give that
$M, \sigma, k \models ((A_{in}[\overline{\alpha_1/v_1}][\overline{\alpha_{6,1}/v_{6,1}}]) \text{-}\!\nabla(\overline{\alpha_3}))[\overline{\alpha_{6,2}/v_{6,2}}]$
And the above gives
$M, \sigma, k \models (A_{in}[\overline{\alpha_1/v_1}][\overline{\alpha_{6,1}/v_{6,1}}][\overline{\alpha_{6,2}/v_{6,2}}]) \text{-}\!\nabla(\overline{\alpha_3})$
The assertion above is variable-free.Therefore, by Lemma G.37 part b. we also obtain
$M\sigma' \models A_{in}[\overline{\alpha_1/v_1}][\overline{\alpha_{6,1}/v_{6,1}}][\overline{\alpha_{6,2}/v_{6,2}}]$
We apply Lemma 6.1 part 1, and Lemma 6.1 part 1, and obtain
$M, \sigma' \models A_{in}[\overline{\lfloor v_6 \rfloor_{\sigma}/v_6}]$

□

Lemma G.40 (From caller to any called)**.** For any assertion A_{in}, states σ, σ', variables $\overline{v_3}$ statement $stmt$, and frame ϕ.
If

(i) $Stb^+(A_{in})$,
(ii) $Fv(A_{in}) = \emptyset$,
(iii) $\sigma' = \sigma \triangledown \phi \quad \wedge \quad Rng(\phi) = \overline{\lfloor v_3 \rfloor_{\sigma}}$,

then

a. $M, \sigma, k \models A_{in} \text{-}\!\nabla(\overline{v_3}) \quad \Longrightarrow \quad M, \sigma' \models A_{in}$.
b. $M, \sigma, k \models (A_{in} \wedge (A_{in} \text{-}\!\nabla(\overline{v_3}))) \quad \Longrightarrow \quad M, \sigma', k \models A_{in}$

Proof. a. Assume that
$M, \sigma, k \models A_{in} \text{-}\!\nabla(\overline{v_3})$
By Lemma 6.1 part 1 this implies that
$M, \sigma, k \models A_{in} \text{-}\!\nabla(\lfloor v_3 \rfloor_{\sigma})$.
We now have a variable-free assertion, and by Lemma G.37, part b., we obtain
$M, \sigma' \models A_{in}$.

b. Assume that
$M, \sigma, k \models A_{in} \wedge A_{in} \text{-}\!\nabla(\overline{v_3})$
By Lemma 6.1 part 1 this implies that

a. Assume
$M, \sigma, k \models A_{in}[\overline{v_3/v_1}]$. By Lemma 6.1 part 1 this implies that
$M, \sigma, k \models A_{in}[\overline{v_3/v_1}][\overline{\alpha_3/v_3}]$ By (*) from above we have
$M, \sigma, k \models A_{in}[\overline{\alpha_1/v_1}[\overline{\alpha_{6,1}/v_{6,1}}]$
The above, and Lemma 6.1 part 1 give that
$M, \sigma, k \models A_{in}[\overline{\alpha_1/v_1}][\overline{\alpha_{6,1}/v_{6,1}}][\overline{\alpha_{6,2}/v_{6,2}}]$
The assertion above is variable-free.Therefore, by Lemma F.37 part a. we also obtain
$M, \sigma', k \models A_{in}[\overline{\alpha_1/v_1}][\overline{\alpha_{6,1}/v_{6,1}}][\overline{\alpha_{6,2}/v_{6,2}}]$
With 6.1 part 1 the above gives
$M, \sigma', k \models A_{in}[\overline{\lfloor v_1 \rfloor_{\sigma'}/v_1}][\overline{\lfloor v_6 \rfloor_{\sigma}/v_6}]$
By Lemma 6.1 part 1, we obtain
$M, \sigma', k \models A_{in}[\overline{\lfloor v_6 \rfloor_{\sigma}/v_6}]$

b. Assume
$M, \sigma, k \models (A_{in}[\overline{v_3/v_1}]) \mathrel{-\!\!\nabla} (\overline{v_3})$. By Lemma 6.1 part 1 this implies that
$M, \sigma, k \models ((A_{in}[\overline{v_3/v_1}]) \mathrel{-\!\!\nabla} (\overline{v_3}))[\overline{\alpha_3/v_3}]$ which implies that
$M, \sigma, k \models (A_{in}[\overline{v_3/v_1}][\overline{\alpha_3/v_3}]) \mathrel{-\!\!\nabla} (\overline{\alpha_3})$ By (*) from above we have
$M, \sigma, k \models (A_{in}[\overline{\alpha_1/v_1}][\overline{\alpha_{6,1}/v_{6,1}}]) \mathrel{-\!\!\nabla} (\overline{\alpha_3})$
The above, and Lemma 6.1 part 1 give that
$M, \sigma, k \models ((A_{in}[\overline{\alpha_1/v_1}][\overline{\alpha_{6,1}/v_{6,1}}]) \mathrel{-\!\!\nabla} (\overline{\alpha_3}))[\overline{\alpha_{6,2}/v_{6,2}}]$
And the above gives
$M, \sigma, k \models (A_{in}[\overline{\alpha_1/v_1}][\overline{\alpha_{6,1}/v_{6,1}}][\overline{\alpha_{6,2}/v_{6,2}}]) \mathrel{-\!\!\nabla} (\overline{\alpha_3})$
The assertion above is variable-free.Therefore, by Lemma F.37 part b. we also obtain
$M\sigma' \models A_{in}[\overline{\alpha_1/v_1}][\overline{\alpha_{6,1}/v_{6,1}}][\overline{\alpha_{6,2}/v_{6,2}}]$
We apply Lemma 6.1 part 1, and Lemma 6.1 part 1, and obtain
$M, \sigma' \models A_{in}[\overline{\lfloor v_6 \rfloor_{\sigma}/v_6}]$

□

Lemma F.40 (From caller to any called). For any assertion A_{in}, states σ, σ', variables $\overline{v_3}$ statement $stmt$, and frame ϕ.
If

(i) $Stb^+(A_{in})$,
(ii) $Fv(A_{in}) = \emptyset$,
(iii) $\sigma' = \sigma \triangledown \phi \quad \land \quad Rng(\phi) = \overline{\lfloor v_3 \rfloor_{\sigma}}$,

then

a. $M, \sigma, k \models A_{in} \mathrel{-\!\!\nabla} (\overline{v_3}) \quad \Longrightarrow \quad M, \sigma' \models A_{in}$.
b. $M, \sigma, k \models (A_{in} \land (A_{in} \mathrel{-\!\!\nabla} (\overline{v_3}))) \quad \Longrightarrow \quad M, \sigma', k \models A_{in}$

Proof. a. Assume that
$M, \sigma, k \models A_{in} \mathrel{-\!\!\nabla} (\overline{v_3})$
By Lemma 6.1 part 1 this implies that
$M, \sigma, k \models A_{in} \mathrel{-\!\!\nabla} (\overline{\lfloor v_3 \rfloor_{\sigma}})$.
We now have a variable-free assertion, and by Lemma F.37, part b., we obtain
$M, \sigma' \models A_{in}$.

b. Assume that
$M, \sigma, k \models A_{in} \land A_{in} \mathrel{-\!\!\nabla} (\overline{v_3})$
By Lemma 6.1 part 1 this implies that

$M, \sigma, k \models A_{in} \wedge A_{in} \neg\nabla(\lfloor v_3 \rfloor_\sigma)$.
We now have a variable-free assertion, and by Lemma G.37, part b., we obtain
$M, \sigma', k \models \models A_{in}$.

□

Discussion of Lemma G.40. In this lemma, as in lemma G.39, σ stands for the last state before entering the method body, and σ' for the first state after entering the method body. A_{in} stands for a module invariant in which all free variables have been substituted by addresses. The lemma is intended for external calls, and therefore we have no knowledge of the method's formal parameters. The variables $\overline{v_3}$ stand for the actual parameters of the call, and therefore $\overline{\lfloor v_3 \rfloor_\sigma}$ is the range of the new frame. Therefore if (a.) the adapted version, $A_{in} \neg\nabla(\overline{v_3})$, holds at the call point, then the unadapted version, A_{in} holds right after pushing ϕ onto the stack. Notice that even though the premise of (a.) requires scoped satisfaction, the conclusion promises only weak satisfaction. Moreover, if (b.) the adapted as well as the unadapted version, $A_{in} \wedge A_{in} \neg\nabla(\overline{v_3})$ holds at the call point, then the unadapted version, A_{in} holds right after pushing ϕ onto the stack. Notice the difference in the conclusion in (a.) and (b.): in the first case we have shallow satisfaction, while in the second case we only have scoped satisfaction.

Lemma G.41 (From internal called to caller)**.** For any assertion A_{out}, states σ, σ', variables res, u variable sequences $\overline{v_1}$, $\overline{v_3}$, $\overline{v_5}$, and statement $stmt$.
If

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

then

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

Discussion of Lemma G.41. State σ stands for the last state in the method body, and σ' for the first state after exiting the method call. A_{out} stands for a method postcondition. The lemma is intended for internal calls, and therefore we know the method's formal parameters. The variables $\overline{v_1}$ stand for the formal parameters of the method, and $\overline{v_3}$ stand for the actual parameters of the call. Therefore the formal parameters of the called have the same values as the actual parameters in the caller $\overline{\lfloor v_1 \rfloor_\sigma} = \overline{\lfloor v_3 \rfloor_{\sigma'}}$. Therefore (a.) and (b.) promise that if the postcondition A_{out} holds before popping the frame, then it also holds after popping frame after replacing the the formal parameters by the actual parameters $A_{out}[\overline{v_3/v_1}]$. As in earlier lemmas, there is an important difference between (a.) and (b.): In (a.), we require *deep satisfaction at the called*, and obtain at the deep satisfaction of the *unadapted* version ($A_{out}[\overline{v_3/v_1}]$) at the return point; while in (b.), we only require *shallow satisfaction at the called*, and obtain deep satisfaction of the *adapted* version ($(A_{out}[\overline{v_3/v_1}]) \neg\nabla \overline{v_5}$), at the return point.

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

a. Assume that
$M, \sigma, k \models A_{out} \wedge A_{out} \neg\nabla res$
By Lemma 6.1 part 1 this implies that
$M, \sigma, k \models A_{out}[\overline{\alpha_1/v_1}] \wedge (A_{out}[\overline{\alpha_1/v_1}]) \neg\nabla \alpha$.

$M, \sigma, k \models A_{in} \wedge A_{in} -\!\!\nabla (\overline{\lfloor v_3 \rfloor_\sigma})$.
We now have a variable-free assertion, and by Lemma F.37, part b., we obtain
$M, \sigma', k \models\, \models A_{in}$.

$\square$

Discussion of Lemma F.40. In this lemma, as in lemma F.39, σ stands for the last state before entering the method body, and σ' for the first state after entering the method body. A_{in} stands for a module invariant in which all free variables have been substituted by addresses. The lemma is intended for external calls, and therefore we have no knowledge of the method's formal parameters. The variables $\overline{v_3}$ stand for the actual parameters of the call, and therefore $\overline{\lfloor v_3 \rfloor_\sigma}$ is the range of the new frame. Therefore if (a.) the adapted version, $A_{in} -\!\!\nabla (\overline{v_3})$, holds at the call point, then the unadapted version, A_{in} holds right after pushing ϕ onto the stack. Notice that even though the premise of (a.) requires scoped satisfaction, the conclusion promises only weak satisfaction. Moreover, if (b.) the adapted as well as the unadapted version, $A_{in} \wedge A_{in} -\!\!\nabla (\overline{v_3})$ holds at the call point, then the unadapted version, A_{in} holds right after pushing ϕ onto the stack. Notice the difference in the conclusion in (a.) and (b.): in the first case we have shallow satisfaction, while in the second case we only have scoped satisfaction.

Lemma F.41 (From internal called to caller)**.** For any assertion A_{out}, states σ, σ', variables res, u variable sequences $\overline{v_1}$, $\overline{v_3}$, $\overline{v_5}$, and statement $stmt$.
If

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

then

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

Discussion of Lemma F.41. State σ stands for the last state in the method body, and σ' for the first state after exiting the method call. A_{out} stands for a method postcondition. The lemma is intended for internal calls, and therefore we know the method's formal parameters. The variables $\overline{v_1}$ stand for the formal parameters of the method, and $\overline{v_3}$ stand for the actual parameters of the call. Therefore the formal parameters of the called have the same values as the actual parameters in the caller $\overline{\lfloor v_1 \rfloor_\sigma} = \overline{\lfloor v_3 \rfloor_{\sigma'}}$. Therefore (a.) and (b.) promise that if the postcondition A_{out} holds before popping the frame, then it also holds after popping frame after replacing the the formal parameters by the actual parameters $A_{out}[\overline{v_3/v_1}]$. As in earlier lemmas, there is an important difference between (a.) and (b.): In (a.), we require *deep satisfaction at the called*, and obtain at the deep satisfaction of the *unadapted* version $(A_{out}[\overline{v_3/v_1}])$ at the return point; while in (b.), we only require *shallow satisfaction at the called*, and obtain deep satisfaction of the *adapted* version $((A_{out}[\overline{v_3/v_1}]) -\!\!\nabla \overline{v_5})$, at the return point.

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

a. Assume that
$M, \sigma, k \models A_{out} \wedge A_{out} -\!\!\nabla res$
By Lemma 6.1 part 1 this implies that
$M, \sigma, k \models A_{out}[\overline{\alpha_1/v_1}] \wedge (A_{out}[\overline{\alpha_1/v_1}]) -\!\!\nabla \alpha$.

We now have a variable-free assertion, and by Lemma G.38, part a., we obtain
$M, \sigma, k \models A_{out}[\overline{\alpha_1/v_1}]$.
By Lemma 6.1 part 1, and because $\overline{\lfloor v_1 \rfloor_\sigma} = \overline{\lfloor v_3 \rfloor_{\sigma'}}$ this implies that
$M, \sigma, k \models A_{out}[\overline{v_3/v_1}]$.

b. Assume that
$M, \sigma \models A_{out}$
By Lemma 6.1 part 1 this implies that
$M, \sigma \models A_{out}[\overline{\alpha_1/v_1}]$
We now have a variable-free assertion, and by Lemma G.38, part b., we obtain
$M, \sigma', k \models A_{out}[\overline{\alpha_1/v_1}] \mathbin{-\!\triangledown} \overline{\alpha}_5$
By Lemma 6.1 part 1, and because $\overline{\lfloor v_1 \rfloor_\sigma} = \overline{\lfloor v_3 \rfloor_{\sigma'}}$ and $\alpha_5 = \overline{\lfloor v_5 \rfloor_{\sigma'}}$, we obtain
$M, \sigma', k \models A_{out}[\overline{v_3/v_1}] \mathbin{-\!\triangledown} \overline{v}_5$

∎

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

(i) $Stb^+(A_{out})$,
(ii) $Fv(A_{out}) = \emptyset$,
(iii) $\overline{\lfloor v_5 \rfloor_{\sigma'}}, \lfloor res \rfloor_\sigma \subseteq LocRchbl(\sigma)$.
(iv) $\sigma' = (\sigma \bigtriangleup)[u \mapsto \lfloor res \rfloor_\sigma][\texttt{cont} \mapsto stmt]$.

then

a. $M, \sigma \models A_{out} \quad \Longrightarrow \quad M, \sigma', k \models A_{out} \mathbin{-\!\triangledown} (\overline{v_5})$.
b. $M, \sigma, k \models A_{out} \;\wedge\; |\sigma'| \geq k \quad \Longrightarrow \quad M, \sigma', k \models A_{out} \;\wedge\; A_{out} \mathbin{-\!\triangledown} (\overline{v_5})$

Proof.

a. Assume that
$M, \sigma \models A_{out}$
Since A_{out} is a variable-free assertion, by Lemma G.38, part a., we obtain
$M, \sigma', k \models A_{out} \mathbin{-\!\triangledown} (\overline{\lfloor v_5 \rfloor_{\sigma'}})$.
By Lemma 6.1 part 1, we obtain
$M, \sigma', k \models A_{out} \mathbin{-\!\triangledown} (\overline{v_5})$

b. Similar argument to the proof of Lemma G.41, part (b).

∎

Discussion of lemma G.42. , Similarly to lemma G.41, in this lemma, state σ stands for the last state in the method body, and σ' for the first state after exiting the method call. A_{out} stands for a method postcondition. The lemma is meant to apply to external calls, and therefore, we do not know the method's formal parameters, A_{out} is meant to stand for a module invariant where all the free variables have been substituted by addresses – *i.e.* A_{out} has no free variables. The variables $\overline{v_3}$ stand for the actual parameters of the call. Parts (a.) and (b.) promise that if the postcondition A_{out} holds before popping the frame, then it its adapted version also holds after popping frame ($A_{out} \mathbin{-\!\triangledown} \overline{v_5}$). As in earlier lemmas, there is an important difference between (a.) and (b.) In (a.), we require *shallow satisfaction at the called*, and obtain deep satisfaction of the *adapted* version ($A_{out} \mathbin{-\!\triangledown} \overline{v_5}$) at the return point; while in (b.), we require *deep satisfaction at the called*, and obtain deep satisfaction of the *conjuction* of the *unadapted* with the *adapted* version ($A_{out} \;\wedge\; A_{out} \mathbin{-\!\triangledown} \overline{v_5}$), at the return point.

We now have a variable-free assertion, and by Lemma F.38, part a., we obtain
$M, \sigma, k \models A_{out}[\overline{\alpha_1/v_1}]$.
By Lemma 6.1 part 1, and because $\overline{\lfloor v_1 \rfloor_\sigma} = \overline{\lfloor v_3 \rfloor_{\sigma'}}$ this implies that
$M, \sigma, k \models A_{out}[\overline{v_3/v_1}]$.

b. Assume that
$M, \sigma \models A_{out}$
By Lemma 6.1 part 1 this implies that
$M, \sigma \models A_{out}[\overline{\alpha_1/v_1}]$
We now have a variable-free assertion, and by Lemma F.38, part b., we obtain
$M, \sigma', k \models A_{out}[\overline{\alpha_1/v_1}] -\!\!\nabla \overline{\alpha}_5$
By Lemma 6.1 part 1, and because $\overline{\lfloor v_1 \rfloor_\sigma} = \overline{\lfloor v_3 \rfloor_{\sigma'}}$ and $\alpha_5 = \overline{\lfloor v_5 \rfloor_{\sigma'}}$, we obtain
$M, \sigma', k \models A_{out}[\overline{v_3/v_1}] -\!\!\nabla \overline{v}_5$

□

Lemma F.42 (From any called to caller)**.** For any assertion A_{out}, states σ, σ', variables res, u variable sequence $\overline{v_5}$, and statement $stmt$.
If

(i) $Stb^+(A_{out})$,
(ii) $Fv(A_{out}) = \emptyset$,
(iii) $\overline{\lfloor v_5 \rfloor_{\sigma'}}, \lfloor res \rfloor_\sigma \subseteq LocRchbl(\sigma)$.
(iv) $\sigma' = (\sigma \vartriangle)[u \mapsto \lfloor res \rfloor_\sigma][\texttt{cont} \mapsto stmt]$.

then

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

Proof.

a. Assume that
$M, \sigma \models A_{out}$
Since A_{out} is a variable-free assertion, by Lemma F.38, part a., we obtain
$M, \sigma', k \models A_{out} -\!\!\nabla (\overline{\lfloor v_5 \rfloor_{\sigma'}})$.
By Lemma 6.1 part 1, we obtain
$M, \sigma', k \models A_{out} -\!\!\nabla (\overline{v_5})$
b. Similar argument to the proof of Lemma F.41, part (b).

□

Discussion of lemma F.42. , Similarly to lemma F.41, in this lemma, state σ stands for the last state in the method body, and σ' for the first state after exiting the method call. A_{out} stands for a method postcondition. The lemma is meant to apply to external calls, and therefore, we do not know the method's formal parameters, A_{out} is meant to stand for a module invariant where all the free variables have been substituted by addresses – *i.e.* A_{out} has no free variables. The variables $\overline{v_3}$ stand for the actual parameters of the call. Parts (a.) and (b.) promise that if the postcondition A_{out} holds before popping the frame, then it its adapted version also holds after popping frame ($A_{out} -\!\!\nabla \overline{v_5}$). As in earlier lemmas, there is an important difference between (a.) and (b.) In (a.), we require *shallow satisfaction at the called*, and obtain deep satisfaction of the *adapted* version ($A_{out} -\!\!\nabla \overline{v_5}$) at the return point; while in (b.), we require *deep satisfaction at the called*, and obtain deep satisfaction of the *conjuction* of the *unadapted* with the *adapted* version ($A_{out} \;\wedge\; A_{out} -\!\!\nabla \overline{v_5}$), at the return point.

G.15 Use of Lemmas G.39-G.40

As we said earlier, Lemmas G.39-G.40 are used to prove the soundness of the Hoare logic rules for method calls.

In the proof of soundness of Call_Int. we will use Lemma G.39 part (a.) and Lemma G.41 part (a.). In the proof of soundness of Call_Int_Adapt we will use Lemma G.39 part (b.) and Lemma G.41 part (b.). In the proof of soundness of Call_Ext_Adapt we will use Lemma G.40 part (a.) and Lemma G.42 part (a.). And finally, in the proof of soundness of Call_Ext_Adapt_Strong we will use Lemma G.40 part (b.) and Lemma G.42 part (b.).

G.16 Proof of Theorem 9.3 – part (A)

Begin Proof

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

$$(1)\quad \vdash M.$$

We will prove that

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

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

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

$$(2)\quad M \vdash \{\, A\,\}\ \sigma.\texttt{cont}\{\, A'\,\} \ \parallel\ \{\, A''\,\}$$
$$(3)\quad \overline{w} = Fv(A) \cap dom(\sigma), \quad \overline{z} = Fv(A) \setminus dom(\sigma)^{16}$$
$$(4)\quad M, \sigma, k \models A[\overline{\alpha/z}]$$

To show

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

We proceed by case analysis on the rule applied in the last step of the proof of (2). We only describe some cases.

MID By Theorem 9.2.

SEQU Therefore, there exist statements $stmt_1$ and $stmt_2$, and assertions A_1, A_2 and A'', so that $A_1 \stackrel{\text{txt}}{=} A$, and $A_2 \stackrel{\text{txt}}{=} A'$, and $\sigma.\texttt{cont} \stackrel{\text{txt}}{=} stmt_1; stmt_2$,. We apply lemma G.25, and obtain that there exists an intermediate state σ_1. The proofs for $stmt_1$ and $stmt_2$, and the intermediate state σ_1 are in the $\ll$ relation. Therefore, we can apply the inductive hypothesis.

COMBINE by induction hypothesis, and unfolding and folding the definitions

CONSEQU using Lemma G.3 part 4 and axiom G.1

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

$$(5)\quad \sigma.\texttt{cont} \stackrel{\text{txt}}{=} u := y_0.m(\overline{y}),$$
$$(6)\quad \vdash\ M\ :\ \{\, A_{pre}\,\}\, D::m(\overline{x:D})\,\{\, A_{post}\,\} \ \parallel\ \{A_{mid}\},$$
$$(7)\quad A \stackrel{\text{txt}}{=} y_0 : D, \overline{y:D} \ \wedge\ A_{pre}[y_0, \overline{y}/\texttt{this}, \overline{x}],$$
$$A' \stackrel{\text{txt}}{=} A_{post}[y_0, \overline{y}, u/\texttt{this}, \overline{x}, \texttt{res}],$$
$$A'' \stackrel{\text{txt}}{=} A_{mid}.$$

Also,

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

where

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

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

F.15 Use of Lemmas F.39-F.40

As we said earlier, Lemmas F.39-F.40 are used to prove the soundness of the Hoare logic rules for method calls.

In the proof of soundness of CALL_INT. we will use Lemma F.39 part (a.) and Lemma F.41 part (a.). In the proof of soundness of CALL_INT_ADAPT we will use Lemma F.39 part (b.) and Lemma F.41 part (b.). In the proof of soundness of CALL_EXT_ADAPT we will use Lemma F.40 part (a.) and Lemma F.42 part (a.). And finally, in the proof of soundness of CALL_EXT_ADAPT_STRONG we will use Lemma F.40 part (b.) and Lemma F.42 part (b.).

F.16 Proof of Theorem 9.3 – part (A)

Begin Proof

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

$$(1) \quad \vdash M.$$

We will prove that

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

by induction on the well-founded ordering $\_ \ll_{M,\overline{M}} \_$.
Take σ, A, A', A'', $\overline{z}$, $\overline{w}$, $\overline{\alpha}$, σ', σ'' arbitrary. Assume that

$$(2) \quad M \vdash \{\, A\,\}\, \sigma.\texttt{cont}\{\, A'\,\} \;\parallel\; \{\, A''\,\}$$
$$(3) \quad \overline{w} = Fv(A) \cap dom(\sigma), \qquad \overline{z} = Fv(A) \setminus dom(\sigma)^{16}$$
$$(4) \quad M, \sigma, k \models A[\overline{\alpha/z}]$$

To show

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

We proceed by case analysis on the rule applied in the last step of the proof of (2). We only describe some cases.

MID By Theorem 9.2.

SEQU Therefore, there exist statements $stmt_1$ and $stmt_2$, and assertions A_1, A_2 and A'', so that $A_1 \stackrel{\text{txt}}{=} A$, and $A_2 \stackrel{\text{txt}}{=} A'$, and $\sigma.\texttt{cont} \stackrel{\text{txt}}{=} stmt_1; stmt_2$,. We apply lemma F.25, and obtain that there exists an intermediate state σ_1. The proofs for $stmt_1$ and $stmt_2$, and the intermediate state σ_1 are in the $\ll$ relation. Therefore, we can apply the inductive hypothesis.

COMBINE by induction hypothesis, and unfolding and folding the definitions

CONSEQU using Lemma F.3 part 4 and axiom F.1

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

$$(5) \quad \sigma.\texttt{cont} \stackrel{\text{txt}}{=} u := y_0.m(\overline{y}),$$
$$(6) \quad \vdash\ M\ :\ \{\, A_{pre}\,\}\, D::m(\overline{x : D})\, \{\, A_{post}\,\} \;\parallel\; \{A_{mid}\},$$
$$(7) \quad A \stackrel{\text{txt}}{=} y_0 : D, \overline{y : D} \;\wedge\; A_{pre}[y_0, \overline{y}/\texttt{this}, \overline{x}],$$
$$A' \stackrel{\text{txt}}{=} A_{post}[y_0, \overline{y}, u/\texttt{this}, \overline{x}, \texttt{res}],$$
$$A'' \stackrel{\text{txt}}{=} A_{mid}.$$

Also,

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

where

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

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

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

We define the shorthands:

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

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

$(9b)$ $A_{poa} \triangleq A_{post}\ \wedge\ A_{post}\text{-}\nabla\texttt{res}$.

By (1), (6), (7), (9), and definition of $\vdash M$ in Section 8.3 rule METHOD and we obtain

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

From (8) we obtain

(11) $(A_{pra}, \sigma_1, A_{poa}, A_{mid}) \ll_{M,\overline{M}} (A, \sigma, A', A'')$

In order to be able to apply the induction hypothesis, we need to prove something of the form ...$\sigma_1 \models A_{pr}[../fv(A_{pr}) \setminus dom(\sigma_1)]$. To that aim we will apply Lemma G.39 part a. on (4), (8a) and (9). For this, we take

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

These definitions give that

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

$(12b)$ $Fv(A_{pr})\ = \overline{v_1}; \overline{v_2}$.

$(12c)$ $Fv(A)\ = \overline{v_3}; \overline{v_4}$.

With (12a), (12b), (12c), (and Lemma G.30 part (7), we obtain that

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

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

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

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

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

Moreover, we have that $\overline{z}\#\overline{v_3}$. From Lemma G.30 part (10) we obtain $\overline{z}\#\overline{v_1}$. And, because $\overline{\alpha}$ are addresses wile $\overline{v_1}$ are variables, we also have that $\overline{\alpha}\#\overline{v_1}$. These facts, together with Lemma G.30 part (9) give that

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

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

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

From (4), (8a), (12a)-(12e) we see that the requirements of Lemma G.39 part a. are satisfied where we take A_{in} to be $A_{pr}[\overline{\alpha/z}]$. We use the definition of y_r in (12d), and define

$(13c)$ $\overline{v_6} \triangleq y_r; (\overline{v_4} \setminus \overline{z})\quad$ which, with (12d) also gives: $\overline{v_2} = \overline{v_6}; \overline{z}$

We apply Lemma G.39 part a. on (13b), (13c) and obtain

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

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

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

With similar re-orderings to earlier, we obain

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

For the proof of $(**)$ as well as for the proof of $(***)$, we will want to apply the inductive hypothesis. For this, we need to determine the value of $Fv(A_{pr}) \setminus dom(\sigma_1)$, as well as the value of $Fv(A_{pr}) \cap dom(\sigma_1)$. This is what we do next. From (8a) we have that

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

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

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

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

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

We define the shorthands:

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

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

$(9b)\ \ A_{poa} \triangleq A_{post}\ \wedge\ A_{post} \neg\nabla \texttt{res}$.

By (1), (6), (7), (9), and definition of $\vdash M$ in Section 8.3 rule Method and we obtain

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

From (8) we obtain

$(11)\ \ (A_{pra}, \sigma_1, A_{poa}, A_{mid}) \ll_{M,\overline{M}} (A, \sigma, A', A'')$

In order to be able to apply the induction hypothesis, we need to prove something of the form $...\sigma_1 \models A_{pr}[../fv(A_{pr}) \setminus dom(\sigma_1)]$. To that aim we will apply Lemma F.39 part a. on (4), (8a) and (9). For this, we take

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

These definitions give that

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

$(12b)\ \ Fv(A_{pr})\ = \overline{v_1}; \overline{v_2}$.

$(12c)\ \ Fv(A)\ = \overline{v_3}; \overline{v_4}$.

With (12a), (12b), (12c), (and Lemma F.30 part (7), we obtain that

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

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

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

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

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

Moreover, we have that $\overline{z} \# \overline{v_3}$. From Lemma F.30 part (10) we obtain $\overline{z} \# \overline{v_1}$. And, because $\overline{\alpha}$ are addresses wile $\overline{v_1}$ are variables, we also have that $\overline{\alpha} \# \overline{v_1}$. These facts, together with Lemma F.30 part (9) give that

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

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

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

From (4), (8a), (12a)-(12e) we see that the requirements of Lemma F.39 part a. are satisfied where we take A_{in} to be $A_{pr}[\overline{\alpha/z}]$. We use the definition of y_r in (12d), and define

$(13c)\ \ \overline{v_6} \triangleq y_r; (\overline{v_4} \setminus \overline{z}) \quad$ which, with (12d) also gives: $\overline{v_2} = \overline{v_6}; \overline{z}$

We apply Lemma F.39 part a. on (13b), (13c) and obtain

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

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

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

With similar re-orderings to earlier, we obain

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

For the proof of $(**)$ as well as for the proof of $(***)$, we will want to apply the inductive hypothesis. For this, we need to determine the value of $Fv(A_{pr}) \setminus dom(\sigma_1)$, as well as the value of $Fv(A_{pr}) \cap dom(\sigma_1)$. This is what we do next. From (8a) we have that

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

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

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

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

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

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

Proving $(**)$. Assume that $\overline{M} \cdot M;\ \sigma \rightsquigarrow^*_{fin} \sigma'$. Then, by the operational semantics, we obtain that there exists state σ_1', such that

$(16)\ \ \overline{M} \cdot M;\ \sigma_1 \rightsquigarrow^*_{fin} \sigma_1'$

$(17)\ \ \sigma' = (\sigma_1' \vartriangle)[u \mapsto \lfloor \texttt{res} \rfloor_{\sigma_1'}][\texttt{cont} \mapsto \epsilon]$.

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

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

We now want to obtain something of the form $...\sigma' \models ...A'$. We now want to be able to apply Lemma G.41, part a. on (18). Therefore, we define

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

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

The wellformedness condition for specifications requires that $Fv(A_{post}) \subseteq Fv(A_{pr}) \cup \{\texttt{res}\}$. This, together with (9), (12d) and (18b) give

$(19a)\ \ Fv(A_{out}) \subseteq \overline{v_{1,a}}$

Also, by (18b), and (17), we have that

$(19b)\ \ \overline{\lfloor v_{3,a} \rfloor_{\sigma'} = \lfloor v_{1,a} \rfloor_{\sigma_1'}}$.

From (4) we obtain that $k \leq |\sigma|$. From (8a) we obtain that $|\sigma_1| = |\sigma| + 1$. From (16) we obtain that $|\sigma_1'| = |\sigma_1|$, and from (17) we obtain that $|\sigma'| = |\sigma_1'| - 1$. All this gives that:

$(19c)\ \ k \leq |\sigma'|$

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

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

We expand the definition from (18a), and re-order the substitutions by a similar argument as in in step (13a), using Lemma part (9), and obtain

$(20a)\ \ M, \sigma', k \models A_{poa}[\overline{v_{3,a}/v_{1,a}}][\overline{\alpha/z}][\overline{\lfloor v_6 \rfloor_\sigma / v_6}]$.

By (20a), (18b), and because by Lemma B.2 we have that $\overline{\lfloor v_6 \rfloor_\sigma} = \overline{\lfloor v_6 \rfloor_{\sigma'}}$, we obtain

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

With (7) we conclude.

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

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

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

Then, from (8) and (15) we also obtain that

$(15)\ \ \overline{M} \cdot M;\ \sigma_1 \rightsquigarrow^* \sigma''$

By (10), (11) and application of the induction hypothesis on (13), (14c), and (15), we obtain that

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

and using (7) we are done.

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

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

Moreover, (12d) and (13d) give that
$(15d)\ \ Fv(A_{pra}) \setminus dom(\sigma_1) = \overline{z_2} = \overline{z}; \overline{v_6}$.

Proving $(**)$. Assume that $\overline{M} \cdot M;\ \sigma \leadsto^*_{fin} \sigma'$. Then, by the operational semantics, we obtain that there exists state σ_1', such that
$(16)\ \ \overline{M} \cdot M;\ \sigma_1 \leadsto^*_{fin} \sigma_1'$
$(17)\ \ \sigma' = (\sigma_1' \vartriangle)[u \mapsto \lfloor \texttt{res} \rfloor_{\sigma_1'}][\texttt{cont} \mapsto \epsilon]$.
We now apply the induction hypothesis on (14), (16), (15d), and obtain
$(18)\ \ M, \sigma_1', k \models (A_{post})[\overline{\alpha/z}][\overline{\lfloor v_6 \rfloor_\sigma / v_6}]$.
We now want to obtain something of the form $...\sigma' \models ...A'$. We now want to be able to apply Lemma F.41, part a. on (18). Therefore, we define
$(18a)\ \ A_{out} \triangleq A_{poa}[\overline{\alpha/z}][\overline{\lfloor v_6 \rfloor_\sigma / v_6}]$
$(18b)\ \ \overline{v_{1,a}} \triangleq \overline{v_1}, \texttt{res}, \quad \overline{v_{3,a}} \triangleq \overline{v_3}, u$.
The wellformedness condition for specifications requires that $Fv(A_{post}) \subseteq Fv(A_{pr}) \cup \{\texttt{res}\}$. This, together with (9), (12d) and (18b) give
$(19a)\ \ Fv(A_{out}) \subseteq \overline{v_{1,a}}$
Also, by (18b), and (17), we have that
$(19b)\ \ \overline{\lfloor v_{3,a} \rfloor_{\sigma'} = \lfloor v_{1,a} \rfloor_{\sigma_1'}}$.
From (4) we obtain that $k \le |\sigma|$. From (8a) we obtain that $|\sigma_1| = |\sigma| + 1$. From (16) we obtain that $|\sigma_1'| = |\sigma_1|$, and from (17) we obtain that $|\sigma'| = |\sigma_1'| - 1$. All this gives that:
$(19c)\ \ k \le |\sigma'|$
We now apply Lemma F.41, part a., and obtain
$(20)\ \ M, \sigma', k \models A_{out}[\overline{v_{3,a}/v_{1,a}}]$.
We expand the definition from (18a), and re-order the substitutions by a similar argument as in in step (13a), using Lemma part (9), and obtain
$(20a)\ \ M, \sigma', k \models A_{poa}[\overline{v_{3,a}/v_{1,a}}][\overline{\alpha/z}][\overline{\lfloor v_6 \rfloor_\sigma / v_6}]$.
By (20a), (18b), and because by Lemma B.2 we have that $\overline{\lfloor v_6 \rfloor_\sigma} = \overline{\lfloor v_6 \rfloor_{\sigma'}}$, we obtain
$(21)\ \ M, \sigma', k \models (A_{poa})[y_0, \overline{y}, u/\texttt{this}, \overline{x}, \texttt{res}][\overline{\alpha/z}]$..
With (7) we conclude.

Proving $(***)$. Take a σ''. Assume that
$(15)\ \ \overline{M} \cdot M;\ \sigma \leadsto^* \sigma''$
$(16)\ \ \overline{M} \cdot M, \sigma'' \models \texttt{extl}$.
Then, from (8) and (15) we also obtain that
$(15)\ \ \overline{M} \cdot M;\ \sigma_1 \leadsto^* \sigma''$
By (10), (11) and application of the induction hypothesis on (13), (14c), and (15), we obtain that
$(\beta')\ \ M, \sigma'', k \models A_{mid}[\overline{\alpha/z}][\overline{\lfloor w \rfloor_\sigma / w}]$.
and using (7) we are done.

Call_Int_Adapt is similar to Call_Int. We highlight the differences in green . Therefore, there exist u, y_o, C, $\overline{y}$, A_{pre}, A_{post}, and A_{mid}, such that
$(5)\ \ \sigma.\texttt{cont} \stackrel{\text{txt}}{=} u := y_0.m(\overline{y})$,

(6) $\vdash\ M\ :\ \{\, A_{pre}\,\}\, D\!::\!m(\overline{x:D})\,\{\, A_{post}\,\}\ \parallel\ \{A_{mid}\}$,
(7) $A \stackrel{\text{txt}}{=} y_0 : D, \overline{y:D}\ \wedge\ (A_{pre}[y_0/\texttt{this}]) \neg\!\nabla(y_0, \overline{y})$,
$A' \stackrel{\text{txt}}{=} (A_{post}[y_0/\texttt{this}, u/res]) \neg\!\nabla(y_0, \overline{y})$,
$A'' \stackrel{\text{txt}}{=} A_{mid}$.

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

where
$(8a)$ $\sigma_1 \triangleq (\sigma \triangledown (\texttt{this} \mapsto \lfloor y_0 \rfloor_\sigma, \overline{x \mapsto \lfloor y \rfloor_\sigma})[\texttt{cont} \mapsto stmt_m]$,
$(8b)$ $\texttt{mBody}(m, D, M) = \overline{y:D}\{\ stmt_m\ \}$.

We define the shorthand:
(9) $A_{pr} \triangleq \texttt{this} : D, \overline{x:D}\ \wedge\ A_{pre}$.
$(9a)$ $A_{pra} \triangleq \texttt{this} : D, \overline{x:D}\ \wedge\ A_{pre}\ \wedge\ A_{pre} \neg\!\nabla(y_0, \overline{y})$.
$(9b)$ $A_{poa} \triangleq A_{post}\ \wedge\ A_{post} \neg\!\nabla \texttt{res}$.

By (1), (6), (7), (9), and definition of $\vdash M$ in Section 8.3, we obtain
(10) $M \vdash \{\ A_{pra}\ \}\ stmt_m \{\ A_{poa}\ \}\ \parallel\ \{\, A_{mid}\, \}$.

From (8) we obtain
(11) $(A_{pra}, \sigma_1, A_{poa}, A_{mid}) \ll_{M,\overline{M}} (A, \sigma, A', A'')$

In order to be able to apply the induction hypothesis, we need to prove something of the form $...\sigma_1 \models A_{pr}[../fv(A_{pr}) \setminus dom(\sigma_1)]$. To that aim we will apply Lemma G.39 part b. on (4), (8a) and (9). For this, we take
(12) $\overline{v_1} \triangleq \texttt{this}, \overline{x}$, $\quad \overline{v_2} \triangleq Fv(A_{pr}) \setminus \overline{v_1}$, $\quad \overline{v_3} \triangleq y_0, \overline{y}$, $\quad \overline{v_4} \triangleq Fv(A) \setminus \overline{v_3}$

These definitions give that
$(12a)$ $A \stackrel{\text{txt}}{=} (A_{pr}[\overline{v_3/v_1}]) \neg\!\nabla(\overline{v_3})$,
$(12b)$ $Fv(A_{pr})\ = \overline{v_1}; \overline{v_2}$.
$(12c)$ $Fv(A)\ = \overline{v_3}; \overline{v_4}$.

With (12a), (12b), (12c),and Lemma G.30 part (7), we obtain that
$(12d)$ $\overline{v_2} = \overline{y_r}; \overline{v_4}$, $\quad$ where $\overline{y_r} \triangleq \overline{v_2} \cap \overline{v_3}$

Furthermore, (8a), and (12) give that:
$(12e)$ $\overline{\lfloor v_1 \rfloor_{\sigma_1} = \lfloor v_3 \rfloor_\sigma}$

Then, (4), (12a) give that
(13) $M, \sigma, k \models A_{pr}[\overline{v_3/v_1}] \neg\!\nabla(\overline{v_3})$

Moreover, we have that $\overline{z} \# \overline{v_3}$. From Lemma G.30 part (10) we obtain $\overline{z} \# \overline{v_1}$. And, because $\overline{\alpha}$ are addresses wile $\overline{v_1}$ are variables, we also have that $\overline{\alpha} \# \overline{v_1}$. These facts, together with Lemma G.30 part (9) give that
$(13a)$ $A_{pr}[\overline{v_3/v_1}][\overline{\alpha/z}] \stackrel{\text{txt}}{=} A_{pr}[\overline{\alpha/z}][\overline{v_3/v_1}]$

From (13a) and (13), we obtain
$(13b)$ $M, \sigma, k \models (A_{pr}[\overline{\alpha/z}][\overline{v_3/v_1}]) \neg\!\nabla(\overline{v_3})$

From (4), (8a), (12a)-(12e) we see that the requirements of Lemma G.39 where we take A_{in} to be $A_{pr}[\overline{\alpha/z}]$. We use the definition of y_r in (12d), and define
$(13c)$ $\overline{v_6} \triangleq y_r; (\overline{v_4} \setminus \overline{z})$, $\quad$ this also gives that $\overline{v_2} = \overline{v_6}; \overline{z}$

We apply Lemma G.39 part b. on (13b), (13c) and obtain
(14) $M, \sigma_1 \models A_{pr}[\overline{\alpha/z}][\overline{\lfloor v_6 \rfloor_\sigma / v_6}]$.

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

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

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

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

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

Also,

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

where

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

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

We define the shorthand:

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

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

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

By (1), (6), (7), (9), and definition of $\vdash M$ in Section 8.3, we obtain

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

From (8) we obtain

$(11)\ \ (A_{pra}, \sigma_1, A_{poa}, A_{mid}) \ll_{M,\overline{M}} (A, \sigma, A', A'')$

In order to be able to apply the induction hypothesis, we need to prove something of the form $...\sigma_1 \models A_{pr}[../fv(A_{pr}) \setminus dom(\sigma_1)]$. To that aim we will apply Lemma F.39 part b. on (4), (8a) and (9). For this, we take

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

These definitions give that

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

$(12b)\ \ Fv(A_{pr})\ = \overline{v_1}; \overline{v_2}.$

$(12c)\ \ Fv(A)\ = \overline{v_3}; \overline{v_4}.$

With (12a), (12b), (12c),and Lemma F.30 part (7), we obtain that

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

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

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

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

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

Moreover, we have that $\overline{z}\#\overline{v_3}$. From Lemma F.30 part (10) we obtain $\overline{z}\#\overline{v_1}$. And, because $\overline{\alpha}$ are addresses wile $\overline{v_1}$ are variables, we also have that $\overline{\alpha}\#\overline{v_1}$. These facts, together with Lemma F.30 part (9) give that

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

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

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

From (4), (8a), (12a)-(12e) we see that the requirements of Lemma F.39 where we take A_{in} to be $A_{pr}[\overline{\alpha/z}]$. We use the definition of y_r in (12d), and define

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

We apply Lemma F.39 part b. on (13b), (13c) and obtain

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

which is equivalent to

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

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

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

For the proof of ($**$) as well as for the proof of ($***$), we will want to apply the inductive hypothesis. For this, we need to determine the value of $Fv(A_{pr}) \setminus dom(\sigma_1)$, as well as the value of $Fv(A_{pr}) \cap dom(\sigma_1)$. This is what we do next. From (8a) we have that

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

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

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

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

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

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

Proving ($**$). Assume that $\overline{M} \cdot M;\ \sigma \rightsquigarrow^*_{fin} \sigma'$. Then, by the operational semantics, we obtain that there exists state σ_1', such that

(16) $\overline{M} \cdot M;\ \sigma_1 \rightsquigarrow^*_{fin} \sigma_1'$

(17) $\sigma' = (\sigma_1' \vartriangle)[u \mapsto \lfloor \texttt{res} \rfloor_{\sigma_1'}][\texttt{cont} \mapsto \epsilon]$.

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

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

We now want to obtain something of the form $...\sigma' \models ...A'$. For this, we want to be able to apply Lemma G.41, part b. on (18). Therefore, we define

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

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

The wellformedness condition for specifications requires that $Fv(A_{post}) \subseteq Fv(A_{pr}) \cup \{\texttt{res}\}$. This, together with (9), (12d) and (18b) give

(19a) $Fv(A_{out}) \subseteq \overline{v_{1,a}}$

Also, by (18b), and (17), we have that

(19b) $\overline{\lfloor v_{3,a}\rfloor_{\sigma'} = \lfloor v_{1,a}\rfloor_{\sigma_1'}}$.

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

(19c) $M, \sigma_1' \models (A_{poa})[\overline{\alpha/z}][\overline{\lfloor v_6\rfloor_\sigma / v_6}]$.

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

(19d) $\overline{\lfloor v_5\rfloor_{\sigma'}} \subseteq LocRchbl(\sigma_1')$

We now apply Lemma G.41, part b., and obtain

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

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

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

We expand the definition from (18a), and re-order the substitutions by a similar argument as in in step (13a), using Lemma part (9), and obtain

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

By Lemma B.2 we have that $\overline{\lfloor v_6\rfloor_\sigma} = \overline{\lfloor v_6\rfloor_{\sigma'}}$, and so we obtain

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

Moreover, $\overline{v_3} \# \overline{z}$. We apply lemma G.31, and obtain:

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

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

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

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

For the proof of (∗∗) as well as for the proof of (∗∗∗), we will want to apply the inductive hypothesis. For this, we need to determine the value of $Fv(A_{pr}) \setminus dom(\sigma_1)$, as well as the value of $Fv(A_{pr}) \cap dom(\sigma_1)$. This is what we do next. From (8a) we have that

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

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

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

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

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

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

Proving (∗∗). Assume that $\overline{M} \cdot M;\ \sigma \rightsquigarrow^*_{fin} \sigma'$. Then, by the operational semantics, we obtain that there exists state σ'_1, such that

(16) $\overline{M} \cdot M;\ \sigma_1 \rightsquigarrow^*_{fin} \sigma'_1$

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

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

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

We now want to obtain something of the form $...\sigma' \models ...A'$. For this, we want to be able to apply Lemma F.41, part b. on (18). Therefore, we define

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

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

The wellformedness condition for specifications requires that $Fv(A_{post}) \subseteq Fv(A_{pr}) \cup \{\texttt{res}\}$. This, together with (9), (12d) and (18b) give

(19a) $Fv(A_{out}) \subseteq \overline{v_{1,a}}$

Also, by (18b), and (17), we have that

(19b) $\overline{\lfloor v_{3,a} \rfloor_{\sigma'} = \lfloor v_{1,a} \rfloor_{\sigma'_1}}$.

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

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

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

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

We now apply Lemma F.41, part b., and obtain

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

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

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

We expand the definition from (18a), and re-order the substitutions by a similar argument as in in step (13a), using Lemma part (9), and obtain

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

By Lemma B.2 we have that $\overline{\lfloor v_6 \rfloor_\sigma} = \overline{\lfloor v_6 \rfloor_{\sigma'}}$, and so we obtain

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

Moreover, $\overline{v_3} \# \overline{z}$. We apply lemma F.31, and obtain:

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

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

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

With (7) we conclude.

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

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

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

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

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

(7) $A \stackrel{\text{txt}}{=} y_0 : \texttt{external}, \overline{x:C}\ \wedge\ A_{inv} \neg\nabla (y_0, \overline{y})$,

$A' \stackrel{\text{txt}}{=} A_{inv} \neg\nabla (y_0, \overline{y})$,

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

Also,

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

where

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

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

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

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

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

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

By Barendregt, we also obtain that

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

This, together with (3) gives that

(10) $\overline{z} = \overline{x}$

From (4), (7) and the definition of satisfaction we obtain

(10) $M, \sigma, k \models (\overline{x:C} \wedge A_{inv} \triangledown y_0, \overline{y})[\overline{\alpha/z}]$.

The above gives that

(10a) $M, \sigma, k \models (\ (\overline{x:C})[\overline{\alpha/z}]\ \wedge\ (A_{inv}[\overline{\alpha/z}])\ ) \triangledown y_0, \overline{y}$.

We take A_{in} to be $(\overline{x:C})[\overline{\alpha/z}]\ \wedge\ (A_{inv}[\overline{\alpha/z}])$, and apply Lemma G.40, part a.. This gives that

(11) $M, \sigma_a \models (\overline{x:C})[\overline{\alpha/z}]\ \wedge\ A_{inv}[\overline{\alpha/z}]$

Proving ($**$). We shall use the short hand

(12) $A_o \triangleq \overline{\alpha:C} \wedge A_{inv}[\overline{\alpha/z}]$.

Assume that $\overline{M} \cdot M;\ \sigma \rightsquigarrow^*_{fin} \sigma'$. Then, by the operational semantics, we obtain that there exists state σ'_b, such that

(16) $\overline{M} \cdot M;\ \sigma_a \rightsquigarrow^*_{fin} \sigma_b$

(17) $\sigma' = (\sigma_b \vartriangle)[u \mapsto \lfloor \texttt{res} \rfloor_{\sigma'_1}][\texttt{cont} \mapsto \epsilon]$.

By Lemma G.27 part 1, and Def. G.26, we obtain that there exists a sequence of states σ_1, ... σ_n, such that

(17) $(\overline{M} \cdot M,\ \sigma_a); \sigma_a \rightsquigarrow^*_{e,p} \sigma_b$ **pb** $\sigma_1 \ldots \sigma_n$

(21) $M, \sigma', k \models (A_{post}[y_0, \overline{y}, u/\texttt{this}, \overline{x}, \texttt{res}] \neg\!\triangledown (y_0, \overline{y}))[\overline{\alpha/z}]$.

With (7) we conclude.

Proving (∗∗∗). This is similar to the proof for Call_Int.

Call_Ext_Adapt is in some parts, similar to Call_Int, and Call_Int_Adaprt. We highlight the differences in green .

Therefore, there exist u, y_o, $\overline{C}$, D, $\overline{y}$, ands A_{inv}, such that

(5) $\sigma.\texttt{cont} \stackrel{\text{txt}}{=} u := y_0.m(\overline{y})$,

(6) $\vdash\ M\ :\ \mathbb{\forall}\overline{x : C}.\{A_{inv}\}$,

(7) $A \stackrel{\text{txt}}{=} y_0 : \texttt{external}, \overline{x : C}\ \wedge\ A_{inv} \neg\!\triangledown (y_0, \overline{y})$,

$A' \stackrel{\text{txt}}{=} A_{inv} \neg\!\triangledown (y_0, \overline{y})$,

$A'' \stackrel{\text{txt}}{=} A_{inv}$.

Also,

(8) $\overline{M} \cdot M;\ \sigma \rightsquigarrow \sigma_a$,

where

(8a) $\sigma_a \triangleq (\sigma \triangledown (\texttt{this} \mapsto \lfloor y_0 \rfloor_\sigma, \overline{p \mapsto \lfloor y \rfloor_\sigma})[\texttt{cont} \mapsto stmt_m]$,

(8b) $\texttt{mBody}(m, D, \overline{M}) = \overline{p : D}\{\ stmt_m\ \}$.

(8c) D is the class of $\lfloor y_0 \rfloor_\sigma$, and D is external.

By (7), and well-formedness of module invariants, we obtain

(9a) $Fv(A_{inv}) \subseteq \overline{x}$,

(9a) $Fv(A) = y_0, \overline{y}, \overline{x}$

By Barendregt, we also obtain that

(10) $dom(\sigma)\ \#\overline{x}$

This, together with (3) gives that

(10) $\overline{z} = \overline{x}$

From (4), (7) and the definition of satisfaction we obtain

(10) $M, \sigma, k \models (\overline{x : C} \wedge A_{inv} \triangledown y_0, \overline{y})[\overline{\alpha/z}]$.

The above gives that

(10a) $M, \sigma, k \models (\ (\overline{x : C})[\overline{\alpha/z}]\ \wedge\ (A_{inv}[\overline{\alpha/z}])\ ) \triangledown y_0, \overline{y}$.

We take A_{in} to be $(\overline{x : C})[\overline{\alpha/z}]\ \wedge\ (A_{inv}[\overline{\alpha/z}])$, and apply Lemma F.40, part a.. This gives that

(11) $M, \sigma_a \models (\overline{x : C})[\overline{\alpha/z}]\ \wedge\ A_{inv}[\overline{\alpha/z}]$

Proving (∗∗). We shall use the short hand

(12) $A_o \triangleq \overline{\alpha : C} \wedge A_{inv}[\overline{\alpha/z}]$.

Assume that $\overline{M} \cdot M;\ \sigma {\rightsquigarrow}^*_{fin} \sigma'$. Then, by the operational semantics, we obtain that there exists state σ'_b, such that

(16) $\overline{M} \cdot M;\ \sigma_a {\rightsquigarrow}^*_{fin} \sigma_b$

(17) $\sigma' = (\sigma_b \vartriangle)[u \mapsto \lfloor \texttt{res} \rfloor_{\sigma'_1}][\texttt{cont} \mapsto \epsilon]$.

By Lemma F.27 part 1, and Def. F.26, we obtain that there exists a sequence of states σ_1, ... σ_n, such that

(17) $(\overline{M} \cdot M,\ \sigma_a); \sigma_a {\rightsquigarrow}^*_{e,p}\ \sigma_b\ \mathbf{pb}\ \sigma_1 ... \sigma_n$

By Def. G.26, the states σ_1, ... σ_n are all public, and correspond to the execution of a public method. Therefore, by rule INVARIANT for well-formed modules, we obtain that

(18) $\forall i \in 1..n.$

$[\ M\ \vdash\ \{\texttt{this}: D_i, \overline{p_i : D_i}, \overline{x : C}\ \wedge\ A_{inv}\ \}\sigma_i.\texttt{cont}\{\ A_{inv,r}\ \}\ \ \|\ \ \{\ A_{inv}\ \}\ ]$

where D_i is the class of the receiver, $\overline{p_i}$ are the formal parameters, and $\overline{D_i}$ are the types of the formal parameters of the i-th public method, and where we use the shorthand $A_{inv,r} \triangleq A_{inv} \neg\!\nabla \texttt{res}$.

Moreover, (17) gives that

(19) $\forall i \in 1..n.[\ \ M \cdot \overline{M};\ \sigma \rightsquigarrow^* \sigma_i\ \ ]$

From (18) and (19) we obtain

(20) $\forall i \in [1..n].$

$[\ \ (\texttt{this}: D_i, \overline{p_i : D_i}, \overline{x : C}\ \wedge\ A_{inv}, \sigma_i, A_{inv,r}, A_{inv})$

$\ll_{M,\overline{M}}$

$(A, \sigma, A', A'')\ \ ]$

We take

(21) $k = |\sigma_a|$ By application of the induction hypothesis on (20) we obtain that

(22) $\forall i \in [1..n].\forall \sigma_f.[\ \ M, \sigma_i, k \models A_o\ \ \wedge\ \ M \cdot \overline{M};\ \sigma_i \rightsquigarrow^*_{fin} \sigma_f\ \ \ \Longrightarrow\ \ M, \sigma_f, k \models A_o\ ]$

We can now apply Lemma G.28, part 3, and because $|\sigma_a| = |\sigma_b|$, we obtain that

(23) $M, \sigma_b \models A_{inv}[\overline{\alpha/x}]$

We apply Lemma G.42 part a., and obtain

(24) $M, \sigma' \models A_{inv}[\overline{\alpha/x}] \neg\!\nabla y_0, \overline{y}$

And since $A_{inv}[\overline{\alpha/x}] \neg\!\nabla y_0, \overline{y}$ is stable, and by rearranging, and applying (10), we obtain

(25) $M, \sigma', k \models (A_{inv} \neg\!\nabla y_0, \overline{y})[\overline{\alpha/z}]$

Apply (7), and we are done.

Proving (∗∗∗). Take a σ''. Assume that

(12) $\overline{M} \cdot M;\ \sigma \rightsquigarrow^* \sigma''$

(13) $\overline{M} \cdot M, \sigma'' \models \texttt{extl}.$

We apply lemma 1, part 2 on (12) and see that there are two cases

1st Case $\overline{M} \cdot M; \sigma_a \rightsquigarrow^*_{e,p} \sigma''$

That is, the execution from σ_a to σ'' goes only through external states. We use (11), and that A_{inv} is encapsulated, and are done with lemma G.28, part 1.

2nd Case for some σ_c, σ_d. we have

$\overline{M} \cdot M; \sigma_a \rightsquigarrow^*_{e,p} \sigma_c\ \wedge\ \overline{M} \cdot M;\ \sigma_c \rightsquigarrow \sigma_d\ \wedge\ M, \sigma_d \models \texttt{pub} \wedge \overline{M} \cdot M;\ \sigma_d \rightsquigarrow^* \sigma'$

We apply similar arguments as in steps (17)-(23) and obtain

(14) $\ M, \sigma_c \models A_{inv}[\overline{\alpha/x}]$

State σ_c is a public, internal state; therefore there exists a Hoare proof that it preserves the invariant. By applying the inductive hypothesis, and the fact that $\overline{z} = \overline{x}$, we obtain:

(14) $\ M, \sigma'' \models A_{inv}[\overline{\alpha/z}]$

CALL_EXT_ADAPT_STRONG is very similar to CALL_EXT_ADAPRT. We will summarize the similar steps, and highlight the differences in green .

Therefore, there exist u, y_o, $\overline{C}$, D, $\overline{y}$, and A_{inv}, such that

(5) $\sigma.\texttt{cont} \stackrel{\text{txt}}{=} u := y_0.m(\overline{y}),$

By Def. F.26, the states σ_1, ... σ_n are all public, and correspond to the execution of a public method. Therefore, by rule Invariant for well-formed modules, we obtain that

(18) $\forall i \in 1..n.$
$$[\ M\ \vdash\ \{\mathtt{this}: D_i, \overline{p_i : D_i}, \overline{x:C}\ \wedge\ A_{inv}\ \}\sigma_i.\mathtt{cont}\{\ A_{inv,r}\ \}\ \ \|\ \ \{\ A_{inv}\ \}\ ]$$

where D_i is the class of the receiver, $\overline{p_i}$ are the formal parameters, and $\overline{D_i}$ are the types of the formal parameters of the i-th public method, and where we use the shorthand $A_{inv,r} \triangleq A_{inv} \neg\!\nabla\, \mathtt{res}$.

Moreover, (17) gives that

(19) $\forall i \in 1..n.[\ \ M \cdot \overline{M};\ \sigma \rightsquigarrow^* \sigma_i\ \ ]$

From (18) and (19) we obtain

(20) $\forall i \in [1..n].$
$$[\ \ (\mathtt{this}: D_i, \overline{p_i : D_i}, \overline{x:C}\ \wedge\ A_{inv}, \sigma_i, A_{inv,r}, A_{inv}) \ll_{M,\overline{M}} (A, \sigma, A', A'')\ \ ]$$

We take

(21) $k = |\sigma_a|$ By application of the induction hypothesis on (20) we obtain that

(22) $\forall i \in [1..n].\forall \sigma_f.[\ \ M, \sigma_i, k \models A_o\ \ \wedge\ \ M \cdot \overline{M};\ \sigma_i \rightsquigarrow^*_{fin} \sigma_f\ \ \ \Longrightarrow\ \ M, \sigma_f, k \models A_o\ ]$

We can now apply Lemma F.28, part 3, and because $|\sigma_a| = |\sigma_b|$, we obtain that

(23) $M, \sigma_b \models A_{inv}[\overline{\alpha/x}]$

We apply Lemma F.42 part a., and obtain

(24) $M, \sigma' \models A_{inv}[\overline{\alpha/x}] \neg\!\nabla\, y_0, \overline{y}$

And since $A_{inv}[\overline{\alpha/x}] \neg\!\nabla\, y_0, \overline{y}$ is stable, and by rearranging, and applying (10), we obtain

(25) $M, \sigma', k \models (A_{inv} \neg\!\nabla\, y_0, \overline{y})[\overline{\alpha/z}]$

Apply (7), and we are done.

Proving $(***)$. Take a σ''. Assume that

(12) $\overline{M} \cdot M;\ \sigma \rightsquigarrow^* \sigma''$

(13) $\overline{M} \cdot M, \sigma'' \models \mathtt{extl}.$

We apply lemma 1, part 2 on (12) and see that there are two cases

1st Case $\overline{M} \cdot M; \sigma_a \rightsquigarrow^*_{e,p} \sigma''$

That is, the execution from σ_a to σ'' goes only through external states. We use (11), and that A_{inv} is encapsulated, and are done with lemma F.28, part 1.

2nd Case for some σ_c, σ_d. we have

$\overline{M} \cdot M; \sigma_a \rightsquigarrow^*_{e,p} \sigma_c\ \wedge\ \overline{M} \cdot M;\ \sigma_c \rightsquigarrow \sigma_d\ \wedge\ M, \sigma_d \models \mathtt{pub} \wedge \overline{M} \cdot M;\ \sigma_d \rightsquigarrow^* \sigma'$

We apply similar arguments as in steps (17)-(23) and obtain

(14) $\ \ M, \sigma_c \models A_{inv}[\overline{\alpha/x}]$

State σ_c is a public, internal state; therefore there exists a Hoare proof that it preserves the invariant. By applying the inductive hypothesis, and the fact that $\overline{z} = \overline{x}$, we obtain:

(14) $\ \ M, \sigma'' \models A_{inv}[\overline{\alpha/z}]$

Call_Ext_Adapt_Strong is very similar to Call_Ext_Adaprt. We will summarize the similar steps, and highlight the differences in green .

Therefore, there exist u, y_o, $\overline{C}$, D, $\overline{y}$, ands A_{inv}, such that

(5) $\sigma.\mathtt{cont} \stackrel{\text{txt}}{=} u := y_0.m(\overline{y}),$

(6) $\vdash\ \ M\ :\ \forall \overline{x:C}.\{A_{inv}\}$,

(7) $A \stackrel{\text{txt}}{=} y_0 : \texttt{external}, \overline{x:C}\ \wedge\ A_{inv}\ \wedge A_{inv} \neg\!\triangledown(y_0, \overline{y})$,

$A' \stackrel{\text{txt}}{=} A_{inv}\ \wedge A_{inv} \neg\!\triangledown(y_0, \overline{y})$,

$A'' \stackrel{\text{txt}}{=} A_{inv}$.

Also,

(8) $\overline{M} \cdot M;\ \sigma \rightsquigarrow \sigma_a$,

By similar steps to (8a)-(10) from the previous case, we obtain

(10a) $\quad M, \sigma, k \models A_{inv}[\overline{\alpha/z}] \wedge\ (\ (\overline{x:C})[\overline{\alpha/z}]\ \wedge\ (A_{inv}[\overline{\alpha/z}])\ ) \triangledown y_0, \overline{y}$.

We now apply lemma apply Lemma G.40, part b.. This gives that

(11) $M, \sigma_a, k \models (\ \ (\overline{x:C})[\overline{\alpha/z}]\ \wedge\ A_{inv}[\overline{\alpha/z}] \wedge\ (\ (\overline{x:C})[\overline{\alpha/z}]) \triangledown y_0, \overline{y}\ \ )$.

the rest is similar to earlier cases

End Proof

G.17 Proof Sketch of Theorem 9.3 – part (B)

Proof Sketch By induction on the cases for the specification S. If it is a method spec, then the theorem follows from 9.3. If it is a conjunction, then by inductive hypothesis.
The interesting case is $S \stackrel{\text{txt}}{=} \forall \overline{x:C}.\{A\}$.
Assume that $M, \sigma, k \models A[\overline{\alpha/x}]$, that $M, \sigma \models \texttt{extl}$, that $M \cdot \overline{M};\ \sigma \rightsquigarrow^* \sigma'$, and that $M, \sigma \models \texttt{extl}$,
We want to show that $M, \sigma', k \models A[\overline{\alpha/x}]$.
Then, by lemma G.27, we obtain that either

(1) $\overline{M} \cdot M; \sigma \rightsquigarrow^*_{e,p} \sigma'$, or

(2) $\exists \sigma_1, \sigma_2.[\ \overline{M} \cdot M; \sigma \rightsquigarrow^*_{e,p} \sigma_1 \wedge\ \overline{M} \cdot M;\ \sigma_1 \rightsquigarrow \sigma_2 \wedge\ M, \sigma_2 \models \texttt{pub} \wedge \overline{M} \cdot M;\ \sigma_2 \rightsquigarrow^* \sigma'\ ]$

In Case (1), we apply G.28, part (3). In order to fulfill the second premise of Lemma G.28, part (3), we make use of the fact that $\vdash\ M$, apply the rule Method, and theorem 9.3. This gives us $M, \sigma', k \models A[\overline{\alpha/x}]$
In Case (2), we proceed as in (1) and obtain that $M, \sigma_1, k \models A[\overline{\alpha/x}]$. Because $M \vdash Enc(A)$, we also obtain that $M, \sigma_2, k \models A[\overline{\alpha/x}]$. Since we are now executing a public method, and because $\vdash M$, we can apply Invariant, and theorem 9.3, and obtain $M, \sigma', k \models A[\overline{\alpha/x}]$
End Proof Sketch

(6) $\vdash\ M : \forall \overline{x:C}.\{A_{inv}\}$,

(7) $A \stackrel{\text{txt}}{=} y_0 : \texttt{external}, \overline{x:C} \ \wedge\ A_{inv}\ \wedge A_{inv} \neg\!\triangledown (y_0, \overline{y})$,

$A' \stackrel{\text{txt}}{=} A_{inv}\ \wedge A_{inv} \neg\!\triangledown (y_0, \overline{y})$,

$A'' \stackrel{\text{txt}}{=} A_{inv}$.

Also,

(8) $\overline{M} \cdot M;\ \sigma \rightsquigarrow \sigma_a$,

By similar steps to (8a)-(10) from the previous case, we obtain

(10a) $M, \sigma, k \models A_{inv}[\overline{\alpha/z}] \wedge (\ (\overline{x:C})[\overline{\alpha/z}]\ \wedge\ (A_{inv}[\overline{\alpha/z}])\ ) \triangledown y_0, \overline{y}$.

We now apply lemma apply Lemma F.40, part b.. This gives that

(11) $M, \sigma_a, k \models (\ \ (\overline{x:C})[\overline{\alpha/z}]\ \wedge\ A_{inv}[\overline{\alpha/z}] \wedge\ (\ (\overline{x:C})[\overline{\alpha/z}]) \triangledown y_0, \overline{y}\ \ )$.

the rest is similar to earlier cases

End Proof

F.17 Proof Sketch of Theorem 9.3 – part (B)

Proof Sketch By induction on the cases for the specification S. If it is a method spec, then the theorem follows from 9.3. If it is a conjunction, then by inductive hypothesis.
The interesting case is $S \stackrel{\text{txt}}{=} \forall \overline{x:C}.\{A\}$.
Assume that $M, \sigma, k \models A[\overline{\alpha/x}]$, that $M, \sigma \models \texttt{extl}$, that $M \cdot \overline{M};\ \sigma \rightsquigarrow^* \sigma'$, and that $M, \sigma \models \texttt{extl}$,
We want to show that $M, \sigma', k \models A[\overline{\alpha/x}]$.
Then, by lemma F.27, we obtain that either

(1) $\overline{M} \cdot M; \sigma \rightsquigarrow^*_{\mathfrak{e},p} \sigma'$, or

(2) $\exists \sigma_1, \sigma_2.[\ \overline{M} \cdot M; \sigma \rightsquigarrow^*_{\mathfrak{e},p} \sigma_1 \wedge\ \overline{M} \cdot M;\ \sigma_1 \rightsquigarrow \sigma_2 \wedge\ M, \sigma_2 \models \texttt{pub} \wedge \overline{M} \cdot M;\ \sigma_2 \rightsquigarrow^* \sigma'\ ]$

In Case (1), we apply F.28, part (3). In order to fulfill the second premise of Lemma F.28, part (3), we make use of the fact that $\vdash M$, apply the rule Method, and theorem 9.3. This gives us $M, \sigma', k \models A[\overline{\alpha/x}]$
In Case (2), we proceed as in (1) and obtain that $M, \sigma_1, k \models A[\overline{\alpha/x}]$. Because $M \vdash Enc(A)$, we also obtain that $M, \sigma_2, k \models A[\overline{\alpha/x}]$. Since we are now executing a public method, and because $\vdash M$, we can apply Invariant, and theorem 9.3, and obtain $M, \sigma', k \models A[\overline{\alpha/x}]$

End Proof Sketch

H PROVING TAMED EFFECTS FOR THE SHOP/ACCOUNT EXAMPLE

In Section 2 we introduced a `Shop` that allows clients to make purchases through the `buy` method. The body if this method includes a method call to an unknown external object (`buyer.pay(...)`).

In this section we use our Hoare logic from Section 8 to outline the proof that the `buy` method does not expose the `Shop`'s `Account`, its `Key`, or allow the `Account`'s balance to be illicitly modified.

We outline the proof that $M_{good} \vdash S_2$, and that $M_{fine} \vdash S_2$. We also show why $M_{bad} \nvdash S_2$.

We first extend the semantics and the logic to deal with scalars (§H.1). We then extend the Hoare Logic with rules for conditionals, case analysis, and a contradiction rule (§??). We then rewrite the code of M_{good} and so M_{fine} so that it adheres to the syntax as defined in Fig. 4 (§H.2). We extend the specification S_2, so that is also makes a specification for the private method `set` (§H.3). After that, we outline the proofs (§H.5) – these proofs have been mechanized in Coq, and the source code will be submitted as an artefact. Finally, we discuss why $M_{bad} \nvdash S_2$ (§??).

H.1 Extend the semantics and Hoare logic to accommodate scalars and conditionals

We extend the notion of protection to also allow it to apply to scalars.

Definition H.1 (Satisfaction of Assertions – Protected From). extending the definition of Def 5.4. We use α to range over addresses, β to range over scalars, and γ to range over addresses or scalars. We define $M, \sigma \models \langle\gamma\rangle \not\leftrightarrow \gamma_o$ as:

(1) $M, \sigma \models \langle\alpha\rangle \not\leftrightarrow \alpha_o \quad \triangleq$
- $\alpha \neq \alpha_0$, and
- $\forall n \in \mathbb{N}.\forall f_1, ... f_n..[\ \lfloor\alpha_o.f_1...f_n\rfloor_\sigma = \alpha \quad \Longrightarrow \quad M, \sigma \models \lfloor\alpha_o.f_1...f_{n-1}\rfloor_\sigma : C \ \wedge\ C \in M\ ]$

(2) $M, \sigma \models \langle\gamma\rangle \not\leftrightarrow \beta_o \quad \triangleq \quad true$

(3) $M, \sigma \models \langle\beta\rangle \not\leftrightarrow \alpha_o \quad \triangleq \quad false$

(4) $M, \sigma \models \langle e\rangle \not\leftrightarrow e_o \quad \triangleq$
$\exists\gamma, \gamma_o.[\ M, \sigma, e \hookrightarrow \gamma\ \wedge M, \sigma, e_0 \hookrightarrow \gamma_0\ \wedge\ M, \sigma \models \langle\gamma\rangle \not\leftrightarrow \gamma_o\ ]$

The definition from above gives rise to further cases of protection; we supplement the triples from Fig. 6 with some further inference rules, given in Fig. ??.

$$M \vdash x : \texttt{int} \longrightarrow \langle y\rangle \not\leftrightarrow x \quad [\textsc{Prot-Int}] \qquad M \vdash x : \texttt{bool} \longrightarrow \langle y\rangle \not\leftrightarrow x \quad [\textsc{Prot-Bool}]$$

$$M \vdash x : \texttt{str} \longrightarrow \langle y\rangle \not\leftrightarrow x \quad [\textsc{Prot-Str1}] \qquad M \vdash \langle e\rangle \not\leftrightarrow e' \longrightarrow e \neq e' \quad [\textsc{Prot-Neq}]$$

Fig. 14. Protection for Scalar Types

H.2 Expressing the `Shop` example in the syntax from Fig. 4

We now express our example in the syntax of Fig. 4. For this, we add a return type to each of the methods; We turn all local variables to parameter; We add an explicit assignment to the variable `res`: and We add a temporary variable `tmp` to which we assign the result of our `void` methods. For simplicity, we allow the shorthands += and −=. And we also allow definition of local variables, *e.g.* `int price := ..`

```
module M_good
  ...
  class Shop
    field accnt : Account,
```

G Proving Tamed Effects for the Shop/Account Example

In Section 2 we introduced a `Shop` that allows clients to make purchases through the `buy` method. The body if this method includes a method call to an unknown external object (`buyer.pay(...)`).

In this section we use our Hoare logic from Section 8 to outline the proof that the `buy` method does not expose the `Shop`'s `Account`, its `Key`, or allow the `Account`'s balance to be illicitly modified.

We outline the proof that $M_{good} \vdash S_2$, and that $M_{fine} \vdash S_2$. We also show why $M_{bad} \nvdash S_2$.

We first extend the semantics and the logic to deal with scalars (§G.1). We then extend the Hoare Logic with rules for conditionals, case analysis, and a contradiction rule (§??). We then rewrite the code of M_{good} and so M_{fine} so that it adheres to the syntax as defined in Fig. 4 (§G.2). We extend the specification S_2, so that is also makes a specification for the private method `set` (§G.3). After that, we outline the proofs (§G.5) – these proofs have been mechanized in Coq, and the source code will be submitted as an artefact. Finally, we discuss why $M_{bad} \nvdash S_2$ (§??).

G.1 Extend the semantics and Hoare logic to accommodate scalars and conditionals

We extend the notion of protection to also allow it to apply to scalars.

Definition G.1 (Satisfaction of Assertions – Protected From)**.** extending the definition of Def 5.4. We use α to range over addresses, β to range over scalars, and γ to range over addresses or scalars. We define $M, \sigma \models \langle\gamma\rangle \not\leftrightarrow \gamma_o$ as:

(1) $M, \sigma \models \langle\alpha\rangle \not\leftrightarrow \alpha_o \quad \triangleq$
  - $\alpha \neq \alpha_0$, and
  - $\forall n \in \mathbb{N}.\forall f_1, ...f_n..[\ \lfloor \alpha_o.f_1...f_n \rfloor_\sigma = \alpha \quad \Longrightarrow \quad M, \sigma \models \lfloor \alpha_o.f_1...f_{n-1} \rfloor_\sigma : C \ \wedge\ C \in M\ ]$

(2) $M, \sigma \models \langle\gamma\rangle \not\leftrightarrow \beta_o \quad \triangleq \quad true$

(3) $M, \sigma \models \langle\beta\rangle \not\leftrightarrow \alpha_o \quad \triangleq \quad false$

(4) $M, \sigma \models \langle e\rangle \not\leftrightarrow e_o \quad \triangleq$
$\exists \gamma, \gamma_o.[\ M, \sigma, e \hookrightarrow \gamma\ \wedge M, \sigma, e_0 \hookrightarrow \gamma_0\ \wedge\ M, \sigma \models \langle\gamma\rangle \not\leftrightarrow \gamma_o\ ]$

The definition from above gives rise to further cases of protection; we supplement the triples from Fig. 6 with some further inference rules, given in Fig. ??.

$$M \vdash x : \texttt{int} \rightarrow \langle y\rangle \not\leftrightarrow x \quad [\text{PROT-INT}] \qquad M \vdash x : \texttt{bool} \rightarrow \langle y\rangle \not\leftrightarrow x \quad [\text{PROT-BOOL}]$$

$$M \vdash x : \texttt{str} \rightarrow \langle y\rangle \not\leftrightarrow x \quad [\text{PROT-STR1}] \qquad M \vdash \langle e\rangle \not\leftrightarrow e' \rightarrow e \neq e' \quad [\text{PROT-NEQ}]$$

Fig. 14. Protection for Scalar Types

G.2 Expressing the `Shop` example in the syntax from Fig. 4

We now express our example in the syntax of Fig. 4. For this, we add a return type to each of the methods; We turn all local variables to parameter; We add an explicit assignment to the variable `res`: and We add a temporary variable `tmp` to which we assign the result of our `void` methods. For simplicity, we allow the shorthands += and −=. And we also allow definition of local variables, *e.g.* `int price := ..`

```
module M_good
  ...
  class Shop
    field accnt : Account,
```

```
    field invntry : Inventory,
    field clients: ..

    public method buy(buyer:external, anItem:Item, price: int,
            myAccnt: Account, oldBalance: int, newBalance: int, tmp:int) : int
      price := anItem.price;
      myAccnt := this.accnt;
      oldBalance := myAccnt.blnce;
      tmp := buyer.pay(myAccnt, price)     // external call!
      newBalance := myAccnt.blnce;
      if (newBalance == oldBalance+price) then
          tmp := this.send(buyer,anItem)
      else
         tmp := buyer.tell("you have not paid me") ;
      res := 0

      private method send(buyer:external, anItem:Item) : int
       ...
  class Account
    field blnce : int
    field key : Key

    public method transfer(dest:Account, key':Key, amt:nat) :int
      if (this.key==key') then
        this.blnce-=amt;
        dest.blnce+=amt
      else
        res := 0
      res := 0

     public method set(key':Key) : int
      if (this.key==null)  then
            this.key:=key'
      else
        res := 0
      res := 0
```

Remember that M_{fine} is identical to M_{good}, except for the method `set`. We describe the module below.

```
module M_fine
  ...
  class Shop
     ...  as in M_good
  class Account
    field blnce : int
    field key : Key

    public method transfer(dest:Account, key':Key, amt:nat) :int
       ...  as in M_good

     public method set(key':Key, k'':Key) : int
      if (this.key==key')  then
            this.key:=key''
      else
        res := 0
      res := 0
```

```
    field invntry : Inventory,
    field clients: ..

    public method buy(buyer:external, anItem:Item, price: int,
            myAccnt: Account, oldBalance: int, newBalance: int, tmp:int) : int
      price := anItem.price;
      myAccnt := this.accnt;
      oldBalance := myAccnt.blnce;
      tmp := buyer.pay(myAccnt, price)     // external call!
      newBalance := myAccnt.blnce;
      if (newBalance == oldBalance+price) then
          tmp := this.send(buyer,anItem)
      else
         tmp := buyer.tell("you have not paid me") ;
      res := 0

      private method send(buyer:external, anItem:Item) : int
       ...
  class Account
    field blnce : int
    field key : Key

    public method transfer(dest:Account, key':Key, amt:nat) :int
      if (this.key==key') then
        this.blnce-=amt;
        dest.blnce+=amt
      else
        res := 0
      res := 0

     public method set(key':Key) : int
      if (this.key==null)  then
            this.key:=key'
      else
        res := 0
      res := 0
```

Remember that M_{fine} is identical to M_{good}, except for the method `set`. We describe the module below.

```
module M_fine
  ...
  class Shop
     ...  as in M_good
  class Account
    field blnce : int
    field key : Key

    public method transfer(dest:Account, key':Key, amt:nat) :int
       ...  as in M_good

     public method set(key':Key, k'':Key) : int
      if (this.key==key')  then
            this.key:=key''
      else
        res := 0
      res := 0
```

H.3 Proving that M_{good} and M_{fine} satisfy S_2

We redefine S_2 so that it also describes the behaviour of method `send`. and have:

$$
\begin{array}{lll}
S_{2a} & \triangleq & \{\, \texttt{a : Account} \land \texttt{e : external} \land \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow \texttt{e} \,\} \\
 & & \quad \texttt{private Shop :: send(buyer : external, anItem : Item)} \\
 & & \{\, \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow e \,\} \parallel \{\, \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow e \,\} \\
S_{2b} & \triangleq & \{\, \texttt{a : Account} \land \texttt{a.blnce} = \texttt{b} \,\} \\
 & & \quad \texttt{private Shop :: send(buyer : external, anItem : Item)} \\
 & & \{\, \texttt{a.blnce} = \texttt{b} \,\} \parallel \{\, \texttt{a.blnce} = \texttt{b} \,\} \\
S_{2,strong} & \triangleq & S_2 \land S_{2a} \land S_{2b}
\end{array}
$$

For brevity we only show that `buy` satisfies our scoped invariants, as the all other methods of the $\texttt{M}_{good}$ interface are relatively simple, and do not make any external calls.

To write our proofs more succinctly, we will use `ClassId::methId.body` as a shorthand for the method body of `methId` defined in `ClassId`.

Lemma H.2 (M_{good} satisfies $S_{2,strong}$). $M_{good} \vdash S_{2,strong}$

Proof Outline In order to prove that

$$M_{good} \vdash \mathbb{\forall}\texttt{a : Account}.\{\langle\!\langle \texttt{a.key} \rangle\!\rangle\}$$

we have to apply Invariant from Fig. 8. That is, for each class C defined in M_{good}, and for each public method m in C, with parameters $\overline{y : D}$, we have to prove that

$$
\begin{array}{l}
M_{good} \vdash \{\, \texttt{this : C}, \overline{y : D}, \texttt{a : Account} \land \langle\!\langle \texttt{a.key} \rangle\!\rangle \land \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow (\texttt{this}, \overline{y}) \,\} \\
\qquad\qquad \texttt{C :: m.body} \\
\qquad \{\, \langle\!\langle \texttt{a.key} \rangle\!\rangle \land \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow \texttt{res} \,\} \parallel \{\, \langle\!\langle \texttt{a.key} \rangle\!\rangle \,\}
\end{array}
$$

Thus, we need to prove three Hoare quadruples: one for `Shop::buy`, one for `Account::transfer`, and one for `Account::set`. That is, we have to prove that

$$
\begin{array}{ll}
(1?) & M_{good} \vdash \{\, \texttt{A}_{buy}, \texttt{a : Account} \land \langle\!\langle \texttt{a.key} \rangle\!\rangle \land \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow \texttt{Ids}_{buy} \,\} \\
 & \qquad\qquad \texttt{Shop :: buy.body} \\
 & \qquad \{\langle\!\langle \texttt{a.key} \rangle\!\rangle \land \langle\!\langle \texttt{a.key} \rangle\!\rangle \mathrel{-\!\!\!\nabla} \texttt{res}\} \parallel \{\langle\!\langle \texttt{a.key} \rangle\!\rangle\} \\
(2?) & M_{good} \vdash \{\, \texttt{A}_{trns}, \texttt{a : Account} \land \langle\!\langle \texttt{a.key} \rangle\!\rangle \land \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow \texttt{Ids}_{trns} \,\} \\
 & \qquad\qquad \texttt{Account :: transfer.body} \\
 & \qquad \{\langle\!\langle \texttt{a.key} \rangle\!\rangle \land \langle\!\langle \texttt{a.key} \rangle\!\rangle \mathrel{-\!\!\!\nabla} \texttt{res}\} \parallel \{\langle\!\langle \texttt{a.key} \rangle\!\rangle\} \\
(3?) & M_{good} \vdash \{\, \texttt{A}_{set}, \texttt{a : Account} \land \langle\!\langle \texttt{a.key} \rangle\!\rangle \land \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow \texttt{Ids}_{set} \,\} \\
 & \qquad\qquad \texttt{Account :: set.body} \\
 & \qquad \{\langle\!\langle \texttt{a.key} \rangle\!\rangle \land \langle\!\langle \texttt{a.key} \rangle\!\rangle \mathrel{-\!\!\!\nabla} \texttt{res}\} \parallel \{\langle\!\langle \texttt{a.key} \rangle\!\rangle\}
\end{array}
$$

where we are using ? to indicate that this needs to be proven, and where we are using the shorthands

$$
\begin{array}{lll}
\texttt{A}_{buy} & \triangleq & \texttt{this : Shop, buyer : external, anItem : Item, price : int,} \\
 & & \texttt{myAccnt : Account, oldBalance : int, newBalance : int, tmp : int.} \\
\texttt{Ids}_{buy} & \triangleq & \texttt{this, buyer, anItem, price, myAccnt, oldBalance, newBalance, tmp.} \\
\texttt{A}_{trns} & \triangleq & \texttt{this : Account, dest : Account, key' : Key, amt : nat} \\
\texttt{Ids}_{trns} & \triangleq & \texttt{this, dest, key', amt} \\
\texttt{A}_{set} & \triangleq & \texttt{this : Account, key' : Key, key'' : Key.} \\
\texttt{Ids}_{set} & \triangleq & \texttt{this, key', key''.}
\end{array}
$$

We will also need to prove that `Send` satisfies specifications S_{2a} and S_{2b}.

G.3 Proving that M_{good} and M_{fine} satisfy S_2

We redefine S_2 so that it also describes the behaviour of method `send`. and have:

$$
\begin{array}{lll}
S_{2a} & \triangleq & \{\, \texttt{a : Account} \wedge \texttt{e : external} \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow \texttt{e} \,\} \\
& & \quad \texttt{private Shop :: send(buyer : external, anItem : Item)} \\
& & \{\, \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow e \,\} \ \| \ \{ \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow e \} \\
S_{2b} & \triangleq & \{\, \texttt{a : Account} \wedge \texttt{a.blnce = b} \,\} \\
& & \quad \texttt{private Shop :: send(buyer : external, anItem : Item)} \\
& & \{\, \texttt{a.blnce = b} \,\} \ \| \ \{ \texttt{a.blnce = b} \} \\
S_{2,strong} & \triangleq & S_2 \ \wedge \ S_{2a} \ \wedge \ S_{2b}
\end{array}
$$

For brevity we only show that `buy` satisfies our scoped invariants, as the all other methods of the $\texttt{M}_{good}$ interface are relatively simple, and do not make any external calls.

To write our proofs more succinctly, we will use `ClassId::methId.body` as a shorthand for the method body of `methId` defined in `ClassId`.

Lemma G.2 (M_{good} satisfies $S_{2,strong}$). $M_{good} \vdash S_{2,strong}$

Proof Outline In order to prove that

$$M_{good} \vdash \mathbb{\forall} \texttt{a : Account}.\{\langle\!\langle \texttt{a.key} \rangle\!\rangle\}$$

we have to apply Invariant from Fig. 8. That is, for each class C defined in M_{good}, and for each public method m in C, with parameters $\overline{y : D}$, we have to prove that

$$
\begin{array}{l}
M_{good} \ \vdash \ \{\, \texttt{this : C}, \overline{y : D}, \texttt{a : Account} \ \wedge \ \langle\!\langle \texttt{a.key} \rangle\!\rangle \ \wedge \ \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow (\texttt{this}, \overline{y}) \,\} \\
\qquad\qquad\qquad \texttt{C :: m.body} \\
\qquad\quad \{\, \langle\!\langle \texttt{a.key} \rangle\!\rangle \ \wedge \ \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow \texttt{res} \,\} \,\|\, \{\, \langle\!\langle \texttt{a.key} \rangle\!\rangle \,\}
\end{array}
$$

Thus, we need to prove three Hoare quadruples: one for `Shop::buy`, one for `Account::transfer`, and one for `Account::set`. That is, we have to prove that

$$
\begin{array}{ll}
(1?) & M_{good} \ \vdash \ \{\, \texttt{A}_{buy}, \texttt{a : Account} \ \wedge \ \langle\!\langle \texttt{a.key} \rangle\!\rangle \ \wedge \ \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow \texttt{Ids}_{buy} \,\} \\
& \qquad\qquad \texttt{Shop :: buy.body} \\
& \qquad \{\langle\!\langle \texttt{a.key} \rangle\!\rangle \ \wedge \ \langle\!\langle \texttt{a.key} \rangle\!\rangle -\!\!\nabla \texttt{res}\} \ \| \ \{\langle\!\langle \texttt{a.key} \rangle\!\rangle\} \\
(2?) & M_{good} \vdash \{\, \texttt{A}_{trns}, \texttt{a : Account} \ \wedge \ \langle\!\langle \texttt{a.key} \rangle\!\rangle \ \wedge \ \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow \texttt{Ids}_{trns} \,\} \\
& \qquad\qquad \texttt{Account :: transfer.body} \\
& \qquad \{\langle\!\langle \texttt{a.key} \rangle\!\rangle \ \wedge \ \langle\!\langle \texttt{a.key} \rangle\!\rangle -\!\!\nabla \texttt{res}\} \ \| \ \{\langle\!\langle \texttt{a.key} \rangle\!\rangle\} \\
(3?) & M_{good} \vdash \{\, \texttt{A}_{set}, \texttt{a : Account} \ \wedge \ \langle\!\langle \texttt{a.key} \rangle\!\rangle \ \wedge \ \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow \texttt{Ids}_{set} \,\} \\
& \qquad\qquad \texttt{Account :: set.body} \\
& \qquad \{\langle\!\langle \texttt{a.key} \rangle\!\rangle \ \wedge \ \langle\!\langle \texttt{a.key} \rangle\!\rangle -\!\!\nabla \texttt{res}\} \ \| \ \{\langle\!\langle \texttt{a.key} \rangle\!\rangle\}
\end{array}
$$

where we are using ? to indicate that this needs to be proven, and where we are using the shorthands

$$
\begin{array}{lll}
\texttt{A}_{buy} & \triangleq & \texttt{this : Shop, buyer : external, anItem : Item, price : int,} \\
& & \texttt{myAccnt : Account, oldBalance : int, newBalance : int, tmp : int.} \\
\texttt{Ids}_{buy} & \triangleq & \texttt{this, buyer, anItem, price, myAccnt, oldBalance, newBalance, tmp.} \\
\texttt{A}_{trns} & \triangleq & \texttt{this : Account, dest : Account, key' : Key, amt : nat} \\
\texttt{Ids}_{trns} & \triangleq & \texttt{this, dest, key', amt} \\
\texttt{A}_{set} & \triangleq & \texttt{this : Account, key' : Key, key'' : Key.} \\
\texttt{Ids}_{set} & \triangleq & \texttt{this, key', key''.}
\end{array}
$$

We will also need to prove that `Send` satisfies specifications S_{2a} and S_{2b}.

We outline the proof of (1?) in Lemma H.4,and the proof of (2) in Lemma H.5. We do not prove (3), but will prove that `set` from M_{fine} satisfies S_2; shown in Lemma H.6 – ie for module M_{fine}.

□

We also want to prove that M_{fine} satisfies the specification $S_{2,strong}$.

Lemma H.3 (M_{fine} satisfies $S_{2,strong}$). $M_{fine} \vdash S_{2,strong}$

Proof Outline The proof of

$$M_{fine} \vdash \forall \texttt{a} : \texttt{Account}.\{\langle\!\langle \texttt{a.key} \rangle\!\rangle\}$$

goes along similar lines to the proof of lemma H.2. Thus, we need to prove the following three Hoare quadruples:

$$
\begin{array}{ll}
(4?) & M_{fine} \vdash \{\, \texttt{A}_{buy},\ \texttt{a} : \texttt{Account} \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow \texttt{Ids}_{buy} \,\} \\
& \qquad \texttt{Shop :: buy.body} \\
& \{\langle\!\langle \texttt{a.key} \rangle\!\rangle \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \text{-}\!\nabla \texttt{res}\} \ \ || \ \ \{\langle\!\langle \texttt{a.key} \rangle\!\rangle\} \\
(5?) & M_{fine} \vdash \{\, \texttt{A}_{trns},\ \texttt{a} : \texttt{Account} \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow \texttt{Ids}_{trns} \,\} \\
& \qquad \texttt{Account :: transfer.body} \\
& \{\langle\!\langle \texttt{a.key} \rangle\!\rangle \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \text{-}\!\nabla \texttt{res}\} \ \ || \ \ \{\langle\!\langle \texttt{a.key} \rangle\!\rangle\} \\
(6?) & M_{fine} \vdash \{\, \texttt{A}_{set},\ \texttt{a} : \texttt{Account} \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow \texttt{Ids}_{set} \,\} \\
& \qquad \texttt{Account :: set.body} \\
& \{\langle\!\langle \texttt{a.key} \rangle\!\rangle \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \text{-}\!\nabla \texttt{res}\} \ \ || \ \ \{\langle\!\langle \texttt{a.key} \rangle\!\rangle\}
\end{array}
$$

The proof of (4?) is identical to that of (1?); the proof of (5?) is identical to that of (2?). We outline the proof (6?) in Lemma H.6 in §H.3.

□

Lemma H.4 (`Shop::buy` satisfies S_2).

$$
\begin{array}{ll}
(1) & M_{good} \vdash \{\, \texttt{A}_{buy}\ \texttt{a} : \texttt{Account} \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow \texttt{Ids}_{buy} \,\} \\
& \qquad \texttt{Shop :: buy.body} \\
& \{\langle\!\langle \texttt{a.key} \rangle\!\rangle \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \text{-}\!\nabla \texttt{res}\} \ \ || \ \ \{\langle\!\langle \texttt{a.key} \rangle\!\rangle\}
\end{array}
$$

Proof Outline We will use the shorthand $\texttt{A}_1 \triangleq \texttt{A}_{buy}, \texttt{a} : \texttt{Account}$. We will split the proof into 1) proving that statements 10, 11, 12 preserve the protection of `a.key` from the `buyer`, 2) proving that the external call

1st Step: proving statements 10, 11, 12

We apply the underlying Hoare logic and prove that the statements on lines 10, 11, 12 do not affect the value of `a.key`, ie that for a $z \notin \{\texttt{price}, \texttt{myAccnt}, \texttt{oldBalance}\}$, we have

$$
\begin{array}{ll}
(10) & M_{good} \vdash_{ul} \{\, \texttt{A}_1 \ \wedge\ z = \texttt{a.key}\} \\
& \qquad \texttt{price:=anItem.price;} \\
& \qquad \texttt{myAccnt:=this.accnt;} \\
& \qquad \texttt{oldBalance := myAccnt.blnce;} \\
& \{z = \texttt{a.key}\}
\end{array}
$$

We then apply Embed_UL, Prot-1 and Prot-2 and Combine and and Types-2 on (10) and use the shorthand $\texttt{stmts}_{10,11,12}$ for the statements on lines 10, 11 and 12, and obtain:

We outline the proof of (1?) in Lemma G.4,and the proof of (2) in Lemma G.5. We do not prove (3), but will prove that `set` from M_{fine} satisfies S_2; shown in Lemma G.6 – ie for module M_{fine}.

□

We also want to prove that M_{fine} satisfies the specification $S_{2,strong}$.

Lemma G.3 (M_{fine} satisfies $S_{2,strong}$). $M_{fine} \vdash S_{2,strong}$

Proof Outline The proof of

$$M_{fine} \vdash \forall \mathtt{a}:\mathtt{Account}.\{\langle\!\langle \mathtt{a.key} \rangle\!\rangle\}$$

goes along similar lines to the proof of lemma G.2. Thus, we need to prove the following three Hoare quadruples:

$$
\begin{array}{ll}
(4?) & M_{fine} \vdash \{\, \mathtt{A}_{buy},\ \mathtt{a}:\mathtt{Account}\ \wedge\ \langle\!\langle \mathtt{a.key} \rangle\!\rangle\ \wedge\ \langle\!\langle \mathtt{a.key} \rangle\!\rangle \not\leftrightarrow \mathtt{Ids}_{buy}\,\} \\
& \qquad \mathtt{Shop :: buy.body} \\
& \{\langle\!\langle \mathtt{a.key} \rangle\!\rangle\ \wedge\ \langle\!\langle \mathtt{a.key} \rangle\!\rangle \not\!\!\nabla \mathtt{res}\}\ \ ||\ \ \{\langle\!\langle \mathtt{a.key} \rangle\!\rangle\} \\
(5?) & M_{fine} \vdash \{\, \mathtt{A}_{trns},\ \mathtt{a}:\mathtt{Account}\ \wedge\ \langle\!\langle \mathtt{a.key} \rangle\!\rangle\ \wedge\ \langle\!\langle \mathtt{a.key} \rangle\!\rangle \not\leftrightarrow \mathtt{Ids}_{trns}\,\} \\
& \qquad \mathtt{Account :: transfer.body} \\
& \{\langle\!\langle \mathtt{a.key} \rangle\!\rangle\ \wedge\ \langle\!\langle \mathtt{a.key} \rangle\!\rangle \not\!\!\nabla \mathtt{res}\}\ \ ||\ \ \{\langle\!\langle \mathtt{a.key} \rangle\!\rangle\} \\
(6?) & M_{fine} \vdash \{\, \mathtt{A}_{set},\ \mathtt{a}:\mathtt{Account}\ \wedge\ \langle\!\langle \mathtt{a.key} \rangle\!\rangle\ \wedge\ \langle\!\langle \mathtt{a.key} \rangle\!\rangle \not\leftrightarrow \mathtt{Ids}_{set}\,\} \\
& \qquad \mathtt{Account :: set.body} \\
& \{\langle\!\langle \mathtt{a.key} \rangle\!\rangle\ \wedge\ \langle\!\langle \mathtt{a.key} \rangle\!\rangle \not\!\!\nabla \mathtt{res}\}\ \ ||\ \ \{\langle\!\langle \mathtt{a.key} \rangle\!\rangle\}
\end{array}
$$

The proof of (4?) is identical to that of (1?); the proof of (5?) is identical to that of (2?). We outline the proof (6?) in Lemma G.6 in §G.3.

□

Lemma G.4 (`Shop::buy` satisfies S_2).

$$
\begin{array}{ll}
(1) & M_{good} \vdash \{\, \mathtt{A}_{buy}\ \mathtt{a}:\mathtt{Account}\ \wedge\ \langle\!\langle \mathtt{a.key} \rangle\!\rangle\ \wedge\ \langle\!\langle \mathtt{a.key} \rangle\!\rangle \not\leftrightarrow \mathtt{Ids}_{buy}\,\} \\
& \qquad \mathtt{Shop :: buy.body} \\
& \{\langle\!\langle \mathtt{a.key} \rangle\!\rangle\ \wedge\ \langle\!\langle \mathtt{a.key} \rangle\!\rangle \not\!\!\nabla \mathtt{res}\}\ \ ||\ \ \{\langle\!\langle \mathtt{a.key} \rangle\!\rangle\}
\end{array}
$$

Proof Outline We will use the shorthand $\mathtt{A}_1 \triangleq \mathtt{A}_{buy},\ \mathtt{a}:\mathtt{Account}$. We will split the proof into 1) proving that statements 10, 11, 12 preserve the protection of `a.key` from the `buyer`, 2) proving that the external call

1st Step: proving statements 10, 11, 12

We apply the underlying Hoare logic and prove that the statements on lines 10, 11, 12 do not affect the value of `a.key`, ie that for a $z \notin \{\mathtt{price}, \mathtt{myAccnt}, \mathtt{oldBalance}\}$, we have

$$
\begin{array}{ll}
(10) & M_{good} \vdash_{ul} \{\, \mathtt{A}_1\ \wedge\ z = \mathtt{a.key}\} \\
& \qquad\qquad \mathtt{price:=anItem.price;} \\
& \qquad\qquad \mathtt{myAccnt:=this.accnt;} \\
& \qquad\qquad \mathtt{oldBalance\ :=\ myAccnt.blnce;} \\
& \qquad \{z = \mathtt{a.key}\}
\end{array}
$$

We then apply Embed_UL, Prot-1 and Prot-2 and Combine and and Types-2 on (10) and use the shorthand $\mathtt{stmts}_{10,11,12}$ for the statements on lines 10, 11 and 12, and obtain:

$$(11) \quad M_{good} \vdash \{\, \texttt{A}_1 \ \wedge\ \langle\!\langle \texttt{a.key} \rangle\!\rangle \ \wedge\ \langle\!\langle \texttt{buyer} \rangle\!\rangle \not\leftrightarrow \texttt{a.key} \}$$
$$\texttt{stmts}_{10,11,12}$$
$$\{\, \langle\!\langle \texttt{a.key} \rangle\!\rangle \ \wedge\ \langle\!\langle \texttt{buyer} \rangle\!\rangle \not\leftrightarrow \texttt{a.key} \}$$

We apply Mɪᴅ on (11) and obtain

$$(12) \quad M_{good} \vdash \{\, \texttt{A}_1 \ \wedge\ \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow \texttt{buyer} \,\}$$
$$\texttt{stmts}_{10,11,12}$$
$$\{\, \texttt{A}_1 \ \wedge\ \langle\!\langle \texttt{a.key} \rangle\!\rangle \ \wedge\ \langle\!\langle \texttt{buyer} \rangle\!\rangle \not\leftrightarrow \texttt{a.key} \,\} \ ||$$
$$\{\, \langle\!\langle \texttt{a.key} \rangle\!\rangle \,\}$$

2nd Step: Proving the External Call

We now need to prove that the external method call `buyer.pay(this.accnt, price)` protects the `key`. i.e.

$$(13?) \quad M_{good} \vdash \{\, \texttt{A}_1 \ \wedge\ \langle\!\langle \texttt{a.key} \rangle\!\rangle, \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow \texttt{buyer} \}$$
$$\texttt{tmp := buyer.pay(myAccnt, price)}$$
$$\{\ \ \texttt{A}_1 \ \wedge\ \langle\!\langle \texttt{a.key} \rangle\!\rangle \ \wedge\ \langle\!\langle \texttt{buyer} \rangle\!\rangle \not\leftrightarrow \texttt{a.key} \} \ \ ||$$
$$\{\, \langle\!\langle \texttt{a.key} \rangle\!\rangle \,\}$$

We use that $M \vdash \forall \texttt{a} : \texttt{Account}.\{\langle\!\langle \texttt{a.key} \rangle\!\rangle\}$ and obtain

$$(14) \quad M_{good} \vdash \{\, \texttt{buyer} : \texttt{external}, \langle\!\langle \texttt{a.key} \rangle\!\rangle \ \wedge\ \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow (\texttt{buyer}, \texttt{myAccnt}, \texttt{price}) \,\}$$
$$\texttt{tmp := buyer.pay(myAccnt, price)}$$
$$\{\, \langle\!\langle \texttt{a.key} \rangle\!\rangle \ \wedge\ \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow (\texttt{buyer}, \texttt{myAccnt}, \texttt{price}) \,\} \ \ ||$$
$$\{\, \langle\!\langle \texttt{a.key} \rangle\!\rangle \,\}$$

In order to obtain (13?) out of (14), we apply Pʀᴏᴛ-Iɴᴛʟ and Pʀᴏᴛ-Iɴᴛ$_1$, which gives us

(15) $\quad M_{good} \vdash \texttt{A}_1 \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \longrightarrow \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow \texttt{myAccnt}$

(16) $\quad M_{good} \vdash \texttt{A}_1 \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \longrightarrow \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow \texttt{price}$

We apply Cᴏɴsᴇǫᴜ on (15), (16) and (14) and obtain (13)!

□

Lemma H.5 (`transfer` satisfies S_2).

$$(2) \quad M_{good} \vdash \{\, \texttt{A}_{trns}, \texttt{a} : \texttt{Account} \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow \texttt{Ids}_{trns} \,\}$$
$$\texttt{Account} :: \texttt{transfer.body}$$
$$\{\langle\!\langle \texttt{a.key} \rangle\!\rangle \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\nabla \texttt{res}\} \ \ || \ \ \{\langle\!\langle \texttt{a.key} \rangle\!\rangle\}$$

Pʀᴏᴏғ Oᴜᴛʟɪɴᴇ

To prove (2), we will need to prove that

$$(11)\quad M_{good} \vdash \{\, \mathtt{A}_1 \ \wedge\ \langle\!\langle \mathtt{a.key} \rangle\!\rangle \ \wedge\ \langle\!\langle \mathtt{buyer} \rangle\!\rangle \not\leftrightarrow \mathtt{a.key} \}$$
$$\mathtt{stmts}_{10,11,12}$$
$$\{\, \langle\!\langle \mathtt{a.key} \rangle\!\rangle \ \wedge\ \langle\!\langle \mathtt{buyer} \rangle\!\rangle \not\leftrightarrow \mathtt{a.key} \}$$

We apply Mid on (11) and obtain

$$(12)\quad M_{good} \vdash \{\, \mathtt{A}_1 \ \wedge\ \langle\!\langle \mathtt{a.key} \rangle\!\rangle \not\leftrightarrow \mathtt{buyer} \,\}$$
$$\mathtt{stmts}_{10,11,12}$$
$$\{\, \mathtt{A}_1 \ \wedge\ \langle\!\langle \mathtt{a.key} \rangle\!\rangle \ \wedge\ \langle\!\langle \mathtt{buyer} \rangle\!\rangle \not\leftrightarrow \mathtt{a.key} \,\} \ \ ||$$
$$\{\, \langle\!\langle \mathtt{a.key} \rangle\!\rangle \,\}$$

2nd Step: Proving the External Call

We now need to prove that the external method call `buyer.pay(this.accnt, price)` protects the `key`. i.e.

$$(13?)\quad M_{good} \vdash \{\, \mathtt{A}_1 \ \wedge\ \langle\!\langle \mathtt{a.key} \rangle\!\rangle, \wedge \langle\!\langle \mathtt{a.key} \rangle\!\rangle \not\leftrightarrow \mathtt{buyer} \}$$
$$\mathtt{tmp\ :=\ buyer.pay(myAccnt,\ price)}$$
$$\{\ \ \mathtt{A}_1 \ \wedge\ \langle\!\langle \mathtt{a.key} \rangle\!\rangle \ \wedge\ \langle\!\langle \mathtt{buyer} \rangle\!\rangle \not\leftrightarrow \mathtt{a.key} \} \ \ ||$$
$$\{\, \langle\!\langle \mathtt{a.key} \rangle\!\rangle \,\}$$

We use that $M \vdash \mathbb{A}\mathtt{a} : \mathtt{Account}.\{\langle\!\langle \mathtt{a.key} \rangle\!\rangle\}$ and obtain

$$(14)\quad M_{good} \vdash \{\, \mathtt{buyer} : \mathtt{external},\ \langle\!\langle \mathtt{a.key} \rangle\!\rangle \ \wedge\ \langle\!\langle \mathtt{a.key} \rangle\!\rangle \not\leftrightarrow (\mathtt{buyer}, \mathtt{myAccnt}, \mathtt{price}) \,\}$$
$$\mathtt{tmp\ :=\ buyer.pay(myAccnt,\ price)}$$
$$\{\, \langle\!\langle \mathtt{a.key} \rangle\!\rangle \ \wedge\ \langle\!\langle \mathtt{a.key} \rangle\!\rangle \not\leftrightarrow (\mathtt{buyer}, \mathtt{myAccnt}, \mathtt{price}) \,\} \ \ ||$$
$$\{\, \langle\!\langle \mathtt{a.key} \rangle\!\rangle \,\}$$

In order to obtain (13?) out of (14), we apply Prot-Intl and Prot-Int$_1$, which gives us

(15) $\quad M_{good} \vdash \mathtt{A}_1 \wedge \langle\!\langle \mathtt{a.key} \rangle\!\rangle \longrightarrow \langle\!\langle \mathtt{a.key} \rangle\!\rangle \not\leftrightarrow \mathtt{myAccnt}$

(16) $\quad M_{good} \vdash \mathtt{A}_1 \wedge \langle\!\langle \mathtt{a.key} \rangle\!\rangle \longrightarrow \langle\!\langle \mathtt{a.key} \rangle\!\rangle \not\leftrightarrow \mathtt{price}$

We apply Consequ on (15), (16) and (14) and obtain (13)!

□

Lemma G.5 (`transfer` satisfies S_2).

$$(2)\quad M_{good} \vdash \{\, \mathtt{A}_{trns},\ \mathtt{a} : \mathtt{Account} \ \wedge\ \langle\!\langle \mathtt{a.key} \rangle\!\rangle \ \wedge\ \langle\!\langle \mathtt{a.key} \rangle\!\rangle \not\leftrightarrow \mathtt{Ids}_{trns} \,\}$$
$$\mathtt{Account} :: \mathtt{transfer.body}$$
$$\{\langle\!\langle \mathtt{a.key} \rangle\!\rangle \ \wedge\ \langle\!\langle \mathtt{a.key} \rangle\!\rangle \!-\!\!\nabla \mathtt{res}\} \ \ || \ \ \{\langle\!\langle \mathtt{a.key} \rangle\!\rangle\}$$

Proof Outline

To prove (2), we will need to prove that

$$(21?)\quad M_{good} \vdash \{\, \mathbb{A}_{trns},\ \texttt{a}:\texttt{Account} \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow \texttt{Ids}_{trns} \,\}$$

```
if (this.key==key') then
 this.blnce:=this.blnce-amt
 dest.blnce:=dest.blnce+amt
else
 res:=0
res:=0
```

$$\{\langle\!\langle \texttt{a.key} \rangle\!\rangle \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \text{-}\!\nabla \texttt{res}\} \;\; || \;\; \{\langle\!\langle \texttt{a.key} \rangle\!\rangle\}$$

Using the underlying Hoare logic we can prove that no account's `key` gets modified, namely

$$(22)\quad M_{good} \vdash_{ul} \{\, \mathbb{A}_{trns},\ \texttt{a}:\texttt{Account} \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle$$

```
if (this.key==key') then
 this.blnce:=this.blnce-amt
 dest.blnce:=dest.blnce+amt
else
 res:=0
res:=0
```

$$\{\langle\!\langle \texttt{a.key} \rangle\!\rangle\}$$

Using (22) and [Prot-1], we obtain

$$(23)\quad M_{good} \vdash \{\, \mathbb{A}_{trns},\ \texttt{a}:\texttt{Account} \wedge z = \texttt{a.key}\}$$

```
if (this.key==key') then
 this.blnce:=this.blnce-amt
 dest.blnce:=dest.blnce+amt
else
 res:=0
res:=0
```

$$\{z = \texttt{a.key}\}$$

Using (23) and [Embed-UL], we obtain

$$
\begin{array}{ll}
(21?) & M_{good} \vdash \{\, \mathtt{A}_{trns},\ \texttt{a}:\texttt{Account} \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow \texttt{Ids}_{trns} \,\} \\
& \qquad\qquad \texttt{if (this.key==key') then} \\
& \qquad\qquad\ \ \texttt{this.blnce:=this.blnce-amt} \\
& \qquad\qquad\ \ \texttt{dest.blnce:=dest.blnce+amt} \\
& \qquad\qquad \texttt{else} \\
& \qquad\qquad\ \ \texttt{res:=0} \\
& \qquad\qquad \texttt{res:=0} \\
& \qquad \{\langle\!\langle \texttt{a.key} \rangle\!\rangle \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \mathrel{-\!\!\nabla} \texttt{res}\} \ \ || \ \ \{\langle\!\langle \texttt{a.key} \rangle\!\rangle\}
\end{array}
$$

Using the underlying Hoare logic we can prove that no account's `key` gets modified, namely

$$
\begin{array}{ll}
(22) & M_{good} \ \vdash_{ul} \{\, \mathtt{A}_{trns},\ \texttt{a}:\texttt{Account} \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \\
& \qquad\qquad \texttt{if (this.key==key') then} \\
& \qquad\qquad\ \ \texttt{this.blnce:=this.blnce-amt} \\
& \qquad\qquad\ \ \texttt{dest.blnce:=dest.blnce+amt} \\
& \qquad\qquad \texttt{else} \\
& \qquad\qquad\ \ \texttt{res:=0} \\
& \qquad\qquad \texttt{res:=0} \\
& \qquad \{\langle\!\langle \texttt{a.key} \rangle\!\rangle\}
\end{array}
$$

Using (22) and [Prot-1], we obtain

$$
\begin{array}{ll}
(23) & M_{good} \ \vdash \{\, \mathtt{A}_{trns},\ \texttt{a}:\texttt{Account} \wedge z = \texttt{a.key}\} \\
& \qquad\qquad \texttt{if (this.key==key') then} \\
& \qquad\qquad\ \ \texttt{this.blnce:=this.blnce-amt} \\
& \qquad\qquad\ \ \texttt{dest.blnce:=dest.blnce+amt} \\
& \qquad\qquad \texttt{else} \\
& \qquad\qquad\ \ \texttt{res:=0} \\
& \qquad\qquad \texttt{res:=0} \\
& \qquad \{z = \texttt{a.key}\}
\end{array}
$$

Using (23) and [Embed-UL], we obtain

$$(24)\quad M_{good} \vdash \{\, A_{trns},\ \texttt{a}:\texttt{Account} \wedge z = \texttt{a.key}\}$$

```
if (this.key==key') then
  this.blnce:=this.blnce-amt
  dest.blnce:=dest.blnce+amt
else
  res:=0
res:=0
```

$$\{z = \texttt{a.key}\}\ \ ||\ \ \{z = \texttt{a.key}\}$$

[Prot_Int] and the fact that z is an `int` gives us that $\langle\texttt{a.key}\rangle -\!\nabla\,\texttt{res}$. Using [Types], and [Prot_Int] and [Consequ] on (24) we obtain (21?).

□

We want to prove that this public method satisfies the specification $S_{2,strong}$, namely

Lemma H.6 (`set` satisfies S_2).

$$(6)\quad M_{fine} \vdash \{\, A_{set}\ \wedge\ \langle\texttt{a.key}\rangle\ \wedge\ \langle\texttt{a.key}\rangle \nleftrightarrow \texttt{Ids}_{set}\,\}$$

```
if (this.key==key') then
  this.key:=key"
else
  res:=0
res:=0
```

$$\{\langle\texttt{a.key}\rangle\ \wedge\ \langle\texttt{a.key}\rangle -\!\nabla\,\texttt{res}\}\ \ ||\ \ \{\langle\texttt{a.key}\rangle\}$$

Proof Outline We will be using the shorthand $A_2 \triangleq \texttt{a}:\texttt{Account},\ A_{set}$.

To prove (6), we will use the Sequence rule, and we want to prove

$$(61?)\quad M_{fine} \vdash \{\, A_2\ \wedge\ \langle\texttt{a.key}\rangle\ \wedge\ \langle\texttt{a.key}\rangle \nleftrightarrow \texttt{Ids}_{set}\,\}$$

```
if (this.key==key') then
  this.key:=key"
else
  res:=0
```

$$\{\, A_2\ \wedge\ \langle\texttt{a.key}\rangle\,\}\ \ ||\ \ \{\langle\texttt{a.key}\rangle\}$$

and that

$$(62?)\quad M_{fine} \vdash \{\, A_2\ \wedge \langle\texttt{a.key}\rangle\,\}$$

```
res:=0
```

$$\{\langle\texttt{a.key}\rangle\ \wedge\ \langle\texttt{a.key}\rangle -\!\nabla\,\texttt{res}\}\ \ ||\ \ \{\langle\texttt{a.key}\rangle\}$$

(62?) follows from the types, and Prot-Int$_1$, the fact that `a.key` did not change, and Prot-1.

We now want to prove (61?). For this, will apply the If-Rule. That is, we need to prove that

$$
\begin{array}{ll}
(24) & M_{good} \vdash \{\, A_{trns},\ \texttt{a} : \texttt{Account} \wedge z = \texttt{a.key} \,\} \\
& \quad \texttt{if (this.key==key') then} \\
& \quad\quad \texttt{this.blnce:=this.blnce-amt} \\
& \quad\quad \texttt{dest.blnce:=dest.blnce+amt} \\
& \quad \texttt{else} \\
& \quad\quad \texttt{res:=0} \\
& \quad \texttt{res:=0} \\
& \{ z = \texttt{a.key} \} \ \ || \ \ \{ z = \texttt{a.key} \}
\end{array}
$$

[Prot_Int] and the fact that z is an `int` gives us that $\langle\!\langle \texttt{a.key} \rangle\!\rangle \rightarrow\!\!\!\!\nabla\, \texttt{res}$. Using [Types], and [Prot_Int] and [Consequ] on (24) we obtain (21?).

□

We want to prove that this public method satisfies the specification $S_{2,strong}$, namely

Lemma G.6 (`set` satisfies S_2).

$$
\begin{array}{ll}
(6) & M_{fine} \vdash \{\, A_{set} \ \wedge\ \langle\!\langle \texttt{a.key} \rangle\!\rangle \ \wedge\ \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow \texttt{Ids}_{set} \,\} \\
& \quad \texttt{if (this.key==key') then} \\
& \quad\quad \texttt{this.key:=key''} \\
& \quad \texttt{else} \\
& \quad\quad \texttt{res:=0} \\
& \quad \texttt{res:=0} \\
& \{ \langle\!\langle \texttt{a.key} \rangle\!\rangle \ \wedge\ \langle\!\langle \texttt{a.key} \rangle\!\rangle \rightarrow\!\!\!\!\nabla\, \texttt{res} \} \ \ || \ \ \{ \langle\!\langle \texttt{a.key} \rangle\!\rangle \}
\end{array}
$$

Proof Outline We will be using the shorthand $A_2 \triangleq \texttt{a} : \texttt{Account},\ A_{set}$.

To prove (6), we will use the Sequence rule, and we want to prove

$$
\begin{array}{ll}
(61?) & M_{fine} \vdash \{\, A_2 \ \wedge\ \langle\!\langle \texttt{a.key} \rangle\!\rangle \ \wedge\ \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow \texttt{Ids}_{set} \,\} \\
& \quad \texttt{if (this.key==key') then} \\
& \quad\quad \texttt{this.key:=key''} \\
& \quad \texttt{else} \\
& \quad\quad \texttt{res:=0} \\
& \{\, A_2 \ \wedge\ \langle\!\langle \texttt{a.key} \rangle\!\rangle \,\} \ \ || \ \ \{ \langle\!\langle \texttt{a.key} \rangle\!\rangle \}
\end{array}
$$

and that

$$
\begin{array}{ll}
(62?) & M_{fine} \vdash \{\, A_2 \ \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \,\} \\
& \quad \texttt{res:=0} \\
& \{ \langle\!\langle \texttt{a.key} \rangle\!\rangle \ \wedge\ \langle\!\langle \texttt{a.key} \rangle\!\rangle \rightarrow\!\!\!\!\nabla\, \texttt{res} \} \ \ || \ \ \{ \langle\!\langle \texttt{a.key} \rangle\!\rangle \}
\end{array}
$$

(62?) follows from the types, and Prot-Int$_1$, the fact that `a.key` did not change, and Prot-1.

We now want to prove (61?). For this, will apply the If-Rule. That is, we need to prove that

$$(63?)\quad M_{fine} \vdash \{\, \texttt{A}_2 \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow \texttt{Ids}_{set} \wedge \texttt{this.key} = \texttt{key}' \,\}$$
$$\texttt{this.key:=key}''$$
$$\{\langle\!\langle \texttt{a.key} \rangle\!\rangle\} \;\;||\;\; \{\langle\!\langle \texttt{a.key} \rangle\!\rangle\}$$

and that

$$(64?)\quad M_{fine} \vdash \{\, \texttt{A}_2 \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow \texttt{Ids}_{set} \wedge \texttt{this.key} \neq \texttt{key}' \,\}$$
$$\texttt{res:=0}$$
$$\{\langle\!\langle \texttt{a.key} \rangle\!\rangle\,\} \;\;||\;\; \{\langle\!\langle \texttt{a.key} \rangle\!\rangle\}$$

(64?) follows easily from the fact that `a.key` did not change, and Prot-1.

We look at the proof of (63?). We will apply the Cases rule, and distinguish on whether `a.key=this.key`. That is, we want to prove that

$$(65?)\quad M_{fine} \vdash \{\, \texttt{A}_2 \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow \texttt{Ids}_{set} \wedge \texttt{this.key} = \texttt{key}' \wedge \texttt{this.key} = \texttt{a.key}\}$$
$$\texttt{this.key:=key}''$$
$$\{\langle\!\langle \texttt{a.key} \rangle\!\rangle\,\} \;\;||\;\; \{\langle\!\langle \texttt{a.key} \rangle\!\rangle\}$$

and that

$$(66?)\quad M_{fine} \vdash \{\, \texttt{A}_2 \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow \texttt{Ids}_{set} \wedge \texttt{this.key} = \texttt{key}' \wedge \texttt{this.key} \neq \texttt{a.key}' \,\}$$
$$\texttt{this.key:=key}''$$
$$\{\langle\!\langle \texttt{a.key} \rangle\!\rangle \;\;||\;\; \{\langle\!\langle \texttt{a.key} \rangle\!\rangle\}$$

We can prove (65?) through application of Absurd, ProtNeq, and Consequ, as follows

$$(61c)\quad M_{fine} \vdash \{\, false\}$$
$$\texttt{this.key:=key}''$$
$$\{\langle\!\langle \texttt{a.key} \rangle\!\rangle\,\} \;\;||\;\; \{\langle\!\langle \texttt{a.key} \rangle\!\rangle\}$$

By ProtNeq, we have $M_{fine} \vdash \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow \texttt{key}' \longrightarrow \texttt{a.key} \neq \texttt{key}'$, and therefore obtain

$$(61d)\quad M_{fine} \vdash \ldots \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow \texttt{Ids}_{set} \wedge \texttt{this.key} = \texttt{a.key} \wedge \texttt{this.key} = \texttt{key}' \longrightarrow false$$

We apply Consequ on (61c) and (61d) and obtain (61aa?).

We can prove (66?) by proving that `this.key≠a.key` implies that `this` ≠ `a` (by the underlying Hoare logic), which again implies that the assignment `this.key := ...` leaves the value of `a.key` unmodified. We apply Prot-1, and are done.

□

(63?) $M_{fine} \vdash \{\ \mathtt{A}_2\ \wedge\ \langle\!\langle \mathtt{a.key} \rangle\!\rangle\ \wedge\ \langle\!\langle \mathtt{a.key} \rangle\!\rangle \not\leftrightarrow \mathtt{Ids}_{set}\ \wedge\ \mathtt{this.key} = \mathtt{key'}\ \}$

$\mathtt{this.key{:=}key''}$

$\{\langle\!\langle \mathtt{a.key} \rangle\!\rangle\}\ \ ||\ \ \{\langle\!\langle \mathtt{a.key} \rangle\!\rangle\}$

and that

(64?) $M_{fine} \vdash \{\ \mathtt{A}_2\ \wedge\ \langle\!\langle \mathtt{a.key} \rangle\!\rangle\ \wedge\ \langle\!\langle \mathtt{a.key} \rangle\!\rangle \not\leftrightarrow \mathtt{Ids}_{set}\ \wedge\ \mathtt{this.key} \neq \mathtt{key'}\ \}$

$\mathtt{res{:=}0}$

$\{\langle\!\langle \mathtt{a.key} \rangle\!\rangle\ \}\ \ ||\ \ \{\langle\!\langle \mathtt{a.key} \rangle\!\rangle\}$

(64?) follows easily from the fact that `a.key` did not change, and Prot-1.

We look at the proof of (63?). We will apply the Cases rule, and distinguish on whether `a.key=this.key`. That is, we want to prove that

(65?) $M_{fine} \vdash \{\ \mathtt{A}_2\ \wedge\ \langle\!\langle \mathtt{a.key} \rangle\!\rangle\ \wedge\ \langle\!\langle \mathtt{a.key} \rangle\!\rangle \not\leftrightarrow \mathtt{Ids}_{set}\ \wedge\ \mathtt{this.key} = \mathtt{key'}\ \ \wedge\ \mathtt{this.key} = \mathtt{a.key}\}$

$\mathtt{this.key{:=}key''}$

$\{\langle\!\langle \mathtt{a.key} \rangle\!\rangle\ \}\ \ ||\ \ \{\langle\!\langle \mathtt{a.key} \rangle\!\rangle\}$

and that

(66?) $M_{fine} \vdash \{\ \mathtt{A}_2\ \wedge\ \langle\!\langle \mathtt{a.key} \rangle\!\rangle\ \wedge\ \langle\!\langle \mathtt{a.key} \rangle\!\rangle \not\leftrightarrow \mathtt{Ids}_{set}\ \wedge\ \mathtt{this.key} = \mathtt{key'}\ \ \wedge \mathtt{this.key} \neq \mathtt{a.key'}\ \}$

$\mathtt{this.key{:=}key''}$

$\{\langle\!\langle \mathtt{a.key} \rangle\!\rangle\ \ ||\ \ \{\langle\!\langle \mathtt{a.key} \rangle\!\rangle\}$

We can prove (65?) through application of Absurd, ProtNeq, and Consequ, as follows

(61c) $M_{fine} \vdash \{\ false\}$

$\mathtt{this.key{:=}key''}$

$\{\langle\!\langle \mathtt{a.key} \rangle\!\rangle\ \}\ \ ||\ \ \{\langle\!\langle \mathtt{a.key} \rangle\!\rangle\}$

By ProtNeq, we have $M_{fine} \vdash \langle\!\langle \mathtt{a.key} \rangle\!\rangle \not\leftrightarrow \mathtt{key'}\ \longrightarrow\ \mathtt{a.key} \neq \mathtt{key'}$, and therefore obtain

(61d) $M_{fine} \vdash \ldots \wedge\ \langle\!\langle \mathtt{a.key} \rangle\!\rangle \not\leftrightarrow \mathtt{Ids}_{set}\ \wedge\ \mathtt{this.key} = \mathtt{a.key}\ \wedge\ \mathtt{this.key} = \mathtt{key'}\ \longrightarrow\ false$

We apply Consequ on (61c) and (61d) and obtain (61aa?).

We can prove (66?) by proving that `this.key≠a.key` implies that `this` ≠ `a` (by the underlying Hoare logic), which again implies that the assignment `this.key := ...` leaves the value of `a.key` unmodified. We apply Prot-1, and are done.

□

H.4 Showing that M_{bad} does not satisfy S_2 nor S_3

H.4.1 M_{bad} does not satisfy S_2. M_{bad} does not satisfy S_2. We can argue this semantically (as in §H.4.2), and also in terms of the proof system (as in H.4.3).

H.4.2 $M_{bad} \nvDash S_2$. The reason is that M_{bad} exports the public method `set`, which updates the key without any checks. So, it could start in a state where the key of the account was protected, and then update it to something not protected.

In more detail: Take any state σ, where $M_{bad}, \sigma \models a_0 : \texttt{Account}, k_0 : \texttt{Key} \wedge \langle\!\langle a_0.\texttt{key} \rangle\!\rangle$. Assume also that $M_{bad}, \sigma \models \texttt{extl}$. Finally, assume that the continuation in σ consists of $a_0.\texttt{set}(k_0)$. Then we obtain that $M_{bad}, \sigma \leadsto^* \sigma'$, where $\sigma' = \sigma[a_0.\texttt{key} \mapsto k_0]$. We also have that $M_{bad}, \sigma' \models \texttt{extl}$, and because k_0 is a local variable, we also have that $M_{bad}, \sigma' \nvDash \langle\!\langle k_0 \rangle\!\rangle$. Moreover, $M_{bad}, \sigma' \models a_0.\texttt{krey} = k_0$. Therefore, $M_{bad}, \sigma' \nvDash \langle\!\langle a_0.\texttt{key} \rangle\!\rangle$.

H.4.3 $M_{bad} \nvdash S_2$. In order to prove that $M_{bad} \vdash S_2$, we would have needed to prove, among other things, that `set` satisfied S_2, which means proving that

$$
\begin{array}{ll}
\text{(ERR\_1?)} \quad M_{bad} \vdash \{\, \texttt{this} : \texttt{Account}, \texttt{k'} : \texttt{Key}, a : \texttt{Account} \wedge \langle\!\langle a.\texttt{key} \rangle\!\rangle \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow \{\texttt{this}, \texttt{k'}\} \,\} \\
\qquad\qquad\qquad \texttt{this.key:=k'}; \\
\qquad\qquad\qquad \texttt{res} := 0 \\
\qquad\qquad \{\, \langle\!\langle a.\texttt{key} \rangle\!\rangle \wedge \langle\!\langle a.\texttt{key} \rangle\!\rangle \not\leftrightarrow \texttt{res} \,\} \ \ || \ \ \{...\}
\end{array}
$$

However, we cannot establish (ERR_1?). Namely, when we take the case where $\texttt{this} = a$, we would need to establish, that

$$
\begin{array}{ll}
\text{(ERR\_2?)} \quad M_{bad} \vdash \{\, \texttt{this} : \texttt{Account}, \texttt{k'} : \texttt{Key} \wedge \langle\!\langle \texttt{this.key} \rangle\!\rangle \wedge \langle\!\langle \texttt{this.key} \rangle\!\rangle \not\leftrightarrow \{\texttt{this}, \texttt{k'}\} \,\} \\
\qquad\qquad\qquad \texttt{this.key:=k'} \\
\qquad\qquad \{\, \langle\!\langle \texttt{this.key} \rangle\!\rangle \,\} \ \ || \ \ \{...\}
\end{array}
$$

And there is no way to prove (ERR_2?). In fact, (ERR_2?) is not sound, for the reasons outlined in §H.4.2.

H.4.4 M_{bad} does not satisfy S_3. We have already argued in §?? that M_{bad} does not satisfy S_3, by giving a semantic argument – ie we are in state where $\langle\!\langle a_0.\texttt{key} \rangle\!\rangle$, and execute $\texttt{a}_0\texttt{.set(k1);a}_0\texttt{.transfer(...k1)}$.

Moroiever, if we attempted to prove that `set` satisfies S_3, we would have to show that

$$
\begin{array}{ll}
\text{(ERR\_3?)} \quad M_{bad} \vdash \{\, \texttt{this} : \texttt{Account}, \ \texttt{k'} : \texttt{Key}, a : \texttt{Account}, \ b : \texttt{int} \ \wedge \\
\qquad\qquad \langle\!\langle a.\texttt{key} \rangle\!\rangle \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow \{\texttt{this}, \texttt{k'}\} \wedge a.\texttt{blnce} \geq b \,\} \\
\qquad\qquad\qquad \texttt{this.key:=k'}; \\
\qquad\qquad\qquad \texttt{res} := 0 \\
\qquad\qquad \{\, \langle\!\langle a.\texttt{key} \rangle\!\rangle \wedge \langle\!\langle a.\texttt{key} \rangle\!\rangle \not\leftrightarrow \texttt{res} \wedge a.\texttt{blnce} \geq b \,\} \ \ || \ \ \{...\}
\end{array}
$$

which, in the case of $a = \texttt{this}$ would imply that

$$
\begin{array}{ll}
\text{(ERR\_4?)} \quad M_{bad} \vdash \{\, \texttt{this} : \texttt{Account}, \ \texttt{k'} : \texttt{Key}, \ b : \texttt{int} \ \wedge \\
\qquad\qquad \langle\!\langle \texttt{this.key} \rangle\!\rangle \wedge \langle\!\langle \texttt{this.key} \rangle\!\rangle \not\leftrightarrow \{\texttt{this}, \texttt{k'}\} \wedge \texttt{this.blnce} \geq b \,\} \\
\qquad\qquad\qquad \texttt{this.key:=k'} \\
\qquad\qquad \{\, \langle\!\langle \texttt{this.key} \rangle\!\rangle \,\} \ \ || \ \ \{...\}
\end{array}
$$

And (ERR_4?) cannot be proven and does not hold.

G.4 Showing that M_{bad} does not satisfy S_2 nor S_3

G.4.1 M_{bad} does not satisfy S_2. M_{bad} does not satisfy S_2. We can argue this semantically (as in §G.4.2), and also in terms of the proof system (as in G.4.3).

G.4.2 $M_{bad} \nvDash S_2$. The reason is that M_{bad} exports the public method `set`, which updates the key without any checks. So, it could start in a state where the key of the account was protected, and then update it to something not protected.

In more detail: Take any state σ, where $M_{bad}, \sigma \models a_0 : \texttt{Account}, k_0 : \texttt{Key} \wedge \langle\!\langle a_0.\texttt{key} \rangle\!\rangle$. Assume also that $M_{bad}, \sigma \models \texttt{extl}$. Finally, assume that the continuation in σ consists of $a_0.\texttt{set}(k_0)$. Then we obtain that $M_{bad}, \sigma \leadsto^* \sigma'$, where $\sigma' = \sigma[a_0.\texttt{key} \mapsto k_0]$. We also have that $M_{bad}, \sigma' \models \texttt{extl}$, and because k_0 is a local variable, we also have that $M_{bad}, \sigma' \nvDash \langle\!\langle k_0 \rangle\!\rangle$. Moreover, $M_{bad}, \sigma' \models a_0.\texttt{krey} = k_0$. Therefore, $M_{bad}, \sigma' \nvDash \langle\!\langle a_0.\texttt{key} \rangle\!\rangle$.

G.4.3 $M_{bad} \nvdash S_2$. In order to prove that $M_{bad} \vdash S_2$, we would have needed to prove, among other things, that `set` satisfied S_2, which means proving that

$$
\begin{array}{ll}
(\text{ERR\_1?}) & M_{bad} \ \vdash \{\, \texttt{this} : \texttt{Account}, \texttt{k'} : \texttt{Key}, a : \texttt{Account} \wedge \langle\!\langle a.\texttt{key} \rangle\!\rangle \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow \{\texttt{this}, \texttt{k'}\,\} \,\} \\
& \qquad\qquad\qquad \texttt{this.key:=k'}; \\
& \qquad\qquad\qquad \texttt{res} := 0 \\
& \qquad \{\, \langle\!\langle a.\texttt{key} \rangle\!\rangle \wedge \langle\!\langle a.\texttt{key} \rangle\!\rangle \not\leftrightarrow \texttt{res} \,\} \ \ || \ \ \{...\}
\end{array}
$$

However, we cannot establish (ERR_1?). Namely, when we take the case where $\texttt{this} = a$, we would need to establish, that

$$
\begin{array}{ll}
(\text{ERR\_2?}) & M_{bad} \ \vdash \{\, \texttt{this} : \texttt{Account}, \texttt{k'} : \texttt{Key} \wedge \langle\!\langle \texttt{this.key} \rangle\!\rangle \wedge \langle\!\langle \texttt{this.key} \rangle\!\rangle \not\leftrightarrow \{\texttt{this}, \texttt{k'}\,\} \,\} \\
& \qquad\qquad\qquad \texttt{this.key:=k'} \\
& \qquad \{\, \langle\!\langle \texttt{this.key} \rangle\!\rangle \,\} \ \ || \ \ \{...\}
\end{array}
$$

And there is no way to prove (ERR_2?). In fact, (ERR_2?) is not sound, for the reasons outlined in §G.4.2.

G.4.4 M_{bad} does not satisfy S_3. We have already argued in §?? that M_{bad} does not satisfy S_3, by giving a semantic argument – ie we are in state where $\langle\!\langle a_0.\texttt{key} \rangle\!\rangle$, and execute $\texttt{a}_0\texttt{.set(k1);a}_0\texttt{.transfer(...k1)}$.

Moroiever, if we attempted to prove that `set` satisfies S_3, we would have to show that

$$
\begin{array}{ll}
(\text{ERR\_3?}) & M_{bad} \ \vdash \{\, \texttt{this} : \texttt{Account}, \ \texttt{k'} : \texttt{Key}, a : \texttt{Account}, b : \texttt{int} \ \wedge \\
& \qquad \langle\!\langle a.\texttt{key} \rangle\!\rangle \wedge \langle\!\langle \texttt{a.key} \rangle\!\rangle \not\leftrightarrow \{\texttt{this}, \texttt{k'}\,\} \wedge a.\texttt{blnce} \geq b \,\} \\
& \qquad\qquad \texttt{this.key:=k'}; \\
& \qquad\qquad \texttt{res} := 0 \\
& \qquad \{\, \langle\!\langle a.\texttt{key} \rangle\!\rangle \wedge \langle\!\langle a.\texttt{key} \rangle\!\rangle \not\leftrightarrow \texttt{res} \wedge a.\texttt{blnce} \geq b \,\} \ \ || \ \ \{...\}
\end{array}
$$

which, in the case of $a = \texttt{this}$ would imply that

$$
\begin{array}{ll}
(\text{ERR\_4?}) & M_{bad} \ \vdash \{\, \texttt{this} : \texttt{Account}, \ \texttt{k'} : \texttt{Key}, b : \texttt{int} \ \wedge \\
& \qquad \langle\!\langle \texttt{this.key} \rangle\!\rangle \wedge \langle\!\langle \texttt{this.key} \rangle\!\rangle \not\leftrightarrow \{\texttt{this}, \texttt{k'}\,\} \wedge \texttt{this.blnce} \geq b \,\} \\
& \qquad\qquad \texttt{this.key:=k'} \\
& \qquad \{\, \langle\!\langle \texttt{this.key} \rangle\!\rangle \,\} \ \ || \ \ \{...\}
\end{array}
$$

And (ERR_4?) cannot be proven and does not hold.

H.5 Demonstrating that $M_{good} \vdash S_3$, and that $M_{fine} \vdash S_3$

H.6 Extending the specification S_3

As in §H.3, we redefine S_3 so that it also describes the behaviour of method `send`. and have:

$$S_{3,strong} \quad \triangleq \quad S_3 \ \wedge \ S_{2a} \ \wedge \ S_{2b}$$

Lemma H.7 (module M_{good} satisfies $S_{3,strong}$). $M_{good} \vdash S_{3,strong}$

Proof Outline In order to prove that

$$M_{good} \vdash \forall \mathtt{a}:\mathtt{Account}, b:\mathtt{int}.\{\ \langle\mathtt{a.key}\rangle \wedge \mathtt{a.blnce} \geq b\ \}$$

we have to apply Invariant from Fig. 8. That is, for each class C defined in M_{good}, and for each public method m in C, with parameters $\overline{y:D}$, we have to prove that they satisfy the corresponding quadruples.

Thus, we need to prove three Hoare quadruples: one for `Shop::buy`, one for `Account::transfer`, and one for `Account::set`. That is, we have to prove that

$$
\begin{array}{ll}
(31?) & M_{good} \ \vdash \{\ \mathtt{A}_{buy},\ \mathtt{a}:\mathtt{Account}, b:\mathtt{int}\ \wedge\ \langle\mathtt{a.key}\rangle\ \wedge\ \langle\mathtt{a.key}\rangle \not\leftrightarrow \mathtt{Ids}_{buy} \wedge \mathtt{a.blnce} \geq b\ \} \\
& \qquad\qquad\qquad \mathtt{Shop::buy.body} \\
& \qquad \{\langle\mathtt{a.key}\rangle\ \wedge\ \langle\mathtt{a.key}\rangle \not\!\triangledown \mathtt{res} \wedge \mathtt{a.blnce} \geq b\}\ \ ||\ \ \{\langle\mathtt{a.key}\rangle \wedge \mathtt{a.blnce} \geq b\} \\
(32?) & M_{good} \vdash \{\ \mathtt{A}_{trns},\ \mathtt{a}:\mathtt{Account}, b:\mathtt{int}\ \wedge\ \langle\mathtt{a.key}\rangle\ \wedge\ \langle\mathtt{a.key}\rangle \not\leftrightarrow \mathtt{Ids}_{trns}\ \wedge \mathtt{a.blnce} \geq b\ \} \\
& \qquad\qquad\qquad \mathtt{Account::transfer.body} \\
& \qquad \{\langle\mathtt{a.key}\rangle\ \wedge\ \langle\mathtt{a.key}\rangle \not\!\triangledown \mathtt{res} \wedge \mathtt{a.blnce} \geq b\ \}\ \ ||\ \ \{\langle\mathtt{a.key}\rangle \wedge \mathtt{a.blnce} \geq b\ \} \\
(33?) & M_{good} \vdash \{\ \mathtt{A}_{set},\ \mathtt{a}:\mathtt{Account}, b:\mathtt{int}\ \wedge\ \langle\mathtt{a.key}\rangle\ \wedge\ \langle\mathtt{a.key}\rangle \not\leftrightarrow \mathtt{Ids}_{set}\ \wedge \mathtt{a.blnce} \geq b\ \} \\
& \qquad\qquad\qquad \mathtt{Account::set.body} \\
& \qquad \{\langle\mathtt{a.key}\rangle \wedge\ \langle\mathtt{a.key}\rangle \not\!\triangledown \mathtt{res} \wedge \mathtt{a.blnce} \geq b\ \}\ \ ||\ \ \{\langle\mathtt{a.key}\rangle \wedge \mathtt{a.blnce} \geq b\ \}
\end{array}
$$

where we are using ? to indicate that this needs to be proven, and where we are using the shorthands $\mathtt{A}_{buy}$, $\mathtt{Ids}_{buy}$, $\mathtt{A}_{trns}$, $\mathtt{Ids}_{trns}$, $\mathtt{A}_{set}$ as defined earlier.

□

The proofs for M_{fine} are similar.

We outline the proof of (31?) in Lemma H.8. We outline the proof of (32?) in Lemma **??**.

H.6.1 Proving that `Shop::buy` from M_{good} satisfies $S_{3,strong}$ and also S_4.

Lemma H.8 (function M_{good} :: `Shop` :: `buy` satisfies $S_{3,strong}$ and also S_4).

$$
\begin{array}{ll}
(31) & M_{good} \ \vdash \{\ \mathtt{A}_{buy},\ \mathtt{a}:\mathtt{Account}, b:\mathtt{int}, \wedge \langle\mathtt{a.key}\rangle\ \wedge\ \langle\mathtt{a.key}\rangle \not\leftrightarrow \mathtt{Ids}_{buy} \wedge \mathtt{a.blnce} \geq b\ \} \\
& \qquad\qquad\qquad \mathtt{Shop::buy.body} \\
& \qquad \{\langle\mathtt{a.key}\rangle\ \wedge\ \langle\mathtt{a.key}\rangle \not\!\triangledown \mathtt{res} \wedge \mathtt{a.blnce} \geq b\}\ \ ||\ \ \{\langle\mathtt{a.key}\rangle \wedge \mathtt{a.blnce} \geq b\}
\end{array}
$$

Proof Outline Note that (31) is a proof that M_{good} :: `Shop` :: `buy` satisfies $S_{3,strong}$ and also hat M_{good} :: `Shop` :: `buy` satisfies S_4. This is so, because application of [Method] on S_4 gives us exactly the proof obligation from (31).

This proof is similar to the proof of lemma H.4, with the extra requirement here that we need to argue about balances not decreasing. To do this, we will leverage the assertion about balances given in S_3.

We will use the shorthand $\mathtt{A}_1 \triangleq \mathtt{A}_{buy}, \ \mathtt{a}:\mathtt{Account}, b:\mathtt{int}$. We will split the proof into 1) proving that statements 10, 11, 12 preserve the protection of `a.key` from the `buyer`, 2) proving that the external call

1st Step: proving statements 10, 11, 12

We apply the underlying Hoare logic and prove that the statements on lines 10, 11, 12 do not affect the value of `a.key` nor that of `a.blnce`. Therefore, for a $z, z' \notin \{\mathtt{price}, \mathtt{myAccnt}, \mathtt{oldBalance}\}$, we have

G.5 Demonstrating that $M_{good} \vdash S_3$, and that $M_{fine} \vdash S_3$

G.6 Extending the specification S_3

As in §G.3, we redefine S_3 so that it also describes the behaviour of method `send`. and have:

$$S_{3,strong} \quad \triangleq \quad S_3 \ \wedge \ S_{2a} \ \wedge \ S_{2b}$$

Lemma G.7 (module M_{good} satisfies $S_{3,strong}$)**.** $M_{good} \vdash S_{3,strong}$

Proof Outline In order to prove that

$$M_{good} \vdash \forall \mathtt{a} : \mathtt{Account}, b : \mathtt{int}.\{ \langle\mathtt{a.key}\rangle \wedge \mathtt{a.blnce} \geq b \}$$

we have to apply Invariant from Fig. 8. That is, for each class C defined in M_{good}, and for each public method m in C, with parameters $\overline{y : D}$, we have to prove that they satisfy the corresponding quadruples.

Thus, we need to prove three Hoare quadruples: one for `Shop::buy`, one for `Account::transfer`, and one for `Account::set`. That is, we have to prove that

$$\begin{array}{ll}
(31?) & M_{good} \ \vdash \ \{ \ \mathtt{A}_{buy}, \ \mathtt{a} : \mathtt{Account}, b : \mathtt{int} \ \wedge \ \langle\mathtt{a.key}\rangle \ \wedge \ \langle\mathtt{a.key}\rangle \not\leftrightarrow \mathtt{Ids}_{buy} \wedge \mathtt{a.blnce} \geq b \ \} \\
 & \qquad\qquad \mathtt{Shop} :: \mathtt{buy.body} \\
 & \quad \{\langle\mathtt{a.key}\rangle \ \wedge \ \langle\mathtt{a.key}\rangle \not\Leftarrow \mathtt{res} \wedge \mathtt{a.blnce} \geq b\} \ \ || \ \ \{\langle\mathtt{a.key}\rangle \wedge \mathtt{a.blnce} \geq b\} \\
(32?) & M_{good} \vdash \{ \ \mathtt{A}_{trns}, \ \mathtt{a} : \mathtt{Account}, b : \mathtt{int} \ \wedge \ \langle\mathtt{a.key}\rangle \ \wedge \ \langle\mathtt{a.key}\rangle \not\leftrightarrow \mathtt{Ids}_{trns} \ \wedge \mathtt{a.blnce} \geq b \ \} \\
 & \qquad\qquad \mathtt{Account} :: \mathtt{transfer.body} \\
 & \quad \{\langle\mathtt{a.key}\rangle \ \wedge \ \langle\mathtt{a.key}\rangle \not\Leftarrow \mathtt{res} \wedge \mathtt{a.blnce} \geq b \ \} \ \ || \ \ \{\langle\mathtt{a.key}\rangle \wedge \mathtt{a.blnce} \geq b \ \} \\
(33?) & M_{good} \vdash \{ \ \mathtt{A}_{set}, \ \mathtt{a} : \mathtt{Account}, b : \mathtt{int} \ \wedge \ \langle\mathtt{a.key}\rangle \ \wedge \ \langle\mathtt{a.key}\rangle \not\leftrightarrow \mathtt{Ids}_{set} \ \wedge \mathtt{a.blnce} \geq b \ \} \\
 & \qquad\qquad \mathtt{Account} :: \mathtt{set.body} \\
 & \quad \{\langle\mathtt{a.key}\rangle \wedge \ \langle\mathtt{a.key}\rangle \not\Leftarrow \mathtt{res} \wedge \mathtt{a.blnce} \geq b \ \} \ \ || \ \ \{\langle\mathtt{a.key}\rangle \wedge \mathtt{a.blnce} \geq b \ \}
\end{array}$$

where we are using ? to indicate that this needs to be proven, and where we are using the shorthands $\mathtt{A}_{buy}$, $\mathtt{Ids}_{buy}$, $\mathtt{A}_{trns}$, $\mathtt{Ids}_{trns}$, $\mathtt{A}_{set}$ as defined earlier.

□

The proofs for M_{fine} are similar.

We outline the proof of (31?) in Lemma G.8. We outline the proof of (32?) in Lemma **??**.

G.6.1 Proving that `Shop::buy` *from M_{good} satisfies $S_{3,strong}$ and also S_4.*

Lemma G.8 (function M_{good} :: `Shop` :: `buy` satisfies $S_{3,strong}$ and also S_4)**.**

$$\begin{array}{ll}
(31) & M_{good} \ \vdash \ \{ \ \mathtt{A}_{buy}, \ \mathtt{a} : \mathtt{Account}, b : \mathtt{int}, \wedge \langle\mathtt{a.key}\rangle \ \wedge \ \langle\mathtt{a.key}\rangle \not\leftrightarrow \mathtt{Ids}_{buy} \wedge \mathtt{a.blnce} \geq b \ \} \\
 & \qquad\qquad \mathtt{Shop} :: \mathtt{buy.body} \\
 & \quad \{\langle\mathtt{a.key}\rangle \ \wedge \ \langle\mathtt{a.key}\rangle \not\Leftarrow \mathtt{res} \wedge \mathtt{a.blnce} \geq b\} \ \ || \ \ \{\langle\mathtt{a.key}\rangle \wedge \mathtt{a.blnce} \geq b\}
\end{array}$$

Proof Outline Note that (31) is a proof that M_{good} :: `Shop` :: `buy` satisfies $S_{3,strong}$ and also hat M_{good} :: `Shop` :: `buy` satisfies S_4. This is so, because application of [Method] on S_4 gives us exactly the proof obligation from (31).

This proof is similar to the proof of lemma G.4, with the extra requirement here that we need to argue about balances not decreasing. To do this, we will leverage the assertion about balances given in S_3.

We will use the shorthand $\mathtt{A}_1 \triangleq \mathtt{A}_{buy}$, `a : Account`, b : `int`. We will split the proof into 1) proving that statements 10, 11, 12 preserve the protection of `a.key` from the `buyer`, 2) proving that the external call

1st Step: proving statements 10, 11, 12

We apply the underlying Hoare logic and prove that the statements on lines 10, 11, 12 do not affect the value of `a.key` nor that of `a.blnce`. Therefore, for a $z, z' \notin \{\mathtt{price}, \mathtt{myAccnt}, \mathtt{oldBalance}\}$, we have

$$(40)\quad M_{good} \vdash_{ul} \{\, \mathtt{A}_1 \ \wedge\ z = \mathtt{a.key} \wedge\ z' = \mathtt{a.blnce} \}$$
```
price:=anItem.price;
myAccnt:=this.accnt;
oldBalance := myAccnt.blnce;
```
$$\{ z = \mathtt{a.key} \wedge\ z' = \mathtt{a.blnce} \}$$

We then apply Embed_UL, Prot-1 and Prot-2 and Combine and and Types-2 on (10) and use the shorthand $\mathtt{stmts}_{10,11,12}$ for the statements on lines 10, 11 and 12, and obtain:

$$(41)\quad M_{good} \vdash \{\, \mathtt{A}_1 \ \wedge\ \langle\mathtt{a.key}\rangle \ \wedge\ \langle\mathtt{buyer}\rangle \not\leftrightarrow \mathtt{a.key} \wedge\ z' = \mathtt{a.blnce} \}$$
$$\mathtt{stmts}_{10,11,12}$$
$$\{\, \langle\mathtt{a.key}\rangle \ \wedge\ \langle\mathtt{buyer}\rangle \not\leftrightarrow \mathtt{a.key} \wedge\ z' = \mathtt{a.blnce} \}$$

We apply Mid on (11) and obtain

$$(42)\quad M_{good} \vdash \{\, \mathtt{A}_1 \ \wedge\ \langle\mathtt{a.key}\rangle \not\leftrightarrow \mathtt{buyer} \ \wedge\ z' = \mathtt{a.blnce} \}$$
$$\mathtt{stmts}_{10,11,12}$$
$$\{\, \mathtt{A}_1 \ \wedge\ \langle\mathtt{a.key}\rangle \ \wedge\ \langle\mathtt{buyer}\rangle \not\leftrightarrow \mathtt{a.key} \ \wedge\ z' = \mathtt{a.blnce} \} \ ||$$
$$\{\, \langle\mathtt{a.key}\rangle \ \wedge\ z' = \mathtt{a.blnce} \}$$

2nd Step: Proving the External Call

We now need to prove that the external method call `buyer.pay(this.accnt, price)` protects the `key`, and does nit decrease the balance, i.e.

$$(43?)\quad M_{good} \vdash \{\, \mathtt{A}_1 \ \wedge\ \langle\mathtt{a.key}\rangle \ \wedge\ \langle\mathtt{a.key}\rangle \not\leftrightarrow \mathtt{buyer} \wedge\ z' = \mathtt{a.blnce} \}$$
$$\mathtt{tmp\ :=\ buyer.pay(myAccnt,\ price)}$$
$$\{\ \ \mathtt{A}_1 \ \wedge\ \langle\mathtt{a.key}\rangle \ \wedge\ \langle\mathtt{buyer}\rangle \not\leftrightarrow \mathtt{a.key} \wedge\ \mathtt{a.blnce} \geq z' \ \} \ \ ||$$
$$\{\, \langle\mathtt{a.key}\rangle \ \wedge\ \mathtt{a.blnce} \geq z' \}$$

We use that $M \vdash \mathbb{V}\mathtt{a} : \mathtt{Account}, \mathtt{b} : \mathtt{int},.\{\langle\mathtt{a.key}\rangle \wedge \mathtt{a.blnce} \geq z'\}$ and obtain

$$(44)\quad M_{good} \vdash \{\, \mathtt{buyer} : \mathtt{external},\ \langle\mathtt{a.key}\rangle \ \wedge\ \langle\mathtt{a.key}\rangle \not\leftrightarrow (\mathtt{buyer}, \mathtt{myAccnt}, \mathtt{price}) \ \wedge\ z' \geq \mathtt{a.blnce} \}$$
$$\mathtt{tmp\ :=\ buyer.pay(myAccnt,\ price)}$$
$$\{\, \langle\mathtt{a.key}\rangle \ \wedge\ \langle\mathtt{a.key}\rangle \not\leftrightarrow (\mathtt{buyer}, \mathtt{myAccnt}, \mathtt{price}) \ \wedge\ z' \geq \mathtt{a.blnce} \} \ \ ||$$
$$\{\, \langle\mathtt{a.key}\rangle \ \wedge\ z' \geq \mathtt{a.blnce} \}$$

In order to obtain (43?) out of (44), we apply Prot-Intl and Prot-Int_1, which gives us

$$(45)\qquad M_{good} \vdash \mathtt{A}_1 \wedge \langle\mathtt{a.key}\rangle \ \longrightarrow\ \langle\mathtt{a.key}\rangle \not\leftrightarrow \mathtt{myAccnt}$$
$$(46)\qquad M_{good} \vdash \mathtt{A}_1 \wedge \langle\mathtt{a.key}\rangle \ \longrightarrow\ \langle\mathtt{a.key}\rangle \not\leftrightarrow \mathtt{price}$$
$$(47)\qquad M_{good} \vdash \mathtt{A}_1 \wedge z' = \mathtt{a.blnce} \ \longrightarrow\ z' \geq \mathtt{a.blnce}$$

We apply Consequ on (44), (45), (46) and (47) and obtain (43)!

3nd Step: Proving the Remainder of the Body

We now need to prove that lines 15-19 of the method preserve the protection of `a.key`, and do not decrease `a.balance`. We outline the remaining proof in less detail.

We prove the internal call on line 16, using the method specification for `send`, using S_{2a} and S_{2b}, and applying rule [Call_Int], and obtain

$$
\begin{array}{ll}
(40) & M_{good} \vdash_{ul} \{\, \texttt{A}_1 \ \wedge\ z = \texttt{a.key} \wedge\ z' = \texttt{a.blnce} \} \\
& \qquad\qquad \texttt{price:=anItem.price;} \\
& \qquad\qquad \texttt{myAccnt:=this.accnt;} \\
& \qquad\qquad \texttt{oldBalance := myAccnt.blnce;} \\
& \qquad \{ z = \texttt{a.key} \wedge z' = \texttt{a.blnce} \}
\end{array}
$$

We then apply Embed_UL, Prot-1 and Prot-2 and Combine and and Types-2 on (10) and use the shorthand $\texttt{stmts}_{10,11,12}$ for the statements on lines 10, 11 and 12, and obtain:

$$
\begin{array}{ll}
(41) & M_{good} \vdash \{\, \texttt{A}_1 \ \wedge\ \langle \texttt{a.key} \rangle \ \wedge\ \langle \texttt{buyer} \rangle \not\leftrightarrow \texttt{a.key} \wedge\ z' = \texttt{a.blnce} \} \\
& \qquad\qquad \texttt{stmts}_{10,11,12} \\
& \qquad \{\, \langle \texttt{a.key} \rangle \ \wedge\ \langle \texttt{buyer} \rangle \not\leftrightarrow \texttt{a.key} \wedge\ z' = \texttt{a.blnce} \}
\end{array}
$$

We apply Mid on (11) and obtain

$$
\begin{array}{ll}
(42) & M_{good} \vdash \{\, \texttt{A}_1 \ \wedge\ \langle \texttt{a.key} \rangle \not\leftrightarrow \texttt{buyer} \ \wedge\ z' = \texttt{a.blnce} \} \\
& \qquad\qquad \texttt{stmts}_{10,11,12} \\
& \qquad \{\, \texttt{A}_1 \ \wedge\ \langle \texttt{a.key} \rangle \ \wedge\ \langle \texttt{buyer} \rangle \not\leftrightarrow \texttt{a.key} \ \wedge\ z' = \texttt{a.blnce} \} \ \parallel \\
& \qquad \{\, \langle \texttt{a.key} \rangle \ \wedge\ z' = \texttt{a.blnce} \}
\end{array}
$$

2nd Step: Proving the External Call

We now need to prove that the external method call `buyer.pay(this.accnt, price)` protects the `key`, and does nit decrease the balance, i.e.

$$
\begin{array}{ll}
(43?) & M_{good} \vdash \{\, \texttt{A}_1 \ \wedge\ \langle \texttt{a.key} \rangle \ \wedge\ \langle \texttt{a.key} \rangle \not\leftrightarrow \texttt{buyer} \wedge\ z' = \texttt{a.blnce} \} \\
& \qquad\qquad \texttt{tmp := buyer.pay(myAccnt, price)} \\
& \qquad \{\ \ \texttt{A}_1 \ \wedge\ \langle \texttt{a.key} \rangle \ \wedge\ \langle \texttt{buyer} \rangle \not\leftrightarrow \texttt{a.key} \wedge\ \texttt{a.blnce} \geq z' \ \} \ \parallel \\
& \qquad \{\, \langle \texttt{a.key} \rangle \ \wedge\ \texttt{a.blnce} \geq z' \}
\end{array}
$$

We use that $M \vdash \forall \texttt{a} : \texttt{Account}, \texttt{b} : \texttt{int},.\{ \langle \texttt{a.key} \rangle \wedge \texttt{a.blnce} \geq z' \}$ and obtain

$$
\begin{array}{ll}
(44) & M_{good} \vdash \{\, \texttt{buyer} : \texttt{external},\ \langle \texttt{a.key} \rangle \ \wedge\ \langle \texttt{a.key} \rangle \not\leftrightarrow (\texttt{buyer}, \texttt{myAccnt}, \texttt{price}) \ \wedge\ z' \geq \texttt{a.blnce} \} \\
& \qquad\qquad \texttt{tmp := buyer.pay(myAccnt, price)} \\
& \qquad \{\, \langle \texttt{a.key} \rangle \ \wedge\ \langle \texttt{a.key} \rangle \not\leftrightarrow (\texttt{buyer}, \texttt{myAccnt}, \texttt{price}) \ \wedge\ z' \geq \texttt{a.blnce} \} \ \parallel \\
& \qquad \{\, \langle \texttt{a.key} \rangle \ \wedge\ z' \geq \texttt{a.blnce} \}
\end{array}
$$

In order to obtain (43?) out of (44), we apply Prot-Intl and Prot-Int_1, which gives us

$$
\begin{array}{ll}
(45) & M_{good} \vdash \texttt{A}_1 \wedge \langle \texttt{a.key} \rangle \ \longrightarrow\ \langle \texttt{a.key} \rangle \not\leftrightarrow \texttt{myAccnt} \\
(46) & M_{good} \vdash \texttt{A}_1 \wedge \langle \texttt{a.key} \rangle \ \longrightarrow\ \langle \texttt{a.key} \rangle \not\leftrightarrow \texttt{price} \\
(47) & M_{good} \vdash \texttt{A}_1 \wedge z' = \texttt{a.blnce} \ \longrightarrow\ z' \geq \texttt{a.blnce}
\end{array}
$$

We apply Consequ on (44), (45), (46) and (47) and obtain (43)!

3nd Step: Proving the Remainder of the Body

We now need to prove that lines 15-19 of the method preserve the protection of `a.key`, and do not decrease `a.balance`. We outline the remaining proof in less detail.

We prove the internal call on line 16, using the method specification for `send`, using S_{2a} and S_{2b}, and applying rule [Call_Int], and obtain

(48) $M_{good} \vdash \{\,$`buyer : external, item : Intem` $\wedge$ ⟪`a.key`⟫ $\wedge$ ⟪`a.key`⟫$\not\leftrightarrow$ (`buyer` $\wedge\ z' =$ `a.blnce`$\}$
`tmp := this.send(buyer,Item)`
$\{$ ⟪`a.key`⟫ $\wedge$ ⟪`a.key`⟫$\not\leftrightarrow$ `buyer` $\wedge\ z' =$ `a.blnce`$\}$ $\|$
$\{$ ⟪`a.key`⟫ $\wedge\ z' =$ `a.blnce`$\}$

We now need to prove that the external method call `buyer.tell("You have not paid me")` also protects the `key`, and does nit decrease the balance. We can do this by applying the rule about protection from strings [Pror_Str], the fact that $M_{good} \vdash S_3$, and rule [Call_Extl_Adapt] and obtain:

(49) $M_{good} \vdash \{\,$`buyer : external, item : Intem` $\wedge$ ⟪`a.key`⟫ $\wedge$ ⟪`a.key`⟫$\not\leftrightarrow$ (`buyer` $\wedge\ z'/geq$`a.blnce`$\}$
`tmp:=buyer.tell("You have not paid me")`
$\{$ ⟪`a.key`⟫ $\wedge$ ⟪`a.key`⟫$\not\leftrightarrow$ `buyer` $\wedge\ z' \geq$ `a.blnce`$\}$ $\|$
$\{$ ⟪`a.key`⟫ $\wedge\ z' \geq$ `a.blnce`$\}$

We can now apply [If_Rule, and [Conseq on (49) and (50), and obtain

(50) $M_{good} \vdash \{\,$`buyer : external, item : Intem` $\wedge$ ⟪`a.key`⟫ $\wedge$ ⟪`a.key`⟫$\not\leftrightarrow$ (`buyer` $\wedge\ z' \geq$ `a.blnce`$\}$
```
if...then
 tmp:=this.send(buyer,anItem)
else
 tmp:=buyer.tell("You have not paid me")
```
$\{$ ⟪`a.key`⟫ $\wedge$ ⟪`a.key`⟫$\not\leftrightarrow$ `buyer` $\wedge\ z' \geq$ `a.blnce`$\}$ $\|$
$\{$ ⟪`a.key`⟫ $\wedge\ z' \geq$ `a.blnce`$\}$

The rest follows through application of [Prot_Int, and [Seq].

□

Lemma H.9 (function M_{good} :: `Account` :: `transfer` satisfies S_3).

(32) $M_{good} \vdash \{\, A_{trns}$, `a : Account`, b : `int` $\wedge$ ⟪`a.key`⟫ $\wedge$ ⟪`a.key`⟫$\not\leftrightarrow$ `Ids`$_{trns}$ $\wedge$ `a.blnce` $\geq b\,\}$
`Account :: transfer.body`
$\{$⟪`a.key`⟫ $\wedge$ ⟪`a.key`⟫$\not\nabla$`res` $\wedge$ `a.blnce` $\geq b\,\}$ $\|$ $\{$⟪`a.key`⟫ $\wedge$ `a.blnce` $\geq b\,\}$

Proof Outline We will use the shorthand $stmts_{28-33}$ for the statements in the body of `transfer`. We will prove the preservation of protection, separately from the balance not decreasing when the key is protcted.
For the former, applying the steps in the proof of Lemma H.5, we obtain

(21) $M_{good} \vdash \{\, A_{trns}$, `a : Account` $\wedge$ ⟪`a.key`⟫ $\wedge$ ⟪`a.key`⟫$\not\leftrightarrow$ `Ids`$_{trns}\,\}$
$stmts_{28-33}$
$\{$⟪`a.key`⟫ $\wedge$ ⟪`a.key`⟫$\not\nabla$`res`$\}$ $\|$ $\{$⟪`a.key`⟫$\}$

For the latter, we rely on the underlying Hoare logic to ensure that no balance decreases, except perhaps that of the receiver, in which case its key was not protected. Namely, we have that

$$(48)\quad M_{good}\ \vdash \{\, \texttt{buyer}:\texttt{external},\ \texttt{item}:\texttt{Intem} \wedge \langle\texttt{a.key}\rangle \wedge \langle\texttt{a.key}\rangle\not\leftrightarrow (\texttt{buyer} \wedge z' = \texttt{a.blnce}\}$$
$$\texttt{tmp := this.send(buyer,Item)}$$
$$\{\, \langle\texttt{a.key}\rangle \wedge \langle\texttt{a.key}\rangle\not\leftrightarrow \texttt{buyer} \wedge z' = \texttt{a.blnce}\} \ \ ||$$
$$\{\, \langle\texttt{a.key}\rangle \wedge z' = \texttt{a.blnce}\}$$

We now need to prove that the external method call `buyer.tell("You have not paid me")` also protects the `key`, and does nit decrease the balance. We can do this by applying the rule about protection from strings [Pror_Str], the fact that $M_{good} \vdash S_3$, and rule [Call_Extl_Adapt] and obtain:

$$(49)\quad M_{good}\ \vdash \{\, \texttt{buyer}:\texttt{external},\ \texttt{item}:\texttt{Intem} \wedge \langle\texttt{a.key}\rangle \wedge \langle\texttt{a.key}\rangle\not\leftrightarrow (\texttt{buyer} \wedge z'/geq\texttt{a.blnce}\}$$
$$\texttt{tmp:=buyer.tell("You have not paid me")}$$
$$\{\, \langle\texttt{a.key}\rangle \wedge \langle\texttt{a.key}\rangle\not\leftrightarrow \texttt{buyer} \wedge z' \geq \texttt{a.blnce}\} \ \ ||$$
$$\{\, \langle\texttt{a.key}\rangle \wedge z' \geq \texttt{a.blnce}\}$$

We can now apply [If_Rule, and [Conseq on (49) and (50), and obtain

$$(50)\quad M_{good}\ \vdash \{\, \texttt{buyer}:\texttt{external},\ \texttt{item}:\texttt{Intem} \wedge \langle\texttt{a.key}\rangle \wedge \langle\texttt{a.key}\rangle\not\leftrightarrow (\texttt{buyer} \wedge z' \geq \texttt{a.blnce}\}$$

```
if...then
 tmp:=this.send(buyer,anItem)
else
 tmp:=buyer.tell("You have not paid me")
```

$$\{\, \langle\texttt{a.key}\rangle \wedge \langle\texttt{a.key}\rangle\not\leftrightarrow \texttt{buyer} \wedge z' \geq \texttt{a.blnce}\} \ \ ||$$
$$\{\, \langle\texttt{a.key}\rangle \wedge z' \geq \texttt{a.blnce}\}$$

The rest follows through application of [Prot_Int, and [Seq].

□

Lemma G.9 (function M_{good} :: `Account` :: `transfer` satisfies S_3)**.**

$$(32)\quad M_{good} \vdash \{\, \texttt{A}_{trns},\ \texttt{a}:\texttt{Account}, b:\texttt{int} \wedge \langle\texttt{a.key}\rangle \wedge \langle\texttt{a.key}\rangle\not\leftrightarrow \texttt{Ids}_{trns} \wedge \texttt{a.blnce} \geq b \,\}$$
$$\texttt{Account :: transfer.body}$$
$$\{\langle\texttt{a.key}\rangle \wedge \langle\texttt{a.key}\rangle\text{-}\!\nabla\,\texttt{res} \wedge \texttt{a.blnce} \geq b\,\} \ \ || \ \ \{\langle\texttt{a.key}\rangle \wedge \texttt{a.blnce} \geq b\,\}$$

Proof Outline We will use the shorthand $stmts_{28-33}$ for the statements in the body of `transfer`. We will prove the preservation of protection, separately from the balance not decreasing when the key is protcted.

For the former, applying the steps in the proof of Lemma G.5, we obtain

$$(21)\quad M_{good} \vdash \{\, \texttt{A}_{trns},\ \texttt{a}:\texttt{Account} \wedge \langle\texttt{a.key}\rangle \wedge \langle\texttt{a.key}\rangle\not\leftrightarrow \texttt{Ids}_{trns}\,\}$$
$$stmts_{28-33}$$
$$\{\langle\texttt{a.key}\rangle \wedge \langle\texttt{a.key}\rangle\text{-}\!\nabla\,\texttt{res}\} \ \ || \ \ \{\langle\texttt{a.key}\rangle\}$$

For the latter, we rely on the underlying Hoare logic to ensure that no balance decreases, except perhaps that of the receiver, in which case its key was not protected. Namely, we have that

$$(71) \quad M_{good} \vdash_u l \{ \mathtt{A}_{trns},\ \mathtt{a}:\mathtt{Account} \wedge \mathtt{a.blnce} = b \ \wedge\ (\mathtt{this} \neq \mathtt{a} \vee prgthis.\mathtt{key} \neq \mathtt{key}') \}$$
$$stmts_{28-33}$$
$$\{\mathtt{a.blnce} \geq b\}$$

We apply rules EMBED_UL and MID on (71), and obtain

$$(72) \quad M_{good} \vdash \{ \mathtt{A}_{trns},\ \mathtt{a}:\mathtt{Account} \wedge \mathtt{a.blnce} = b \ \wedge\ (\mathtt{this} \neq \mathtt{a} \vee prgthis.\mathtt{key} \neq \mathtt{key}') \}$$
$$stmts_{28-33}$$
$$\{\mathtt{a.blnce} \geq b\} \ \ || \ \ \{\mathtt{a.blnce} \geq b\}$$

Moreover, we have

$$
\begin{array}{llll}
(73) & M_{good} & \vdash & \langle\mathtt{a.key}\rangle\not\leftrightarrow \mathtt{Ids}_{trns} \ \ \rightarrow\ \ \langle\mathtt{a.key}\rangle\not\leftrightarrow \mathtt{key}' \\
(74) & M_{good} & \vdash & \langle\mathtt{a.key}\rangle\not\leftrightarrow \mathtt{key}' \ \ \rightarrow\ \ \mathtt{a.key} \neq \mathtt{key}' \\
(75) & M_{good} & \vdash & \mathtt{a.key} \neq \mathtt{key}' \ \ \rightarrow\ \ \mathtt{a} \neq \mathtt{this} \vee \mathtt{this.key} \neq \mathtt{key}'
\end{array}
$$

normalsize

Applying (73), (74), (75) and CONSEQ on (72) we obtain:

$$(76) \quad M_{good} \vdash \{ \mathtt{A}_{trns},\ \mathtt{a}:\mathtt{Account} \wedge \mathtt{a.blnce} = b \ \wedge\ \langle\mathtt{a.key}\rangle\not\leftrightarrow \mathtt{Ids}_{trns} \ \}$$
$$stmts_{28-33}$$
$$\{\mathtt{a.blnce} \geq b\} \ \ || \ \ \{\mathtt{a.blnce} \geq b\}$$

We combine (72) and (76) through COMBINE and obtain (32).

□

H.7 Dealing with polymorphic function calls

The case split rules together with the rule of consequence allow our Hoare logic to formally reason about polymorphic calls, where the receiver may be internal or external.

We demonstrate this through an example where we may have an external receiver, or a receiver from a class C. Assume we had a module M with a scoped invariant (as in A), and an internal method specification as in (B).

$$
\begin{array}{llll}
(A) & M & \vdash & \forall y_1 : D.\{A\} \\
(B) & M & \vdash & \{ A_1 \}\, \mathtt{private}\ C::m(y_1 : D)\, \{ A_2 \} || \{A_3\}
\end{array}
$$

Assume also implications as in (C)-(H)

$$
\begin{array}{llll}
(C) & M & \vdash & A_0 \ \rightarrow\ A\text{-}\nabla(y_0, y_1) \\
(D) & M & \vdash & A\text{-}\nabla(y_0, y_1) \rightarrow A_4 \\
(E) & M & \vdash & A \rightarrow A_5 \\
(F) & M & \vdash & A_0 \rightarrow A_1[y_0/\mathtt{this}] \\
(G) & M & \vdash & A_2[y_0, u/\mathtt{this}, res] \rightarrow A_4 \\
(H) & M & \vdash & A_3 \rightarrow A_5
\end{array}
$$

Then, by application of CALL_EXT_ADAPT on (A) we obtain (I)

$$(I) \ \ M \vdash \{ \ y_0 : external, y_1 : D \wedge A\text{-}\nabla(y_0, y_1) \ \} \, u := y_0.m(y_1) \, \{ \ A\text{-}\nabla(y_0, y_1) \ \} \ \ || \ \ \{ \ A \ \}$$

By application of the rule of consequence on (I) and (C), (D), and (E), we obtain

$$(J) \ \ M \vdash \{ \ y_0 : external, y_1 : D \wedge A_0 \ \} \, u := y_0.m(y_1) \, \{ \ A_4 \ \} \ \ || \ \ \{ \ A_5 \ \}$$

Then, by application of [CALL_INTL] on (B) we obtain (K)

$$(K) \ \ M \vdash \{ \ y_0 : C, y_1 : D \wedge A_1[y_0/\mathtt{this}] \ \} \, u := y_0.m(y_1) \, \{ \ A_2[y_0, u/\mathtt{this}, res] \ \} \ \ || \ \ \{ \ A_3 \ \}$$

By application of the rule of consequence on (K) and (F), (G), and (H), we obtain

(71) $M_{good} \vdash_u l \{ \mathtt{A}_{trns},\ \mathtt{a:Account} \wedge \mathtt{a.blnce} = b\ \wedge\ (\mathtt{this} \neq \mathtt{a} \vee prgthis.\mathtt{key} \neq \mathtt{key'}) \}$

$stmts_{28-33}$

$\{\mathtt{a.blnce} \geq b\}$

We apply rules Embed_UL and Mid on (71), and obtain

(72) $M_{good} \vdash \{ \mathtt{A}_{trns},\ \mathtt{a:Account} \wedge \mathtt{a.blnce} = b\ \wedge\ (\mathtt{this} \neq \mathtt{a} \vee prgthis.\mathtt{key} \neq \mathtt{key'}) \}$

$stmts_{28-33}$

$\{\mathtt{a.blnce} \geq b\}\ \ ||\ \ \{\mathtt{a.blnce} \geq b\}$

Moreover, we have

(73) $M_{good} \vdash \langle\mathtt{a.key}\rangle\not\leftrightarrow \mathtt{Ids}_{trns} \ \rightarrow\ \langle\mathtt{a.key}\rangle\not\leftrightarrow \mathtt{key'}$

(74) $M_{good} \vdash \langle\mathtt{a.key}\rangle\not\leftrightarrow \mathtt{key'} \ \rightarrow\ \mathtt{a.key} \neq \mathtt{key'}$

(75) $M_{good} \vdash \mathtt{a.key} \neq \mathtt{key'} \ \rightarrow\ \mathtt{a} \neq \mathtt{this} \vee \mathtt{this.key} \neq \mathtt{key'}$

normalsize

Applying (73), (74), (75) and Conseq on (72) we obtain:

(76) $M_{good} \vdash \{ \mathtt{A}_{trns},\ \mathtt{a:Account} \wedge \mathtt{a.blnce} = b\ \wedge\ \langle\mathtt{a.key}\rangle\not\leftrightarrow \mathtt{Ids}_{trns}\ \}$

$stmts_{28-33}$

$\{\mathtt{a.blnce} \geq b\}\ \ ||\ \ \{\mathtt{a.blnce} \geq b\}$

We combine (72) and (76) through Combine and obtain (32).

□

G.7 Dealing with polymorphic function calls

The case split rules together with the rule of consequence allow our Hoare logic to formally reason about polymorphic calls, where the receiver may be internal or external.

We demonstrate this through an example where we may have an external receiver, or a receiver from a class C. Assume we had a module M with a scoped invariant (as in A), and an internal method specification as in (B).

$(A)\quad M \vdash \forall y_1 : D.\{A\}$

$(B)\quad M \vdash \{ A_1 \}\, \mathtt{private}\ C :: m(y_1 : D) \{ A_2 \} || \{A_3\}$

Assume also implications as in (C)-(H)

$(C)\quad M \vdash A_0 \rightarrow A\text{-}\nabla(y_0, y_1)$

$(D)\quad M \vdash A\text{-}\nabla(y_0, y_1) \rightarrow A_4$

$(E)\quad M \vdash A \rightarrow A_5$

$(F)\quad M \vdash A_0 \rightarrow A_1[y_0/\mathtt{this}]$

$(G)\quad M \vdash A_2[y_0, u/\mathtt{this}, res] \rightarrow A_4$

$(H)\quad M \vdash A_3 \rightarrow A_5$

Then, by application of Call_Ext_Adapt on (A) we obtain (I)

$(I)\ \ M \vdash \{\ y_0 : external, y_1 : D \wedge A\text{-}\nabla(y_0, y_1)\ \}\ u := y_0.m(y_1)\ \{\ A\text{-}\nabla(y_0, y_1)\ \}\ \ ||\ \ \{\ A\ \}$

By application of the rule of consequence on (I) and (C), (D), and (E), we obtain

$(J)\ \ M \vdash \{\ y_0 : external, y_1 : D \wedge A_0\ \}\ u := y_0.m(y_1)\ \{\ A_4\ \}\ \ ||\ \ \{\ A_5\ \}$

Then, by application of [Call_Intl] on (B) we obtain (K)

$(K)\ \ M \vdash \{\ y_0 : C, y_1 : D \wedge A_1[y_0/\mathtt{this}]\ \}\ u := y_0.m(y_1)\ \{\ A_2[y_0, u/\mathtt{this}, res]\ \}\ \ ||\ \ \{\ A_3\ \}$

By application of the rule of consequence on (K) and (F), (G), and (H), we obtain

$(L)\ \ M \vdash \{\ y_0 : C, y_1 : D \wedge A_0\ \}\ u := y_0.m(y_1)\ \{\ A_4\ \}\ \ \|\ \ \{\ A_5\ \}$

By case split, [Cases], on (J) and (L), we obtain

$(polymoprhic)\ \ M \vdash \{\ (y_0 : external \vee y_0 : C), y_1 : D \wedge A_0\ \}\ u := y_0.m(y_1)\ \{\ A_4\ \}\ \ \|\ \ \{\ A_5\ \}$

$$(L) \;\; M \vdash \{\; y_0 : C, y_1 : D \wedge A_0 \;\}\; u := y_0.m(y_1) \;\{\; A_4 \;\}\;\; \| \;\; \{\; A_5 \;\}$$

By case split, [Cases], on (J) and (L), we obtain

$$(polymoprhic) \;\; M \vdash \{\; (y_0 : external \vee y_0 : C), y_1 : D \wedge A_0 \;\}\; u := y_0.m(y_1) \;\{\; A_4 \;\}\;\; \| \;\; \{\; A_5 \;\}$$