Foundations for Reasoning about Holistic Specifications

Duc Than Nguyen

Supervisor: A/Prof. Toby Murray
A/Prof. Ben Rubinstein

Melbourne School of Engineering
The University of Melbourne

This thesis is submitted for the degree of
Master of Philosophy

School of Computing and
Information Systems

January 2021

Declaration

This is to certify that

- the thesis comprises only my original work towards the Master of Philosophy,
- due acknowledgment has been made in the text to all other material used,
- the thesis is less than 50,000 words in length, exclusive of tables, maps, bibliographies, and appendices.

Duc Than Nguyen
January 2021

Abstract

Specifications of sufficient conditions may be enough for reasoning about complete and unchanging programs of a closed system. Nevertheless, there is no luxury of trusting external components of probably unknown provenance in an open world that may be buggy or potentially malicious. It is critical to ensure that our components are robust when cooperating with a wide variety of external components. Holistic specifications, which are concerned with *sufficient* and *necessary* conditions, could make programs more robust in an open-world setting.

In this thesis, we lay the foundations for reasoning about holistic specifications. We give an Isabelle/HOL mechanization of holistic specifications focusing on object-based programs. We also pave a way to reason about holistic specifications via proving some key lemmas that we hope will be useful in the future to establish a general logic for holistic specifications.

Contents

List of Figures

Chapter 1

Introduction

Traditional system designs are often implicitly based on a closed world assumption where a component can trust to interact with other component's operations: a client who supplies arguments meeting that operation's pre-conditions can invoke it and obtain the associated effect. Nevertheless, a system could be more complex, buggy, or potentially malicious when it collaborates with a wide range of external components. Such a system is considered to deal with open-world settings. Indeed, in an open world, we do wish to trust the other components with which we collaborate. Trusting our personal information to software that operates in an open world might make us vulnerable to hackers and exploits. As a result, there is a need for our software to be robust. We expect that software to perform correctly, even if used by external parties of probably unknown provenance, buggy, or potentially malicious. For instance, medical patients expect their health data not to be sent to their employer(s) unless they authorized the release.

There are numerous studies on the specification and verification of programs' functional correctness [15, 29, 19, 13, 1]. Most of these methods based on pre- and post-conditions are rooted in design-by-contract assumptions: "If the pre-condition is not satisfied, the routine is not bound to do anything" [19]. Such specifications describe what sufficient conditions are for some effect to happen. This approach is enough to reason about complete, unchanging programs of closed systems. However, in an open world, things are more complicated; systems must deal with a range of external components that might not be under our control. Since methods of external components cannot control when they are invoked, we must work as if all externally visible methods have the pre-condition `true`. Sufficient conditions are not adequate in open-world settings; therefore, there is a need to have the necessary conditions to ensure that bad things will not happen. To do that, Drossopoulou et al. [8] first introduced holistic specifications, which are kinds of specifications dealing with sufficient conditions as well as necessary conditions. They, namely, proposed a specification language, named *Chainmail*, to present holistic specifications. In addition to traditional specification languages, the design of Chainmail draws concepts from object capabilities [21], temporal logic, and spatial connection. Thanks to these features, Chainmail can express holistic specifications for several examples not only from object-capability programs but also the smart contracts applications such as Bank/Account example [22] and DAO (Decentralised Autonomous Organisation) [9], a famous Ethereum contract aiming to provide smart contracts, managed by the DAO owners and not affected by a central government.

Although a formal specification is more precise than a natural language, it could disagree or contain errors with requirements. Verified formal specifications, in this case, are needed when we desire to be sure that the specification follows its requirements. With the same goal, we want to have a verified formal specification framework for *Chainmail*. We choose Isabelle [26], a higher-order logic (HOL) theorem prover, to give *Chainmail* a verified formal specification. Therefore, our primary goal is to lay the foundations for reasoning about holistic specifications. To have a skeleton for reasoning about holistic specifications, we give a formalization of holistic specifications focusing on object-based programs Isabelle in this thesis. Mainly, we provide an Isabelle mechanization of *Chainmail* with several definitions, theorems, and technical lemmas. Moreover, to pave a way to reason about holistic specifications, we prove some key lemmas that we hope will help establish a general logic for holistic specifications.

The rest of this thesis is structured as follows. We present the background of Isabelle/HOL, holistic specifications, and survey related work in Chapter 2. Then, we describe our main contribution to Chapter 3 and 4. We address future work and conclude the thesis in Chapter 5. Finally, Appendix A describes several technical definitions, functions, and lemmas supporting the formalization of holistic specifications and presents a partial proof of a theorem mentioned in Chapter 4.

1.1 Contributions

The main contributions of this thesis are the following:

1. We have given the first version of the formalization of *Chainmail* in Isabelle and provide several lemmas related to the holistic concepts.
2. We have built lemmas and provide proofs and "pen-and-paper" proofs [1] to place the foundations for reasoning about holistic specifications.

[1]This thesis will distinguish between "proofs" done in Isabelle and "pen-and-paper proofs" sketched but not verified in Isabelle.

Chapter 2

Background and Related Work

This chapter introduces the minimum necessary preliminaries of holistic specifications and Isabelle/HOL to follow the subsequent sections. Furthermore, we present a survey on current work related to the object-capability model [21], focusing on specification and verification for object-capability programs.

Chapter Outline

- **Background.** Section 2.1 gives a brief overview of Isabelle/HOL and its details of the necessary technical background, covers definitions and feature concepts of holistic specifications, and presents Bank/Account example.

- **Related Work.** Section 2.2 provides surveys on the foundations and current publications related to the object-capability model.

2.1 Background

2.1.1 Isabelle/HOL

Isabelle/HOL [26] is a proof assistant, a computer program that assists in conducting proofs of theorems using higher-order logic. It is being developed at the University of Cambridge and Technische Universität München. Isabelle/HOL gives the languages for mathematical reasoning and the rules similar to natural deduction's rules to carry out proofs. We can utilize Isabelle/HOL to show mathematical proofs and reason and prove the semantics of a programming language and its properties. Isabelle/HOL developments comprise a list of theories, the definition of functions, types, sets, and a set of lemmas, theorems, and so forth, interpreted by the theorem prover. It also provides two ways of writing proofs: (1) a tactic script and (2) a structured proof language called Isar (Intelligible Semi-Automated Reasoning). In this thesis, we frequently shift between tactic script and Isar to produce proofs of lemmas.

Furthermore, Isabelle/HOL supplies us with three means to validate whether our theories or lemmas are correct or not. These are, respectively, counter-example commands, tactics, and inference tools. The commands **quickcheck** and **nitpick** are used to search for and generate counter-examples. Tactics include **auto** and **metis** and assist us in proving specific goals automatically. Other automated tools, such as **sledgehammer** or **solve_direct**, help us create proofs for the current goal by recommending tactics.

We now provide in detail the necessary technical background on Isabelle/HOL to understand this thesis's remainder. First, we want to talk about **Types**. They are (1) **basic types**, in particular `nat`, the type of natural numbers, (2) **type constructors**, particularly `list`, the type of lists, (3) **arbitrary types** represented by variable (denoted by `'a`), (4) **recursive types** represented by the `datatype` command introducing recursive data types.

Moreover, the keyword `typedecl` describes a new type name without any additional assumptions, e.g., `typedecl FieldName` describes a set of field names `FieldName`. Besides, the keyword `type_synonym` introduces a synonym for the type specified, e.g., `type_synonym Identifier = nat` introduces a synonym `Identifier` for the type of natural number `nat`. Datatype `option` is used to add a new element `None` to an existing type `'a`. For instance, in this thesis, we usually use the `option` type to represent partial functions. In particular, the partial function from `a` to `b` is represented by the type `a ⇒ b option`.

Second, we speak about **Terms**: (1) **function application** `f t` is the call function `f` with `t` is an argument, (2) **function abstraction** `λx. t` is the function, where `x` is a parameter and returning value `t`.

The third is a **definition**, also called a non-recursive definition. It is defined as a `definition` command, for instance, `inc` as follows.

```
definition inc:: "nat ⇒ nat"
  where
"inc n = n + 1"
```

Besides, we use the **recursive function** in this thesis. We most use two common ways to define a recursive function: (1) `fun` defines a more expressive function, and we might need to prove termination manually, and (2) `primrec`, a restrictive version of `fun`, defines a recursive function in which we need to state every rule. Furthermore, the **inductive** definition is essential. Thus, it is the key construct of operational semantics in the next part of the thesis.

Forth, **records** are used in the thesis, generalizing tuples' concept, but their components have names instead of position. A `record` declaration introduces new types and types of abbreviations. For example, the record of type `pt` represents a point in three-dimensional space, in which three fields named `x`, `y`, and `z` of type `int`.

```
record pt = x :: int y :: int z :: int
```

Finally, as we mentioned earlier, in this thesis, there are two ways of writing proofs: a tactic script and an Isar. Let us give a glimpse at both of them. A tactic script, called "apply" style proofs, is backward reasoning, progressing from goal to premises. On the other hand, Isar appears like a mathematical reasoning style with structured proofs and similar notations such as `assume`, `have`, `thus`, and `hence`.

We give a toy example that proves `P ∧ Q ⟶ Q ∧ P`, to say the difference between "apply" and Isar.

In the first one `applyStyle` utilizing the `apply` tactic, the audience will probably face difficulty to read since the proof does not show the result of each step.

```
lemma applyStyle: "P ∧ Q ⟶ Q ∧ P"
  apply (rule impI)
  apply (rule conjI)
   apply (rule conjunct2)
   apply assumption
  apply (rule conjunct1)
  apply assumption
  done
```

On the other hand, the second one, `IsarStyle`, is more structured and readable. Similar to the mathematical language, we want to show intermediate steps as statement `G1` and statement `G2`. Then, from these intermediate steps, we can prove the ultimate goal.

```
lemma IsarStyle: "P ∧ Q ⟶ Q ∧ P"
proof
  assume H: "P ∧ Q"
  from H have G1: "P"
```

```
    by (rule conjunct1)
  from H have G2: "Q"
    by (rule conjunct2)
  from G2 and G1 show "Q ∧ P"
    by (rule conjI)
qed
```

2.1.2 Holistic Specifications

Specifications of functional correctness of programs describe what sufficient conditions for some effect to happens are. Considering a bank system as an example, if we have enough money and make a payment request to the bank, the money will be transferred, and as a consequence, our balance will be reduced. *Enough money* and *the payment request* are a sufficient condition for the decrease in money. In contrast, necessary conditions guarantee that things will not happen. For instance, we desire the bank to ensure no reduction in our bank balance will occur unless requested. The difference between sufficient and necessary conditions is that sufficient conditions are described on an individual function. Necessary conditions, on the other hand, are about the behavior of a component as a whole. When our component cooperates with other unknown provenance systems in an open world, we want the component to meet its sufficient and necessary conditions. Consequently, its specifications should be *holistic*. Namely, it illustrates the overall behavior of a component: each function's behavior and limitations on the behavior that emerges from combinations of functions. In other words, holistic specifications [8] must, therefore, address sufficient as well as necessary conditions. The discrepancy between classical invariants and holistic specifications is that the classical invariants reflect on the current program state. In contrast, holistic specifications reflect on all aspects of a program's execution, possibly over all the components making up that program.

Holistic specifications extend traditional program specifications with ideas from object capabilities (permission and authority), temporal logic, spatial connectives, and viewpoint, describing in [8] as follows.

- **Permission**: Objects may have access to which other objects; this is fundamental as access to an object privileges access to its functions.
- **Control:** Objects called functions on other objects; this is beneficial in identifying the causes of specific effects.
- **Time:** What holds some time in the past, the future, and what changes with time.

- **Space:** Which parts of the heap are considered when establishing some property or when performing program execution.
- **Viewpoint:** What objects and runtime configurations [2] are internal to our component, which are external to it.

Chainmail [8] is a specification language to draw the holistic specifications addressed beforehand. Mainly, Chainmail assertions include pure expressions, classical assertions about the contents of heap and stack, comparisons between expressions, and the usual logical connectives. Also, Chainmail can address holistic ideas, including permission, control, time, space, and viewpoint.

- **Permission:** Permission states an object has a direct path to another object, represented as assertion ⟨`x Access y`⟩ saying that if there is a direct path from the object `x` to another object `y`: either object `x` and `y` are aliases, or object `x` points to an object with a field whose value is the same as the object `y`, or object `x` is currently executing an object, and object `y` is a local parameter.
- **Control:** Control assertion represents the object making a function call on another object, expressed as assertion ⟨`x Calls y.m(zs)`⟩ stating that it holds if in a runtime configuration in which there is a method on the object `x` that performs the method call `y.m(zs)`. Here, object `x` is a caller, object `y` is a receiver, and `x` calls method `m` on `y` with arguments `zs`.
- **Time:** Temporal assertions are a part of the holistic assertions consisting of `Next`⟨`A`⟩, `Will`⟨`A`⟩, `Prev`⟨`A`⟩, and `Was`⟨`A`⟩. In particular, assertions `Next`⟨`A`⟩ and `Will`⟨`A`⟩ talk about the future, in which `A` holds at the immediate successor step, and some future point, respectively. Otherwise, assertions `Prev`⟨`A`⟩ and `Was`⟨`A`⟩ say about the past, in which `A` holds at the predecessor step and a number of steps in the past, respectively.
- **Space:** An assertion ⟨`A In S`⟩ says that `A` holds in a runtime configuration restricted to objects from a given set `S`. In other words, the objects that make `A` valid should be included in the set `S`.
- **Viewpoint:** Assertions `Internal`⟨`x`⟩ and `External`⟨`x`⟩ state whether the object at `x` belongs to the module under consideration or not, respectively.

[2] Runtime configurations include all the information about an execution snapshot: the stack and heap of frames.

- **Change and Authority:** An assertion $\texttt{Changes}\langle\texttt{x}\rangle$ provides conditions for change to occur; it also called authority.

2.1.3 Bank/Account example

We now put features stated earlier together through the Bank/Account application obtained from object capabilities literature [22]. In particular, we choose the policy **Pol_1**: " With two accounts of the same bank, one can transfer money between them," and the policy **Pol_2**: "Only someone with the bank of a given currency can violate conservation of that currency."

Pol_1 states that clients can transfer money between accounts as long as their accounts belong to the same bank. It is not a surprise to recognize that it merely sufficient conditions, expressible within pre-condition and post-condition.

In contrast, the policy **Pol_2** captures necessary conditions: Avoiding a bank's currency from getting changed without access to the bank is necessary. The policy says that currency might be increased or decreased by some code if the code involves a function call in which it is performed by the bank holding the currency. Employing holistic assertions presented above, we have a holistic specification of **Pol_2** as follows.

$$\begin{aligned}\textbf{Pol\_2} \triangleq\ & \texttt{b:Bank} \land \texttt{Will}\big(\texttt{Changes}\big(\texttt{b.currency}\big)\big)\,\texttt{In S} \\ & \Longrightarrow \exists \texttt{o} \in \texttt{S.}[(\texttt{o Access b}) \land (\texttt{o} \not\in \texttt{Internal(b)})]\end{aligned} \tag{2.1}$$

Formula 2.1 says that if some execution involving objects, given by the set S, changes the currency in the bank b at some future time, then there is at least one object from the given set S that can access the bank b directly, and this object is external to the bank b.

We reformulate Formula 2.1 of **Pol_2** into an equivalent one below.

$$\begin{aligned}\textbf{Pol\_2} \triangleq\ & \texttt{b:Bank} \land [\ \forall \texttt{o} \in \texttt{S.}[\ \neg(\texttt{o Access b}) \lor (\texttt{o} \in \texttt{Internal(b)})\ ]\ ] \\ & \Longrightarrow \neg\big(\texttt{Will}\big(\texttt{Changes}\big(\texttt{b.currency}\big)\big)\,\texttt{In S}\big)\end{aligned} \tag{2.2}$$

Formula 2.2 represents that the set S whose elements have direct access to b only if they are internal to b is *insufficient* to modify the currency in b at some future time.

Now, let us give some outline to consider how to reason the policy **Pol_2**. Formally, we assign:

- $\texttt{P} := \forall \texttt{o} \in \texttt{S}.[\neg(\texttt{o Access b}) \vee (\texttt{o} \in \texttt{Internal(b)})]$
- $\texttt{T(o)} := [\neg(\texttt{o Access b}) \vee (\texttt{o} \in \texttt{Internal(b)})]$
- $\texttt{Q} := \texttt{Changes}\big(\texttt{b.currency}\big)$

Formula 2.2 is now equivalent to $\forall \texttt{o} \in \texttt{S}.\,\texttt{T(o)} \implies \neg(\texttt{Will(Q) In S})$. We want to prove that P is an invariant in the set S. Therefore, we "hope" to establish two separate hypotheses with the first one considered as an invariant as follows.

- (1) $\forall \texttt{o} \in \texttt{S}.\,\texttt{T(o)} \implies (\texttt{Next}(\forall \texttt{o} \in \texttt{S}.\,\texttt{T(o)}))\ \texttt{In S}$
- (2) $(\forall \texttt{o} \in \texttt{S}.\,\texttt{T(o)} \implies \neg\,\texttt{Next(Q)})\ \texttt{In S}$

We "hope" that (1) and (2) are correct, and the conjunction of (1) and (2) that implies Formula 2.2 is correct as well. So far, to support the above proofs, we also need lemmas related to spatial connective assertions.

Lemma 1. *For any assertions A, B, and a set S, we have*
$(\langle A\ \ In\ \ S\ \rangle \implies \langle B\ \ In\ \ S\ \rangle) \equiv \langle (A \implies B)\ In\ \ S\ \rangle$.

Lemma 2. *For any assertions A and a set S, we have*
$\neg\langle A\ \ In\ \ S\ \rangle \equiv \langle(\neg A)\ In\ \ S\ \rangle$.

We prove these two technical lemmas in Isabelle in Section 3.7.4 of the next chapter as well.

2.2 Related Work

The section reviews the foundations and current publications related to the object-capability model [21]. It also directs on specification and verification for object-capability programs.

2.2.1 Behavioral Specification Languages

Meyer [20] first presented verification techniques, called "Design by Contract", for object-oriented programs, whose specifications along with the form of pre-conditions and post-conditions on methods, as exemplified by the programming language Eiffel. These ideas appear in modern specification languages aimed at realistic employment, including Spec# [1], Dafny [14], JML [13], as well as Whiley [28].

Leino and Schulte [15] used history invariants that have a two-state predicate to specify the behavior of an object. Their technique was built on an object invariant concept and was used to verify the invariants of the observer pattern. Another track of work on a specification for object-oriented programs is Summers and Drossopoulou's Considerate Reasoning [29]. They proposed another approach based on object invariants to construct a specification and verification technique for object-oriented languages.

While these approaches concern specifications on object-oriented languages, they assume their systems live in a closed world, where other components can be trusted to collaborate.

2.2.2 Object Capabilities and Sandboxes

Miller [21] first introduced the object-capability model, and several recent studies manage to verify the correctness or verify the safety of object-capability programs. Google's Caja [23], an object-capability subset of Javascript, utilizes sandboxes for securing web mashups to restrict access of components to ambient authority. Some programming languages and web systems have utilized the object-capability model, including E [18], Grace [2], Dart [3], and Wyvern [17]. Miller et al. [21, 24] also define the fundamental way as defensive consistency "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 misbehaviors by its other clients."

Maffeis et al. [16] developed a notation of authority safety based on object capabilities. They proved that it follows two principles: authority safety is implied from capability safety and adequate for providing resource isolation. From that, they modeled semantics for these two principles utilizing small-step operational semantics. They also defined a language *Js*, a Javascript subset, to show it is sufficient to provide isolation among untrusted applications. Moreover, another contribution of the paper pointed out that these two principles hold for a class of object-capability languages, such as Cajita, a Caja-based object-capability subset Javascript. However, the Js language does not carry the object-capability model.

2.2.3 Verification of Object-Capability Programs

There are numerous preliminary studies on the specification and verification of object-capability programs.

Murray [25] early attempted to formalize defensive consistency and correctness utilizing process calculi. Murray constructed a process algebra CSP (communicating sequential processes) model to reason about the security properties of concurrent object-capability patterns. Then, Murray applied CSP's model checker FDR to analyze patterns in the presence of untrusted objects.

Drossopoulou and Noble analyzed Miller's Mint and Purse [22], a simple example of capability-based money, and implemented in Joe-E [5] and Grace [27] to argue that current specifications are insufficient for reasoning about all aspects of capability policies. Drossopoulou et al. [6] also proposed specifications for an open world in modeling risk and trust. They used it to specify an Escrow exchange contract [24], a trusted third agent that handles exchanges money between untrusted parties. While they focus on the specification language, they do not define semantics for their predicates, and they only focus on the example of the Escrow exchange.

A following technical report [7] formalizes and provides the semantics making a connection for a gap of their early work by proposing a Hoare logic style for reasoning about risk and trust between objects and formally proving the Escrow protocol meets the specification. Although they provide the Hoare logic, their Hoare logic lacks rules for dynamic allocation. Also, there is no proof of the soundness of the Hoare logic. Recently, Drossopoulou et al. [8] present *Chainmail*, a specification language for writing holistic specifications, which concerns both sufficient conditions and necessary conditions.

Devriese et al. [4] adopted a step-indexed Kripke logical relation, as well as a notion called *effect parametricity*, to reason about integrity properties of various examples of capability-wrapped user code in a language with the higher-order state. Furthermore, they verified some non-trivial examples proving the preservation of invariants on shared data structures of the user code in the presence of untrusted or unknown code. They also showed solutions to some example problems, such as a mashup application and the DOM wrapper. Nevertheless, the model is insufficient in providing no way specifying what an object-capability pattern does compositionally.

Swasey et al. [30] presented a program logic to specify and verify object-capability programs compositionally, called OCPL. First, they build their logic based on a framework for concurrent separation logic, called Iris [10, 12, 11], and mechanized in Coq, a proof assistant. The core idea of OCPL adapts the notion of a low-integrity value in which it can be shared with untrusted code safely. From that, it identifies the interface between the piece of verified user and untrusted code. Then, compared to the previous work of Devriese et al. [4], OCPL gives a modular way of specifying a general property compositionally that untrusted code can share with used code safely. Finally, they also apply the logic to several object-capability patterns, including reason about a general membrane, sealer-unsealer pair, and the caretaker.

Chapter 3

Formalizing Holistic specifications in Isabelle/HOL

This chapter formalizes most of the core and formal semantics of *Chainmail* [8] in Isabelle/HOL version 2020. It involves formalized definitions, theorems, and technical lemmas, as well as lemmas supporting Holistic specifications.

Chapter Outline

- **Language Syntax.** Section 3.1 describes the language syntax of underlying object-oriented programming.
- **Operational Semantics of the Language.** Section 3.2 presents the operational semantics of the language mentioned in Section 3.1.
- **Module Linking.** Section 3.3 covers definitions module linking and proofs of its properties.
- **Module pairs and visible-states semantics.** Section 3.4 formalizes the visible-states semantics and the pairing of an internal module and external module.
- **Initial and Arising configurations.** Section 3.5 provides the definitions of Initial and Arising configurations.
- **Assertions-Classical Assertions.** Section 3.6 gives the formalization of syntax and semantics of holistic assertions.

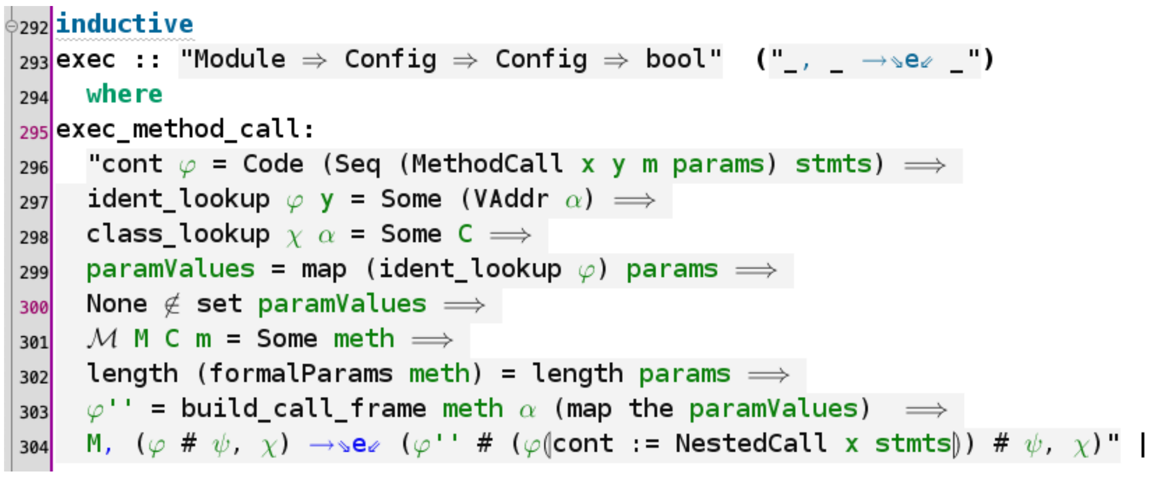
- **Assertions - Access, Control, Space, Authority, and Viewpoint.** Section 3.7 focuses on the formalization of holistic concepts.
- **Summary.** Section 3.8 summarizes the formalization of holistic specifications.

We first want to give motivation as to how to formalize the *Chainmail* in Isabelle/HOL. Figure 3.1 depicts the method calls rule (`methCall_OS`) in the operational semantics of *Chainmail*, taken from [8]. Given the abstract syntax's recursive nature, it will not be a wonder that our pick is an inductive definition. Let us rephrase the method calls rule (`methCall_OS`) in natural language. The rule `exec_method_call` defines the semantics for method calls of the form $\mathtt{x} := \mathtt{x}_0.\mathtt{m}(\mathtt{x}_1,\ldots,\mathtt{x}_n)$. It looks up in the current stack frame ϕ the object $\mathtt{x}_0$ being invoked, producing the address α, which is used to retrieve its class Class from the heap χ. The method name `m` of class Class is looked up in the module `M` from which the new stack frame ϕ for the execution of the method is produced. The continuation `cont` is updated to remember that a nested call is being executed, whose result will be assigned to `x`.

methCall_OS

$$\frac{\begin{array}{l}\phi.\mathtt{contn} = \mathtt{x} := \mathtt{x}_0.\mathrm{m}(\mathtt{x}_1,\ldots\mathtt{x}_n)\,;\ \mathtt{Stmts}\\ \lfloor \mathtt{x}_0 \rfloor_\phi = \alpha\\ \mathcal{M}(\mathtt{M}, \mathit{Class}(\alpha)_\chi, \mathtt{m}) = \mathtt{m}(\mathtt{p}_1,\ldots\mathtt{p}_n)\,\{\,\mathtt{Stmts}_1\}\\ \phi'' = (\,\mathtt{Stmts}_1,\ (\mathtt{this} \mapsto \alpha, \mathtt{p}_1 \mapsto \lfloor \mathtt{x}_1 \rfloor_\phi, \ldots \mathtt{p}_n \mapsto \lfloor \mathtt{x}_n \rfloor_\phi\,)\,)\end{array}}{\mathtt{M},\ (\,\phi\cdot\psi,\ \chi\,)\ \rightsquigarrow\ (\,\phi''\cdot\phi[\mathtt{contn} \mapsto \mathtt{x} := \bullet\,;\ \mathtt{Stmts}]\cdot\psi,\ \chi\,)}$$

Figure 3.1 Operational semantics of rule `methCall_OS` of *Chainmail* [8].



```
inductive
exec :: "Module ⇒ Config ⇒ Config ⇒ bool"  ("_, _ →↘e↙ _")
  where
exec_method_call:
  "cont φ = Code (Seq (MethodCall x y m params) stmts) ⟹
  ident_lookup φ y = Some (VAddr α) ⟹
  class_lookup χ α = Some C ⟹
  paramValues = map (ident_lookup φ) params ⟹
  None ∉ set paramValues ⟹
  ℳ M C m = Some meth ⟹
  length (formalParams meth) = length params ⟹
  φ'' = build_call_frame meth α (map the paramValues) ⟹
  M, (φ # ψ, χ) →↘e↙ (φ'' # (φ⦇cont := NestedCall x stmts⦈) # ψ, χ)" |
```

Figure 3.2 Rule `exec_method_call` presenting in Isabelle for rule `methCall_OS`.

Following the rules of Figure 3.1, we formalize the rule `methCall_OS` in Isabelle/HOL, drawing in Figure 3.2. To formalize it, we define it as an inductive predicate using the command `inductive`. The type of `exec` is `Module` ⇒ `Config` ⇒ `Config` ⇒ `bool`, where `Module` is the set of mappings from class names to class descriptions, and `Config` is runtime configurations. We also introduce the concrete syntax (_, _ $\rightarrow_e$ _) for `exec`. The Isabelle definition of the method calls rule (`exec_method_call`) does not directly connect with the method calls rule (`methCall_OS`) of *Chainmail*. The rules make use of some auxiliary definitions: (1) (`ident_lookup` ϕ y) function is used to look up identifier y in the stack frame ϕ; (2) (`class_lookup` χ α) function is used to retrieve the class of the object whose address is α in the heap χ; (3) `build_call_frame` is used to create a new frame in which each value assigns each parameter of method `meth` in `paramValues`. The remaining rules (variable assignment, field assignment, object creation, and return) are formalized and detailed in Section 3.2.4.

3.1 Language Syntax

In the first part of the formalization, we give the language syntax of underlying object-oriented programming and its operational semantics and connect other modules to the module under consideration. We formalize all of them in Isabelle/HOL. Firstly, we are required to formalize the syntax of the language. We present them by a given set of program variables *`Identifier`*, a set of field names *`FieldName`*, a set of class names *`ClassName`*, and a set of method names *`MethodName`*. The fields of the language declared are untyped, the method consists of an untyped parameter, with no return type as well, because the core language of Holistic specifications is untyped.

```
type_synonym Identifier = nat
typedecl FieldName
typedecl ClassName
typedecl MethodName
```

Statements *`Stmt`* in the programming language consist of field read *`ReadFromField`*, field write *`AssignToField`*, method call *`MethodCall`*, object creation *`NewObject`*, and return statement *`Return`*.

```
datatype Stmt = AssignToField FieldName Identifier
              | ReadFromField Identifier FieldName
              | MethodCall Identifier Identifier MethodName "Identifier list"
              | NewObject Identifier ClassName "Identifier list"
```

```
                | Return Identifier
```

The sequences of a statement are declared as `Stmts` as well. It consists of a single statement and a sequence of statements.

```
datatype Stmts = SingleStmt Stmt | Seq Stmt Stmts
```

The method body consists of sequences of statements and their parameters, which are a list of identifiers. Notice that we store the method names inside the class to make lookup easier.

```
record Method =  formalParams :: "Identifier list"
                  body :: Stmts
```

The class description consists of field and method declarations. Note that we omit the `ClassId` as it appears to be duplicated in the `Module`. We also assume that every method is defined on every object for the sake of simplicity.

```
record Class =   objFields :: "FieldName list"
                  methods   :: "MethodName ⇒ Method"
```

A module is defined as the set of mappings from class names to class descriptions. Note that the class name is distinct from local variables. Every class is not defined by every module, as otherwise linking becomes meaningless.

```
type_synonym Module  = "ClassName ⇒ Class option"
```

Method lookup function $\mathscr{M}$ returns a method `m` for a class `C` inside the module `M`.

```
definition
𝓜 :: "Module ⇒ ClassName ⇒ MethodName ⇒ Method option"
  where
"𝓜 M C m ≡ case (M C) of None ⇒ None | Some c ⇒ Some ((methods c) m)"
```

3.2 Operational Semantics of the Language

3.2.1 Interpretations

Heaps χ are mappings from addresses α to objects `obj`. Stack frames φ are mappings from identifiers `x` plus the distinguished identifier `this` to values, where values include addresses. Stack frames also store the current continuation (code to be executed).

Configurations σ are pairs $(\psi,\ \chi)$ where ψ is a list of stack frames φ and χ is the heap.

The following notation we use throughout the upcoming sections.

- Lookup of fields `f` on object with address α in the heap χ, `field_lookup` χ α `f`, is written $\chi(\alpha,\ f)$.
- The class of the object whose address is α in heap χ, `class_lookup` χ α, is written $Class(\alpha)_\chi$.
- Lookup of the class of the `this` object in the runtime configurations σ, `this_class_lookup` σ, is written $Class(this)_\sigma$.
- Lookup of identifier `x` in the frame φ, `ident_lookup` φ `x`, is written $\lfloor x \rfloor_\varphi$.
- Lookup of identifier `x` in context σ, `evalVar x` σ, is written $\lfloor x \rfloor_\sigma$.
- Lookup of field `f` from `this` object in frame φ and heap χ, `this_field_lookup` φ χ `f`, is written $\lfloor this.f \rfloor_{(\varphi,\chi)}$.
- Update the variable map of frame φ so that variable `x` maps to value `v`, `frame_ident_update` φ `x v`, is written $\varphi[x \mapsto v]$.
- Update the object at address α on the heap χ to the object `obj`, `heap_update` χ α `obj`, is written $\chi[\alpha \mapsto obj]$.
- To obtain the continuation `cont` from the frame φ, we write `cont` φ.

3.2.2 Runtime Entities

We are ready to introduce the addresses `Addr` as an enumerable set and null `Null`.

type_synonym *Addr = nat*
consts *Null :: Addr*

Then, we also define values `Values`, consisting of null, addresses, and sets of addresses `VAddrSet`.

datatype *Value = VAddr Addr | VAddrSet "Addr set"*

Continuations are either statements or a nested call followed by statements. Frames consist of a continuation, a mapping from identifiers to values, and an address `this`.

datatype *Continuation = Code Stmts | NestedCall Identifier Stmts*

```
record Frame =  cont :: Continuation
                vars :: "Identifier ⇒ Value option"
                this :: Addr
```

Stacks are sequences of frames. Objects consist of a class identifier `className` and a mapping from field name `FieldName` to values `values`. Heaps `Heap` are defined as mappings from addresses to objects. Lastly, runtime configurations `Config` are pairs of stacks and heaps.

We use the option type to represent partial functions. i.e., the partial function from `a` to `b` is represented by the type `a ⇒ b option`. For instance, the `objFields` is a partial function from `FieldName` to `Value` defined below.

```
type_synonym Stack = "Frame list"
record Object = className :: ClassName
                objFields :: "FieldName ⇒ Value option"
type_synonym Heap = "Addr ⇒ Object option"
type_synonym Config = "Stack × Heap"
```

3.2.3 Lookup and update of runtime configurations

We represent interpretations of a field lookup and a class lookup. The `field_lookup` function is used to retrieve the value stored in field `f` of the object at address α in the heap χ. It is a partial function since there might be no such object at address α.

```
fun
field_lookup :: "Heap ⇒ Addr ⇒ FieldName ⇒ Value option"
  where
"field_lookup χ α f =
        (case (χ α) of None ⇒ None
                    | Some obj ⇒ (objFields obj) f)"
```

The function `class_lookup` is used to retrieve the class of the object `obj` whose address is α in the heap χ. It is a partial function since there might be no such class at address α.

```
fun
class_lookup :: "Heap ⇒ Addr ⇒ ClassName option"
  where
"class_lookup χ α =
        (case (χ α) of None ⇒ None
                     | Some obj ⇒ Some (className obj))"
```

The function `ident_lookup` is used to retrieve identifier `x` in the frame φ. It is a partial function since there might be no such identifier in the frame φ.

fun

```
ident_lookup :: "Frame ⇒ Identifier ⇒ Value option"
```

where

```
"ident_lookup φ x = vars φ x"
```

The function `this_field_lookup` is used to retrieve the class of the `this` object in the runtime configuration σ. It is a partial function since there might be no such class in the runtime configuration σ.

fun

```
this_field_lookup :: "Frame ⇒ Heap ⇒ FieldName ⇒ Value option"
```

where

```
"this_field_lookup φ χ f =
    (case χ (this φ) of None ⇒ None
                 | Some obj ⇒ objFields obj f)"
```

The auxiliary function `this_field_update` is used to update the field `f` of the distinguished `this` object to refer to value `v`, given the stack frame φ and heap χ.

fun

```
this_field_update :: "Frame ⇒ Heap ⇒ FieldName ⇒ Value ⇒ Heap option"
```

where

```
"this_field_update φ χ f v =
    (case χ (this φ) of None ⇒ None
                 | Some obj ⇒ Some (χ(this φ :=
            Some (obj⦇objFields := ((objFields obj)(f := Some v))⦈))))"
```

The function `frame_ident_update` updates the variable map of the stack frame φ so that variable `x` maps to value `v`.

fun

```
frame_ident_update :: "Frame ⇒ Identifier ⇒ Value ⇒ Frame"
```

where

```
"frame_ident_update φ x v = (φ⦇vars := ((vars φ)(x := Some v))⦈)"
```

The function `heap_update` updates the object at address α on the heap χ to map to the object `obj`.

```
fun
heap_update :: "Heap ⇒ Addr ⇒ Object ⇒ Heap"
  where
"heap_update χ α obj = χ (α := Some obj)"
```

The function `this_class_lookup` is used to look up the class of the `this` object in the runtime configuration σ.

```
fun
this_class_lookup :: "Config ⇒ ClassName option"
  where
"this_class_lookup σ =
   (case σ of (φ#ψ,χ) ⇒
       (case χ (this φ) of None ⇒ None
                      | Some obj ⇒ Some (className obj))
           | _ ⇒ None)"
```

Note that these above functions return `option` in case of lookup failure.

3.2.4 Operational semantics

Now, we turn to the operational semantics of the programming language. We define it as an inductive predicate called `exec`. Its rules make use of the following auxiliary definitions.

The auxiliary function `build_call_frame` is used to create a new frame in which the `this` object is assigned by address α, and each value also assigns each parameter of method `meth` in `paramValues`. We also need a condition that the length of method `meth` equals the length of the list `paramValues`. We implement such a condition in the rule `exec_method_call`.

```
definition
build_call_frame :: "Method ⇒ Addr ⇒ Value list ⇒ Frame"
  where
"build_call_frame meth α paramValues ≡
      ⦇cont = Code (body meth),
       vars = map_of (zip (formalParams meth) paramValues),
       this = α⦈"
```

The auxiliary function `build_new_object` creates a new object in which each value in the list `fieldValues` assigns each field in class `c`. Such a function is useful for the object creation's rule `exec_new`. Same as the function `build_call_frame`, the length of both the list of fields in class `c` and the list `fieldValues` must equal.

```
definition
build_new_object :: "ClassName ⇒ Class ⇒ Value list ⇒ Object"
  where
"build_new_object C c fieldValues ≡
      ⦇className = C,
       objFields = map_of (zip (Class.objFields c) fieldValues)⦈"
```

In object creation's rule `exec_new`, we need a fresh address in heap χ, i.e., the new address utterly different from addresses including the current heap χ. The function `fresh_nat` generates such a fresh address.

```
definition
fresh_nat :: "Identifier set ⇒ Identifier"
  where
"fresh_nat X = (if X = {} then 0 else (Suc (last (sorted_list_of_set X))))"
```

Lemma `fresh_nat_is_fresh` says that such a fresh address is not in the current heap χ.

```
lemma fresh_nat_is_fresh [simp]:
  "finite X ⟹ fresh_nat X ∉ X"
  apply (induct rule: finite.induct)
   apply simp
  apply(clarsimp simp: fresh_nat_def)
  using not_eq_a not_in_A by auto
```

An operational semantics is represented as a set of rules given formally below. These rules are method calls `exec_method_call`, variable assignment `exec_var_assign`, field assignment `exec_field_assign`, object creation `exec_new`, and return `exec_return`.

The rule `exec_method_call` defines the semantics for method calls of the form `x := y.m(params)`. It looks up in the current stack frame φ the object `y` being invoked, producing the address α, which is used to retrieve its class `C` from the heap χ; the rule also looks up each identifier in `params`, checking to make sure that each such lookup succeeds. The method name `m` of class `C` is looked up in the module `M` from which the new stack frame φ'' for the execution of the method is produced. The continuation is updated to remember that a nested call is being executed, whose result will be assigned to `x`.

The rule *exec_var_assign* defines the semantics for field read of form *x := this.y*. The continuation is updated to remember that a sequence of statements *stmts* is being executed. The result of the lookup of the field *y* from *this* object in the frame φ and the heap χ will be assigned to *x*.

The rule *exec_field_assign* defines the semantics for field write of form *this.y = x*. The continuation is updated to remember that a sequence of statements *stmts* is being executed. The rule looks up identifier *x* in the current stack frame φ, producing the value *v*, which is updated to the field *y* to create a new heap χ*'* given the stack frame φ and heap χ.

The rule *exec_new* defines the semantics for object creation of form *x := new C(params)*. The rule looks up each identifier in *params*, checking to make sure that each such lookup succeeds. A fresh address α will update the evaluation of identifier *x* in the current heap σ. The heap χ at the address α will be updated by a new object, which each field declared in the method *m* of class *C* will be updated by each value of each identifier in *params*, respectively.

The rule *exec_return* defines the semantics for the return statement of form *return x*. The value of identifier *x* in stack frame σ is assigned to the identifier *x'* where the result of a nested call is assigned. The continuation is updated to remember that a statement *stmts'* followed by the nested call is being executed.

```
inductive
exec :: "Module ⇒ Config ⇒ Config ⇒ bool"  ("_, _ →e _")
  where
exec_method_call:
 "cont φ = Code (Seq (MethodCall x y m params) stmts) ⟹
  ident_lookup φ y = Some (VAddr α) ⟹
  class_lookup χ α = Some C ⟹
  paramValues = map (ident_lookup φ) params ⟹
  None ∉ set paramValues ⟹
  ℳ M C m = Some meth ⟹
  length (formalParams meth) = length params ⟹
  φ'' = build_call_frame meth α (map the paramValues)  ⟹
  M, (φ # ψ, χ) →e (φ'' # (φ⦇cont := NestedCall x stmts⦈) # ψ, χ)" |

exec_var_assign:
 "cont φ = Code (Seq (ReadFromField x y) stmts) ⟹
```

```
  M, (φ # ψ, χ) →e
  ((φ ⦇ cont := Code stmts ,
        vars := ((vars φ)(x := this_field_lookup φ χ y)) ⦈ ) # ψ, χ)" |

exec_field_assign:
 "cont φ = Code (Seq (AssignToField y x) stmts) ⟹
  ident_lookup φ x = Some v ⟹
  this_field_update φ χ y v = Some χ' ⟹
  M, (φ # ψ, χ) →e (φ ⦇cont := Code stmts ⦈ # ψ, χ')" |

exec_new:
 "cont φ = Code (Seq (NewObject x C params) stmts) ⟹
  paramValues = map (ident_lookup φ) params ⟹
  None ∉ set paramValues ⟹
  M C = Some c ⟹
  length params = length (Class.objFields c) ⟹
  obj' = build_new_object C c (map the paramValues) ⟹
  α = fresh_nat (dom χ) ⟹
  χ' = χ(α := Some obj') ⟹
  M, (φ # ψ, χ) →e (φ ⦇cont := Code stmts,
        vars := ((vars φ)(x := Some (VAddr α)))⦈ # ψ, χ')" |

exec_return:
 "cont φ = Code (SingleStmt (Return x)) ∨
  cont φ = Code (Seq (Return x) stmts) ⟹
  cont φ' = NestedCall x' stmts' ⟹
  M, (φ # φ' # ψ, χ) →e ((φ' ⦇ cont := Code stmts',
        vars := ((vars φ) (x' := ident_lookup φ x))⦈) # ψ, χ )"
```

We formally define the execution of more steps `exec_rtrancl`, which is the reflexive, transitive closure of `exec`, as follows.

```
inductive
exec_rtrancl:: "Module ⇒ Config ⇒ Config ⇒ bool" ("_, _ →e* _")
  where
exec_rtrancl_equiv: "σ = σ' ⟹ M, σ →e* σ'" |
exec_rtrancl_trans: "⟦ M, σ →e* σ''; M, σ'' →e σ'⟧ ⟹ M, σ →e* σ'"
```

To have connections with Chainmail, we introduce the concrete syntax `(_, _ →e _)` for `exec` and `(_, _ →e* _)` for `exec_rtrancl`.

3.3 Module Linking

In `an open world`, to reason about the operation of a module, we need to talk about how it behaves when operating in the presence of other (possibly untrusted) code. For that purpose, we define what it means to combine two modules in a module linking operator. Later, this will be used to model the operation of module `M` in the presence of other untrusted modules `M'` that it is linked to.

We know that the linking should be well-formed. So, the function `link_wf` represents the well-formedness of the linking, saying that when the two modules do not both define the same class (i.e., when their domains are disjoint).

```
definition
link_wf :: "Module ⇒ Module ⇒ bool"
  where
"link_wf M M' ≡ dom M ∩ dom M' = {}"
```

```
definition
moduleAux :: "Module ⇒ Module ⇒ ClassName ⇒ Class option" (infixl "∘aux" 55)
  where
"(M ∘aux  M') ≡ λC. (if C ∈ dom M then M C else M' C)"
```

The next step is to define a module linking operator. We introduce concrete syntax $\circ_l$ for the module linking operator `moduleLinking`. The function `moduleLinking` takes two modules `M` and `M'` and returns the union of the two.

```
definition
moduleLinking :: "Module ⇒ Module  ⇒ Module" (infix "∘l" 55)
  where
"M ∘l M' ≡ (M ∘aux  M')"
```

When the linking is well-formed, it should be commutative and associative. We prove that linking is commutative (`link_commute`) and associative (`link_assoc`) when well-formed.

```
lemma link_commute [simp]: "link_wf M M' ⟹  M ∘l M' = M' ∘l M"
  unfolding moduleLinking_def moduleAux_def dom_def
  apply (simp cong: if_cong)+
  apply (auto simp: link_wf_def)
  by fastforce
```

```
lemma link_assoc [simp]:
  "link_wf M M' ⟹ (M ∘l M') ∘l M'' = M ∘l (M'∘l M'')"
  unfolding moduleLinking_def moduleAux_def dom_def link_wf_def
  apply (simp cong: if_cong)+
  by auto
```

3.4 Module pairs and visible-states semantics

Holistic specifications are useful to talk about the interactions between a module M and other potentially untrustworthy modules `M'` that it might interact with. We formally capture these interactions via `visible-state semantics` in which the visible states are those as seen from outside the module `M`, i.e., those in which some `M'` object is running. We name the module `M` and `M'` for the internal module and external module, respectively.

This section introduces the formalization of module pairs and visible-states semantics. A visible execution is a sequence of execution steps that looks like this: $\sigma \rightarrow_e \sigma_2 \rightarrow_e \ldots \sigma_{n-1} \rightarrow_e \sigma_n$ where the class of the `this` object in σ comes from the external module `M'` and the class of the `this` object in σ_n also comes from the external module `M'`. However, the class of the `this` object in every other $\sigma_2, \ldots, \sigma_{(n-1)}$ comes from the internal module `M`.

We capture this using two inductive definitions. The first one, defined as `internal_exec`, talks about the first `n - 2` steps of execution, each such step leads to a configuration σ_i, where $2 \leq i < n$ in which the `this` object of σ_i is in the internal module `M`.

```
inductive
internal_exec ::
  "Module ⇒ Module ⇒ Config ⇒ (Config list) ⇒ Config ⇒ bool"
  ("_;_,_ →ei⟨_⟩ _") for M :: Module and M' :: Module
  where
internal_exec_first_step:
 "link_wf M M' ⟹
  (M ∘l M'), σ →e σ' ⟹
  this_class_lookup σ = Some c ⟹ c ∈ dom M' ⟹
  this_class_lookup σ' = Some c' ⟹ c' ∈ dom M ⟹
  internal_exec M M' σ [σ'] σ'" |
internal_exec_more_steps:
 "internal_exec M M' σ tr σ' ⟹
  (M ∘l M'), σ' →e σ'' ⟹
  this_class_lookup σ'' = Some c ⟹ c ∈ dom M  ⟹
  internal_exec M M' σ (tr@[σ'']) σ''"
```

The second inductive definition as `visible_exec` is just for the final step from $\sigma_{(n-1)}$ to σ_n.

inductive

```
visible_exec :: "Module ⇒ Module ⇒ Config ⇒ Config ⇒ bool" ("_;_,_ →e _")

  where
visible_exec_intro:
 "internal_exec M M' σ tr σ' ⟹
  (M ∘l M'), σ' →e σ'' ⟹
  this_class_lookup σ'' = Some c ⟹
  c ∈ dom M'  ⟹
  visible_exec M M' σ σ''"
```

3.4.1 Determinism

There is a critical thing that we need to prove in the execution of the language, and the execution of the visible-state semantics is deterministic. The language is deterministic when any two executions start from the same state step to the same next state. Formally, the execution of the language or the visible-states is deterministic if for the same initial state σ, there is at most one next state that is reached after one step of execution, i.e., if σ steps to σ' and also to σ'', then σ' = σ''. The lemma `exec_det` shows that the execution of the language is deterministic. The lemma `visible_exec_det` also shows that the execution of the visible-state semantics is deterministic. To prove this lemma, we need several technical definitions and sub-lemmas such as `internal_exec_rev'`, `internal_exec_is_internal`, or `internal_exec_appD` (See Appendix A.2).

To prove the determinism of visible-states semantics, we first need to prove that the `internal_exec` definition is deterministic. To do that, it helps to have an equivalent definition of it that operates in reverse. That definition we call `internal_exec_rev`, which is defined as follows via the intermediate definition `internal_exec_rev'`.

inductive

```
internal_exec_rev' ::
 "Module ⇒ Module ⇒ Config ⇒ (Config list) ⇒ Config ⇒ bool"
  ("_;_,_ →eir1⟨_⟩ _") for M :: Module and M' :: Module
  where
internal_refl:
 "internal_exec_rev' M M' σ [] σ" |
internal_step:
 "(M ∘l M'), σ →e σ' ⟹
  this_class_lookup σ = Some c ⟹ c ∈ dom M ⟹
  this_class_lookup σ' = Some c' ⟹ c' ∈ dom M ⟹
  internal_exec_rev' M M' σ' tr σ'' ⟹
  internal_exec_rev' M M' σ (σ'#tr) σ''"
```

```
inductive
internal_exec_rev ::
  "Module ⇒ Module ⇒ Config ⇒ (Config list) ⇒ Config ⇒ bool"
  ("_;_,_ →eir⟨_⟩ _") for M :: Module and M' :: Module
  where
internal_exec_rev_first_step:
 "link_wf M M' ⟹
  (M ∘l M'), σ →e σ' ⟹
  this_class_lookup σ = Some c ⟹ c ∈ dom M' ⟹
  this_class_lookup σ' = Some c' ⟹ c' ∈ dom M ⟹
  internal_exec_rev' M M' σ' tr σ'' ⟹
  internal_exec_rev M M' σ (σ'#tr) σ''"
```

Finally, we conclude that `internal_exec_det` is deterministic. It is one of the main lemmas needed to show that the visible-states semantics is deterministic.

Lemma `internal_exec_det` says that in the first `n - 2` steps of execution if for the same initial state σ, there is at most one next state that is reached after one step of execution, i.e., if σ steps to σ' and also to v, then $v = \sigma'$.

```
lemma internal_exec_det:
 "M;M', σ →ei⟨tr⟩ σ' ⟹ M;M', σ →ei⟨tr⟩ v ⟹  v = σ'"
  by (auto simp: internal_exec_det_aux)
```

Lemma `visible_exec_det` asserts that the execution of the visible-states semantics is deterministic as below. Similar to Lemma `internal_exec_det`, Lemma `visible_exec_det` also says that if σ steps to σ' and also to σ'', then $\sigma'' = \sigma'$, but in this case, it is the final step from $\sigma_{(n-1)}$ to σ_n.

```
lemma visible_exec_det:
  "M;M', σ →e σ' ⟹ link_wf M M' ⟹ M;M', σ →e σ'' ⟹ σ'' = σ'"
  by (auto simp: visible_exec_det_aux)
```

The lemma `exec_det` below asserts that the execution of the languages is deterministic. The proof is by structural induction on the definition of `exec`.

```
lemma exec_det:
  "M, σ →e σ' ⟹ M, σ →e σ'' ⟹ σ'' = σ'"
  by (auto simp: exec_det_aux)
```

We make the proof details of the determinism of language and visible-states at the lemma `exec_det_aux` and lemma `visible_exec_det_aux`, respectively, in Appendix A.2.

3.4.2 Linking modules preserving execution

Intuitively, taking a module *M* and placing it in a larger context *M'* cannot reduce the behaviors of *M*. Therefore, if *M* can perform some execution step on its own, we would expect it also to perform that same step when linked against an arbitrary module *M'*. We formally prove this below.

A similar argument also applies to the visible state semantics. If *M* when linked against *M'* has a visible execution, it should still have that same execution when linked against $M' \circ_l M''$. We formally prove this property also.

Together these properties tell us that linking is monotonic for a module's executions (i.e., increasing the context increases the possible executions but does not remove any), as should be expected.

In this section, we place proofs of module linking preserving execution. First, we need to define `link_wf_3M` for three modules whose domains are pairwise disjoint.

```
definition
link_wf_3M :: "Module ⇒ Module ⇒ Module ⇒ bool"
  where
"link_wf_3M M M' M'' ≡ ((dom M ∩ dom M' = {}) ∧
                          (dom M' ∩ dom M'' = {}) ∧
                           (dom M'' ∩ dom M = {}))"
```

Technical lemma `link_dom` says that the union of two domains of two modules `M` and `M'` is the two's union domain.

```
lemma link_dom [simp]:
  "dom (M ∘l M') = dom M ∪ dom M'"
  by (auto simp: moduleLinking_def moduleAux_def dom_def)
```

We also need technical lemma `link_wf_3M_dest`, saying that if three modules whose domains are pairwise disjoint are well-formed, pair arbitrary modules are also well-formed.

```
lemma link_wf_3M_dest [simp,intro,dest]:
      "link_wf_3M M M' M'' ⟹ link_wf M M'"
      "link_wf_3M M M' M'' ⟹ link_wf M' M''"
      "link_wf_3M M M' M'' ⟹ link_wf M M''"
      "link_wf_3M M M' M'' ⟹ link_wf (M ∘l M') M''"
  by(fastforce simp: link_wf_def link_wf_3M_def)+
```

Then, we put the lemma `link_exec` to show that the module linking preserves one-module if its module linking is defined.

```
lemma link_exec:
  "⟦ M, σ →e σ'; link_wf M M' ⟧ ⟹ (M ∘l M'), σ →e σ'"
  by (simp add: link_exec_aux)
```

Moreover, the module linking preserves visible state semantics also, as shown in lemmas `visible_exec_linking_1` and `visible_exec_linking_2`. Since the definition of visible-state semantics `visible_exec`, is based on the definitions of internal execution `internal_exec`, we need to prove that the internal execution is also preserved by linking as shown in lemmas `internal_linking_1` and `internal_linking_2` as well.
Proofs of lemma `internal_linking_1` and `internal_linking_2` are by structural induction on the definition of `internal_exec`.

The lemma `internal_linking_1` guarantees that `M; M',σ →ei⟨tr⟩ σ'` implies that all intermediate configurations are external to `M'` and thus also to `M' ∘l M''`.

```
lemma internal_linking_1:
  "⟦M; M',σ →ei⟨tr⟩ σ'; link_wf_3M M M' M''⟧ ⟹
         M;(M' ∘l M''), σ →ei⟨tr⟩ σ'"
  by (simp add: internal_linking_1_aux)
```

Similarly, the lemma `internal_linking_2` guarantees that `M; M',σ →ei⟨tr⟩ σ'` implies that all intermediate configurations are internal to `M` and thus also to `M ∘l M''`.

```
lemma internal_linking_2:
  "⟦ M; M',σ →ei⟨tr⟩ σ' ; link_wf_3M M M' M''⟧ ⟹
   (M ∘l M''); M', σ →ei⟨tr⟩ σ'"
  by (simp add: internal_linking_2_aux)
```

Thanks to two useful lemmas `internal_linking_1` and `internal_linking_1`, we also gain the guarantee of module linking preserves visible state semantics. Proofs of lemmas `visible_exec_linking_1` and `visible_exec_linking_2` are by structural induction on the definition of `visible_exec`.

```
lemma visible_exec_linking_1:
  "⟦(M;M',σ →e σ'); (link_wf_3M M M' M'')⟧ ⟹
    M; (M' ∘l M''), σ →e σ'"
  by (simp add: visible_exec_linking_1_aux)

lemma visible_exec_linking_2:
  "⟦(M;M',σ →e σ'); (link_wf_3M M M' M'')⟧ ⟹
    (M ∘l M''); M', σ →e σ'"
  by (simp add: visible_exec_linking_2_aux)
```

We make the proof details of Linking modules preserving execution in Appendix A.3.

3.5 Initial and Arising configurations

What does it mean for a holistic specification to hold for a module M when linked against some external module `M'`? It means that the property holds for all `Arising` configurations of `M` with `M'`. These are the configurations that can be reached in the visible state semantics of `M` with `M'` when execution begins from the initial, empty configuration. We formally define these ideas below.

Now, we assume that the initial stack frame maps no local variables. Note that we let the continuation be arbitrary.

```
definition
initial_frame :: "Frame ⇒ bool"
  where
"initial_frame φ ≡ (vars φ = Map.empty ∧ this φ = Null)"
```

Suppose we have defined the execution of more steps `exec_rtrancl`, which is the reflexive, transitive closure of `exec`. In that case, we also want to define the execution of more steps `exec_module`, saying that it is the reflexive, transitive closure of visible execution `exec`. We formally define the execution `exec_module` as follows.

```
inductive
exec_module ::
  "Module ⇒ Module ⇒ Config ⇒ Config ⇒ bool" ("_;_, _ →e* _")
  where
exec_module_equiv: "σ = σ' ⟹ M;M', σ →e* σ'" |
exec_module_trans: "⟦(M;M', σ →e* σ''); (M; M', σ'' →e σ')⟧ ⟹
                     M;M', σ →e* σ'"
```

Initial configurations `Initial` might contain arbitrary code but no objects.

```
definition
Initial :: "(Stack × Heap) ⇒ bool"
  where
"Initial σ ≡ (case σ of (ψ,χ) ⇒
                  (case ψ of ([φ]) ⇒
                    initial_frame φ ∧ χ = Map.empty
                | _ ⇒ False))"
```

From initial configurations `Initial`, execution of code from module-pair `(M; M')`, creates a set of Arising configurations `Arising`.

```
definition
Arising :: "Module ⇒ Module ⇒ Config set "
  where
"Arising M M' ≡ {σ.∃σ0. (Initial σ0 ∧ (M;M', σ0 →e* σ))}"
```

Notice that $M;M',\ \sigma_0 \rightarrow_e^* \sigma$ is `visible-state semantics` introduced in Section 3.4.

3.6 Assertions - Classical Assertions

We have defined the object-based programming language and its semantics, including the visible state semantics, and proved them deterministic. However, we have not yet defined the language in which holistic specifications are expressed. We now do that by formally defining the assertions of holistic specifications and giving them meaning over the visible state semantics of the programming language defined above. We give the formalization of syntax and semantics of holistic assertions in this section.

3.6.1 Syntax of Assertions and its standard semantics

The validity of assertions `Assertion` has a form of `M; M',σ ⊨ A` where the module `M` and `M'` are internal and external, respectively. The assertion returns a `bool option` type rather than a `bool`. For instance, if we compare two expressions `e` and `e'` and one of them evaluates to `None`, then the semantics of the comparison is undefined. Hence the semantics of assertions is partial, represented using the option type as with other partial functions.

Unlike the syntax of the programming language, which is deeply embedded, we decided to embed assertions in our formalization shallowly. It was done to enable us to extend the set of assertions, later on, more efficiently.

```
type_synonym Assertion = "Module ⇒ Module ⇒ Config ⇒ bool option"
```

Assertions consist of pure expressions such as `atrue` and `afalse`.

```
datatype Expr =  ENull | EId Identifier | EField Expr FieldName
```

```
definition
atrue :: "Assertion"
  where
"atrue  ≡ λM M' σ. Some True"
```

```
definition
afalse :: "Assertion"
  where
"afalse  M M' σ ≡ Some False"
```

Expressions support nested field lookups, e.g., `x.f.g` via `(EField (EField (EId x) f) g)`.

```
fun
evalVar :: " Identifier ⇒ Config ⇒ Value option"
  where
"evalVar x (φ#ψ,χ) =  ident_lookup φ x" |
"evalVar x ([],χ)  =  None"
```

Recall that expressions denote values. We, therefore, define the semantics of expressions via the following partial function. Note that expression might not evaluate a value, e.g., for a field lookup for a non-existent object, in which case the semantics returns `None`. Otherwise, it returns `Some v`, where `v` is the value the expression denotes, in configuration σ.

primrec
```
expr_eval :: "Expr ⇒ Config ⇒ Value option"
  where
"expr_eval ENull σ = Some (VAddr Null)" |
"expr_eval (EId x) σ = evalVar x σ" |
"expr_eval (EField e f) σ =
     (case (expr_eval e σ) of Some (VAddr a) ⇒
           field_lookup (snd σ) a f)"
```

We define generic comparisons between expressions. For example, the notation of *greater than* would be expressed as `acompare (>) e e'`.

definition
```
acompare :: "(Value ⇒ Value ⇒ bool) ⇒ Expr ⇒ Expr ⇒ Assertion"
  where
"acompare c e e' ≡ λM M' σ.
    (case (expr_eval e σ) of Some v ⇒
        (case (expr_eval e' σ) of Some v' ⇒ Some (c v v')
                                 | None ⇒ None)
                           | None ⇒ None)"
```

We give formalized definitions of the semantics of assertions involving expressions. The partial function `expr_class_lookup` is used to look up the class where expression `e` is located in the runtime configuration σ.

fun
```
expr_class_lookup :: "Config ⇒ Expr ⇒ ClassName option"
  where
"expr_class_lookup σ e =
   (case σ of (φ#ψ,χ) ⇒
       (case (expr_eval e σ) of Some (VAddr a) ⇒
            (case χ a of Some obj ⇒ Some (className obj) |  None ⇒ None ))
            | _ ⇒ None)"
```

The assertion `aExpClassId` states whether an expression `e` belongs to a class identifier `ClassId`.

```
definition
aExpClassId :: "Expr ⇒ ClassName ⇒ Assertion"
  where
"aExpClassId e ClassId ≡
     λM M' σ. (case (expr_class_lookup σ e) of None ⇒ None
                                              | Some cid ⇒ Some (cid = ClassId))"
```

The function `expInS` checks the address of the expression `e` is in the set of addresses of the given set `S`.

```
fun
expInS :: "Config ⇒ Expr ⇒ Identifier ⇒ bool option"
  where
"expInS σ e S =
   (case (expr_eval e σ) of Some (VAddr a) ⇒
                (case (evalVar S σ) of Some v ⇒
                       (case v of VAddr addr ⇒ None |
                            VAddrSet addrSet ⇒ (Some (a ∈ addrSet))) |
                 None ⇒ None) |
         None ⇒ None)"
```

The assertion `aExpInS` presents whether an expression `e` belongs to a given set `S`.

```
definition
aExpInS :: "Expr ⇒ Identifier ⇒ Assertion"
  where
"aExpInS e S  ≡ λ M M' σ. (expInS σ e S)"
```

We formalize the meaning of standard logical connectives between assertions. For example, the logical conjunction of two assertions `A` and `A'` is expressed as `aAnd A A'`. Similarly, the logical disjunction, negation, and implication are expressed as `aOr A A'`, `aNot A`, and `aImp A A'`, respectively.

To support such assertions, we define a generic binary operator between assertions. For example, the notation of *implication* would be expressed as `bopt (⟶) (A M M' σ) (A' M M' σ)`.

definition

```
bopt ::
"(bool ⇒ bool ⇒ bool) ⇒ bool option ⇒ bool option ⇒ bool option"
  where
"bopt f a b ≡
   (case a of Some a' ⇒
       (case b of Some b' ⇒ Some (f a' b')
                    | None ⇒ None)
               | None ⇒ None)"
```

definition

```
aImp :: "Assertion ⇒ Assertion ⇒ Assertion"
  where
"aImp A A' ≡ λM M' σ. bopt (⟶) (A M M' σ) (A' M M' σ)"
```

definition

```
aAnd:: "Assertion ⇒ Assertion ⇒ Assertion"
  where
"aAnd A A' ≡ λM M' σ. bopt (∧) (A M M' σ) (A' M M' σ)"
```

definition

```
aOr:: "Assertion ⇒ Assertion ⇒ Assertion"
  where
"aOr A A' ≡ λM M' σ. bopt (∨) (A M M' σ) (A' M M' σ)"
```

definition

```
aNot:: "Assertion  ⇒ Assertion"
  where
"aNot A  ≡λM M' σ. case (A M M' σ) of None ⇒ None | Some a' ⇒ Some (¬ a')"
```

We also give the universal and existential quantification for holistic assertions. The universal quantification is expressed formally as `aAll fA`, and the existential quantification is presented as `aEx fA`. We represent an assertion like `∀x. P x`, by having P be a function that takes the identifier x as an argument and returns an assertion. It is an instance of Higher-Order Abstract Syntax.

definition

```
aAll :: "(Identifier ⇒ Assertion) ⇒ Assertion"
  where
"aAll fA ≡ λM M' σ. (if (∃v'. fA v' M M' σ = None)
                    then None
                    else Some (∀v. the (fA v M M' σ)))"
```

It is similar to an assertion like ∃x. P x.

definition
```
aEx :: "(Identifier ⇒ Assertion) ⇒ Assertion"
  where
"aEx fA ≡ λM M' σ. (if (∃v'. fA v' M M' σ = None)
                    then None
                    else Some (∃v. the (fA v M M' σ)))"
```

3.6.2 Properties of classical logic

Here, we deliver formal proofs of properties that apply to conjunction, disjunction, negation, implication, universal quantification, as well as existential quantification to show that holistic assertions are classical. Remember that assertions are partial. These properties hold only for well-formed assertions whose semantics are not undefined. We capture this formally via the definition `Assert_wf` below.

definition
```
Assert_wf:: "Assertion ⇒ Module ⇒ Module ⇒ Config ⇒ bool"
  where
"Assert_wf A M M' σ  ≡ A M M' σ ≠ None"
```

We are taking an example to show that holistic assertions are distributive property `aDistributive_1` of logical conjunction over logical disjunction for assertion `A`, `A'` and `A''`. Other properties can be found in Section A.5.

lemma `aDistributive_1:`
```
  "⟦Assert_wf A M M' σ;  Assert_wf A' M M' σ; Assert_wf A'' M M' σ ⟧ ⟹
    (aAnd (aOr A  A') A''  M M' σ) = (aOr (aAnd A A'') (aAnd A' A'') M M' σ)"
  unfolding aAnd_def aOr_def bopt_def option.case_eq_if by auto
```

3.7 Assertions - Access, Control, Space, Authority, and Viewpoint

In this section, we focus on the formalization of holistic concepts. These consist of permission, control, space, authority, and viewpoint.

3.7.1 Access

Access or permission states an object has a direct path to another object. In more detail, in the current frame, the access assertions are defined as three cases: (1) two objects are aliases, (2) the first points to an object with a field whose value is the same as the second object, (3) the first object is currently executing an object, and the second object is a local parameter that appears in the code in the continuation.

In particular, we formalize access assertion `Access` with supporting functions `thisEval` and `evalThis`. The partial function `thisEval` is used to look up the address of `this` object in runtime configuration σ. The function returns `None` in case of lookup failure.

```
fun
thisEval :: "Config ⇒ Value option"
  where
"thisEval σ =
   (case σ of (φ#_, _) ⇒
        (case (this φ) of
                addr ⇒ Some (VAddr addr))
            | _ ⇒ None)"
```

The partial function `evalThis` is used to look up the field `f` from `this` object in runtime configuration σ. The function returns `None` in case of lookup failure.

```
fun
evalThis :: "FieldName ⇒ Config ⇒ Value option"
  where
"evalThis f σ =
    (case σ of (ψ,χ) ⇒
          (case ψ of [] ⇒ None
               | (φ#ψ) ⇒ this_field_lookup φ χ f))"
```

The function `ConfigCont` obtains the continuation `cont` from the configuration σ.

```
definition
ConfigCont :: "Config ⇒ Continuation"
  where
"ConfigCont σ ≡ (case σ of (φ#_, _) ⇒ cont φ)"
```

The assertion `Access x y` holds if in runtime configuration σ,

(1) the value of identifier `x` and `y` is the same, or

(2) there exists a field `f` such that the value of the field `f` and the identifier `y` is the same, or

(3) the value of identifier `x` and the `this` object is the same, as well as the value of identifier `y` and `z` is also the same, where `z` is a local parameter in continuation `cont`.

```
definition
Access:: "Identifier ⇒ Identifier ⇒ Assertion"
  where
"Access x y ≡ λM M' σ.
                    if (evalVar x σ = None ∨ evalVar y σ = None)
                    then None
                    else Some ( (evalVar x σ = evalVar y σ) ∨
                          (∃f. (evalThis f σ = evalVar y σ)) ∨
                          ((evalVar x σ = thisEval σ) ∧
                          (∃z z1 stmts. ((Code stmts =  ConfigCont σ) ∨
                          (NestedCall z1 stmts =  ConfigCont σ)) ∧
                          (z ∈  stmts_idents stmts) ∧
                          (evalVar y σ = evalVar z σ))))"
```

3.7.2 Control

Control assertion represents the object making a function call on another object. We give a formalized definition of `Calls`, which goes along with supporting functions: `idents_list_undef` and `idents_list_equal`.

Since the control assertion has a form `x calls y.m(params)`, and the identifier `x`, `y`, as well as all elements in the list `params` should be defined, we give an auxiliary function `idents_list_undef` to check for any undefined identifiers in the list of identifiers, given the configuration σ.

```
fun
idents_list_undef:: "Identifier list ⇒ Config ⇒ bool"
  where
"idents_list_undef [] σ = False" |
"idents_list_undef (x#xs) σ =
                ((evalVar x σ = None) ∨ (idents_list_undef xs σ))"
```

Then, the function `idents_list_equal` checks that the value of each element of the first and second identifier list are equal in runtime configuration σ.

```
fun
idents_list_equal:: "Identifier list ⇒ Identifier list ⇒ Config ⇒ bool"
  where
"idents_list_equal [] [] σ = True" |
"idents_list_equal (z#zs) [] σ = False" |
"idents_list_equal [] (v#vs) σ = False" |
"idents_list_equal (z#zs) (v#vs) σ =
       ((evalVar z σ = evalVar v σ) ∧ (idents_list_equal zs vs σ))"
```

The assertion `Calls x y m zs` holds if in runtime configuration σ,

(1) the identifier x, y and all identifiers in list `zs` are defined, and

(2) the address of `this` object equals to the value of the caller x

(3) there are a receiver u and arguments `vs` with the same method m in runtime configuration σ such that the value of the identifier y and u is the same, and the value of each element of the list `zs` and `vs` is equal.

```
definition
Calls ::
  "Identifier ⇒ Identifier ⇒ MethodName ⇒ Identifier list ⇒ Assertion"
  where
"Calls x y m zs ≡ λM M' σ.
           if (evalVar x σ = None ∨ evalVar y σ = None ∨
              (idents_list_undef zs σ))
           then None
           else
                Some (( thisEval σ = evalVar x σ) ∧
                (∃ a u vs stmts.(ConfigCont σ =
                Code (Seq (MethodCall a u m vs) stmts)) ∧
                (evalVar y σ = evalVar u σ) ∧ idents_list_equal zs vs σ))"
```

3.7.3 Viewpoint

Viewpoint assertion represents whether an object belongs to the internal or external module. The formalization of `Internal` and `External` can be found as follows.

First, we define a function `Ident_class_lookup` to look up the class where identifier x is located in runtime configuration σ. The function returns `None` in case of failure.

fun
```
Ident_class_lookup :: "Config ⇒ Identifier ⇒ ClassName option"
  where
"Ident_class_lookup σ x =
   (case σ of (φ#ψ,χ) ⇒
       (case (evalVar x σ) of
                Some (VAddr a) ⇒ (case χ a of
                  Some obj ⇒ Some (className obj)
                | None ⇒ None ))
            | _ ⇒ None)"
```

Then, the assertion `External x` holds if the object `x` is outside the scope of module `M` in configuration σ.

definition
```
External :: "Identifier ⇒ Assertion"
  where
"External x ≡ λ M M' σ.
              (case (Ident_class_lookup σ x) of
                    None   ⇒ None |
                    Some c ⇒ Some (c ∉ dom M))"
```

Otherwise, the assertion `External x` asserts that the object `x` is in module `M` in runtime configuration σ.

definition
```
Internal :: "Identifier ⇒ Assertion"
  where
"Internal x ≡ λ M M' σ.
              (case (Ident_class_lookup σ x) of
                    None ⇒ None  |
                  Some c ⇒ Some (c ∈ dom M))"
```

3.7.4 Space

To define space assertion, we give a function `restrictConf` to create a new heap with only objects in the given set `S` in runtime configuration σ.

definition
```
hRst :: "Heap ⇒ Config ⇒ Identifier ⇒ Heap"
```

```
  where
"hRst χ σ S  ≡
     λa. (case (evalVar S σ)  of
              None ⇒ None |
              Some v ⇒ (case v of VAddr addr ⇒ None |
                     VAddrSet addrSet ⇒ if a ∈ addrSet
                                         then χ a
                                         else None ))"
```

The function *restrictConf* updates the new heap *χ'* after restricting the given set *S* in runtime configuration *σ*.

```
definition
restrictConf :: "Identifier ⇒ Config ⇒ Config option"
  where
"restrictConf S σ ≡
  (case σ of (ψ, χ) ⇒ (let χ' = (hRst χ σ S) in
                        Some (ψ, χ')))"
```

```
definition
transConf :: "(Config ⇒ Config option) ⇒ Assertion ⇒ Assertion"
  where
"transConf transf A  ≡
     λM M' σ. (case transf σ of None ⇒ None | Some b ⇒ A M M' b)"
```

Thanks to the restriction operator *restrictConf*, we obtain the semantics of the space assertion *In*.

```
definition
In :: "Identifier ⇒ Assertion ⇒ Assertion"
  where
"In S ≡ transConf (restrictConf S)"
```

After having the definition of space assertion *In*, we also provide lemmas related to spatial connective assertions. Lemma *Distrib_In* proves the distributive property of space assertion over logical implication.

```
lemma Distrib_In:
  "(aImp (In S A) (In S B) M M' σ) =
       (In S(aImp A B)) M M' σ"
  by (simp add: In_def aImp_def bopt_def option.case_eq_if transConf_def)
```

Also, the lemma `not_In` demonstrates the negation property of space assertion.

```
lemma not_In:
  "aNot (In S A) M M’ σ = (In S (aNot A)) M M’ σ"
proof -
  have "∀f fa p fb fc. aNot (transConf fc fb) fa f p =
          transConf fc (aNot fb) fa f p ∨ fc p = None"
    using aNot_def transConf_def
    by force
  thus ?thesis
    using In_def aNot_def transConf_def
    by fastforce
qed
```

3.7.5 Adaptation on runtime configurations

This section is the most challenging part of giving a formalization. Thus, to define whether a runtime configuration satisfies a time assertion, we need to adapt a runtime configuration to another to deal with time.

To cope with the time concept, we encounter some challenges: (a) the validity of assertions in the future must be evaluated in the future configuration but utilizing the current configuration's bindings. For example, the assertion `Will(x.f = 1)` is satisfied if the field `f` of the object pointed at by `x` in the current configuration has the value 1 in some future configuration. Note that `x` may be pointing to a different object in the future configuration or may no longer be in scope. Therefore, the operator ◁ is used to combine runtime configurations. In particular, `σ ◁ σ’` adapts the following configuration to the view of top frame view of the former, returning a new one whose stack has the top frame as received from σ and where the `cont` has been renamed consistently, while the heap is taken from `σ’`. It permits to interpret expressions in the newer configuration `σ’` but with the variables tied in keeping with the top frame from σ.

The second obstacle we need to grab is that (b) the current configuration requires to store the code executed to determine future configurations. We cope with it by storing the residual code in the continuation in each frame.

Next, (c) we do not desire to observe configurations beyond the frame at the top of the stack. We handle it by only getting the top of the frame as pondering future executions.

We give a formalized definition of adaptation on runtime configuration as `adaptation`. In the definition, we need support from `adapt_frame σ σ'`, returning a new frame that consists of

(1) A new continuation: it is the same as the continuation of the configuration `σ'`. However, we replace all variables `zs` with fresh names `zs'` using `cont_subst_list (cont φ') zs zs'`. The set `zs'` comes from `fresh_idents (dom (vars φ)) zs`, which is a function generating a list of fresh identifiers where none of the new identifiers appear in `dom (vars φ)` or `zs`.

(2) A combination of the variable map from the configuration `σ` with the variable map from the configuration `σ'` through the renaming `vars φ(zs' [↦] map (λz. the (vars φ' z)) zs)`.

We present all additional definitions and lemmas to formalize adaptation on runtime configurations in Appendix A.4.

The function `cont_subst_list` replaces all variables `zs` with fresh names `zs'`.

```
fun
cont_subst_list ::
"Continuation ⇒ Identifier list ⇒ Identifier list ⇒ Continuation"
  where
"cont_subst_list (Code stmts) zs zs' =
      (Code (stmts_subst_list stmts zs zs'))" |
"cont_subst_list (NestedCall x stmts) zs zs' =
      (NestedCall (ident_subst_list x zs zs') (stmts_subst_list stmts zs zs'))"
```

```
definition
adapt_frame :: "Frame ⇒ Frame ⇒ Frame"
  where
"adapt_frame φ φ' ≡
  (let zs = sorted_list_of_set (dom (vars φ'));
       zs' = fresh_idents (dom (vars φ)) zs;
       contn'' = cont_subst_list (cont φ') zs zs' ;
       vars'' = map_upds (vars φ) zs' (map (λz. the ((vars φ') z)) zs) in
       ⦇cont = contn'', vars = vars'', this = (this φ')⦈)"
```

The operator ◁ denotes adaptation between two runtime configurations, defined in function `adaptation` below.

definition

```
adaptation :: "Config ⇒ Config ⇒ Config option" (" _ ◁ _")
  where
"σ ◁ σ' ≡ (case σ of (φ#_,_) ⇒
             (case σ' of (φ'#ψ',χ') ⇒
               let φ'' = adapt_frame φ φ' in
                Some (φ''#ψ',χ') | _ ⇒ None)
                      | _ ⇒ None)"
```

3.7.6 Time

With full support from the definition of Adaptation `adaptation`, we define *Next* assertion `Next` and *Will* assertion `Will`.

The function `next_visible`, a partial function, is used to reach the next state of visible-state semantics if it exists.

definition

```
next_visible :: "Module ⇒ Module ⇒ Config ⇒ Config option"
  where
"next_visible M M' σ ≡
        if (∃σ'. (M;M', σ →e σ'))
        then Some (THE σ'. (M;M', σ →e σ'))
        else None"
```

Similarly, the function `will_visible` is used to reach the future state of visible-state semantics when it existed.

definition

```
will_visible :: "Module ⇒ Module ⇒ Config ⇒ Config option"
  where
"will_visible M M' σ ≡
        if (∃σ'. (M;M', σ →e* σ'))
        then Some (THE σ'. (M;M', σ →e* σ'))
        else None"
```

The assertion `Next A` holds if `A` holds in some configuration `σ'` which arises from execution φ, where φ is the top frame of σ. By requiring that `next_visible M M' ([φ], χ)` rather than `next_visible M M' σ`, we are restricting the set of possible next configurations to those caused by the top frame.

definition

```
Next :: "Assertion ⇒ Assertion"
  where
"Next A ≡
   λM M' σ. (case σ of (φ#_,χ) ⇒
              (case (next_visible M M' ([φ],χ)) of Some σ' ⇒
                  (case (([φ],χ) ◁  σ') of
                     Some adpt ⇒ (A M M' adpt)|
                  None ⇒ None) |
              None ⇒ None)  |
            _  ⇒ None)"
```

Similar to the assertion `Next A`, we define the assertion `Will A`. It says that the assertion holds when `A` holds in some configuration σ' which arises from execution φ, where φ is the top frame of σ. However, it considers in more future steps instead of in the successive step.

definition

```
Will :: "Assertion ⇒ Assertion"
  where
"Will A ≡
   λM M' σ. (case σ of (φ#_,χ) ⇒
              (case (will_visible M M'([φ],χ) ) of Some σ' ⇒
                  (case (([φ],χ) ◁  σ') of
                     Some adpt ⇒ (A M M' adpt)|
                  None ⇒ None) |
              None ⇒ None)  |
            _  ⇒ None)"
```

3.7.7 Authority

Authority (*Changes*) assertion `Changes` is defined to give conditions for change to occur. The partial function `Changes` on expression `e` is used to assert the evaluation of expression `e` in the next configuration σ' distinguishes from one in the current configuration σ.

In particular, the `Changes e` says that there exists an expression `v` such that the value of the expression `v` and `e` is the same. However, the expression `v` and `e`'s values are no longer the same in the next configuration.

definition

```
Changes :: "Expr ⇒ Assertion"
  where
"Changes e ≡  λM M' σ.
              (case (next_visible M M' σ) of Some σ' ⇒
                  (case (σ ⊲ σ') of Some adpt ⇒
                      (case expr_eval e σ of Some v1 ⇒
                            (case expr_eval e adpt of Some v2 ⇒
                                     Some (v1 ≠ v2) |
                                                        None ⇒ None ) |
                                                None ⇒ None) |
                                          None ⇒ None) |
                                                None ⇒ None)"
```

3.7.8 Modules Satisfying Assertions

The section exhibits how we formally define whether a module satisfies an assertion `Module_sat`. Here, `Module_sat M A` holds when for all external modules `M'` and all `Arising` runtime configuration σ, the assertion `A` is satisfied. Note that all runtime configuration σ is observed as `Arising` configurations.

```
definition
Module_sat :: "Module ⇒ Assertion ⇒ bool"
  where
"Module_sat M A ≡ (∀ M' σ. (σ ∈ Arising M M') ⟶
                  (A M M' σ = None ∨ A M M' σ = Some True))"
```

3.8 Summary

So far, we have formalized the theory of holistic specifications and several proofs comprising about 1800 lines of code of definitions and proofs in Isabelle/HOL. We have also proved several lemmas related to the theory of holistic specifications, as follows.

1. *Execution in $\mathscr{L}_{oo}$ and visible states is deterministic.* The deterministic execution of the language $\mathscr{L}_{oo}$ is shown at Lemma `exec_det` in Section 3.4.1. Also, the deterministic execution of the visible states is proved at Lemma `visible_exec_det` in Section 3.4.1.

2. *Properties of Linking.* The linking is associative, and commutative can be found at Lemma `link_assoc` and Lemma `link_commute` in Section 3.3. Moreover, the linking preserves both one-module and two-module execution is shown at Lemma `visible_exec_linking_1` and Lemma `visible_exec_linking_2` in Section 3.4.2.

3. *Assertions are classical.* Lemmas of assertions related to standard logical operators and quantifiers ($\wedge$, $\vee$, $\rightarrow$, $\neg$, $\forall$ and $\exists$) are shown in Section 3.6.2. They include `aComplement_1`, `aComplement_2`, `aCommutative_1`, `aCommutative_2`, `aAssociative_1`, `aAssociative_2`, `aDistributive_1`, `aDistributive_2`, `aDeMorgan_1`, `aDeMorgan_2`, `aUniversal_existential_1`, `aUniversal_existential_2`, `aImplication`, and `aNeverHold`.

Chapter 4

Lemmas towards reasoning about Holistic specifications

This chapter paves a way to reason about Holistic specifications. First, we briefly consider an example to reflect that the traditional specification is insufficient to guarantee the code is secure. Next, we construct a new specification as a form of holistic specifications to make the code safer. Then, from the holistic specifications made earlier, we form lemmas and provide proofs and "pen-and-paper" proofs to establish the foundations for reasoning about holistic specifications.

Chapter Outline

- **Motivating example.** Section 4.1 provides the example of class `Wallet` and the construction of its specifications as forms of holistic specifications.
- **Lemmas for reasoning about holistic specifications.** Section 4.2 presents lemmas and proofs with hoping to place the foundations for reasoning about holistic specifications.

4.1 Motivating example

Let consider the code snippet from Figure 4.1. Class `Wallet` consists of a `balance` field, a `secret` field, and the method pay, which takes a role as the only holder of the secret, can use the `Wallet` to make payments – for the sake of simplicity, we allow balances to grow negative.

```
class Wallet {
   fld balance
   fld secret
   mthd pay(who, amt, scr) {
      if (secret == src) {
         balance = balance - amt
         who.balance = who.balance + amt
      }
   }
}
```

Listing 4.1 Example of class `Wallet` modified from class `Safe` [8].

We give a Hoare triple specification below detailing the behavior of the method pay.

$$
\begin{aligned}
&\texttt{method pay(who, amt, scr)}\\
&\quad\texttt{PRE:}\ (\texttt{this, who:Wallet}) \land (\texttt{this} \neq \texttt{who}) \land (\texttt{amt}:\mathbb{N}) \land (\texttt{scr = secret})\\
&\quad\texttt{POST:}\ (\texttt{this.balance} = \texttt{this.balance}_{\texttt{pre}} - \texttt{amt}) \land\\
&\qquad\qquad (\texttt{who.balance} = \texttt{who.balance}_{\texttt{pre}} + \texttt{amt})
\end{aligned}
\tag{4.1}
$$

The specification in Formula 4.1 shows that the `secret` is a sufficient condition to make a payment. Namely, if the secret succeeds in providing, then the pay method can perform a payment. Though, it does not show that the specification is a necessary condition. In the case the pre-condition is not satisfied, how does the traditional Hoare triple represent the behavior of the method pay? We also describe the behavior of the method pay if the pre-condition is not satisfied, as follows.

$$\begin{array}{rl}\texttt{method pay(who, amt, scr)} & \\ \texttt{PRE:} & (\texttt{this, who:Wallet}) \land \neg((\texttt{this} \neq \texttt{who}) \land (\texttt{amt}:\mathbb{N}) \land (\texttt{scr = secret})) \\ \texttt{POST:} & \forall \texttt{w:Wallet}.(\texttt{w.balance} = \texttt{w.balance}_{pre} - \texttt{amt})\end{array} \tag{4.2}$$

The specification in Formula 4.2 meets that if the secret fails to provide, then the method pay cannot make a payment. However, the specification cannot prevent some other classes (including Wallet) containing more methods, making it possible to affect a reduction in the balance without knowing the secret. To circumvent this, we form a new specification using the concepts of holistic specification presented below in Formula 4.3.

$$\begin{array}{l}\texttt{Spec} \triangleq \forall \texttt{w,m}.(\ \texttt{w:Wallet} \land \texttt{w.balance = m} \land \texttt{Will(w.balance} \neq \texttt{m)} \Longrightarrow \\ \qquad \exists \texttt{o}.(\texttt{External(o)} \land (\texttt{o Access w.secret})))\end{array} \tag{4.3}$$

The specification expresses that for any wallet w defined in the current configuration, if the balance of w were to change in the future, then no less than one external object in the current configuration has direct access to the secret.

Straightforwardly, the question here is when a module meets the holistic specification outlined in Formula 4.3. In particular, the module is a consideration module (a.k.a., internal module), which consists of the code of the class Wallet from Figure 4.1.

Drossopoulou et al. [8] in Section 4.1 answered that $\texttt{M} \models \texttt{Spec}$ holds if for all untrusted modules $\texttt{M}'$ and all Arising configurations σ of execution of code from module-pair $(\texttt{M},\texttt{M}')$, the Spec holds.

We have specified it formally as follows.

$$\texttt{M} \models \texttt{Spec if } \forall\, \texttt{M}'.\ \forall \sigma \in \mathit{Arising}(\texttt{M};\texttt{M}').[\texttt{M};\texttt{M}',\sigma \models \texttt{Spec}] \tag{4.4}$$

Based on definitions of holistic assertions defined in [8], we open Formula 4.4 with the Spec with purposes to go deeply and from that to produce lemmas or theorems to reasoning the Spec.

First, we expand $\texttt{M};\texttt{M}',\sigma \models \texttt{Spec}$. It is equivalent to the formula below.

$$\begin{aligned}\mathtt{M;M'},\sigma \models [\forall \mathtt{w,m}.(\ \mathtt{w:Wallet} \land \mathtt{w.balance\ =\ m} \land \mathtt{Will(w.balance} \neq \mathtt{m)} \implies \\ \exists \mathtt{o}.(\mathtt{External(o)} \land (\mathtt{o\ Access\ w.secret})))]\end{aligned} \tag{4.5}$$

We have the below formula by employing assertions directly with logical connectives and quantifiers.

$$\begin{gathered}\forall \mathtt{w,m}.\ (\mathtt{M;M'},\sigma \models \mathtt{w:Wallet}\ \land \mathtt{M;M'},\sigma \models (\mathtt{w.balance\ =\ m}) \land \\ \mathtt{M;M'},\sigma \models \mathtt{Will(w.balance} \neq \mathtt{m)} \\ \implies \exists \mathtt{o}.\big(\mathtt{M;M'},\sigma \models \mathtt{External(o)} \land \mathtt{M;M'},\sigma \models (\mathtt{o\ Access\ w.secret})\big)\end{gathered} \tag{4.6}$$

We produce Formula 4.7 using the definition of `Will` assertion. Note that the operator $\lhd$ denotes adaptation between two runtime configurations, formalized in Section 3.7.5.

$$\begin{gathered}\mathtt{M;M'},\sigma \models \mathtt{Will(w.balance} \neq \mathtt{m)} \\ \mathtt{if}\ \ \exists \sigma'.(\mathtt{M;M'},\sigma \rightarrow^*_{\mathtt{e}} \sigma' \land \mathtt{M;M'},\sigma \lhd \sigma' \models (\mathtt{w.balance} \neq \mathtt{m}))\end{gathered} \tag{4.7}$$

After having Formula 4.7, we produce Formula 4.8 from Formula 4.6 below.

$$\begin{gathered}\forall \mathtt{w,m}.\ (\mathtt{M;M'},\sigma \models \mathtt{w:Wallet}\ \land \\ \mathtt{M;M'},\sigma \models (\mathtt{w.balance\ =\ m}) \land (\exists \sigma'.(\mathtt{M;M'},\sigma \rightarrow^*_{\mathtt{e}} \sigma')) \land \\ \forall \mathtt{o}.\big(\mathtt{M;M'},\sigma \models (\mathtt{o\ Access\ w.secret}) \implies \mathtt{M;M'},\sigma \models \mathtt{Internal(o)}\big) \\ \implies \mathtt{M;M'},\sigma \lhd \sigma' \models (\mathtt{w.balance} = \mathtt{m})\end{gathered} \tag{4.8}$$

4.2 Lemmas for reasoning about holistic specifications

We understand that reasoning on temporal logic $\mathtt{Will}\langle \mathtt{A} \rangle$ assertion is difficult to reason about the future in more number steps time. Hence, we choose to reason in one step instead of in more number of steps.

To show that reasoning in one step is still sufficient, we have a technical Theorem 1. Assume that we have several finite steps, and instead of writing $\sigma_0 \rightarrow^*_{\mathtt{e}} \sigma_n$, we also assign a counter variable n. Let denote $\sigma_0 \rightarrow^n_{\mathsf{e}} \sigma_n$ as a path $\sigma_0 \rightarrow_{\mathsf{e}} \sigma_1 \rightarrow_{\mathsf{e}} \sigma_2 \rightarrow_{\mathsf{e}} \cdots \rightarrow_{\mathsf{e}} \sigma_k \cdots \rightarrow_{\mathsf{e}} \sigma_n$, where $1 \leq k \leq n$.

Theorem 1. *Let σ_0, σ, and σ' be an initial configuration, and arbitrary configurations respectively. Let S, A, and B be as follows.*

$$S := P(\sigma_0) \wedge (\sigma_0 \rightarrow^n_{\mathsf{e}} \sigma_n) \wedge \neg Q(\sigma_0) \implies \neg W(\sigma_n),$$

$$A := P(\sigma) \wedge (\sigma \rightarrow_{\mathsf{e}} \sigma') \wedge \neg Q(\sigma) \implies P(\sigma') \wedge \neg Q(\sigma'),$$

and

$$B := P(\sigma) \wedge (\sigma \rightarrow_{\mathsf{e}} \sigma') \wedge \neg Q(\sigma) \implies \neg W(\sigma').$$

We have $A \wedge B \vdash S$.

Proof. We show the lemma using induction on n.

- **Case** $n = 1$**.**

  We need to prove $S_1 := P(\sigma_0) \wedge (\sigma_0 \rightarrow^1_{\mathsf{e}} \sigma_1) \wedge \neg Q(\sigma_0) \implies \neg W(\sigma_1)$.
  Since $\sigma_0 \rightarrow_{\mathsf{e}} \sigma_1$, we have
  $A_1 := P(\sigma_0) \wedge (\sigma_0 \rightarrow_{\mathsf{e}} \sigma_1) \wedge \neg Q(\sigma_0) \implies P(\sigma_1) \wedge \neg Q(\sigma_1)$, and
  $B_1 := P(\sigma_0) \wedge (\sigma_0 \rightarrow_{\mathsf{e}} \sigma_1) \wedge \neg Q(\sigma_0) \implies \neg W(\sigma_1)$.
  S_1 is shown obviously from B_1.

- **Case** $n = 2$**.**

  We need to prove that $S_2 := P(\sigma_0) \wedge (\sigma_0 \rightarrow^2_{\mathsf{e}} \sigma_2) \wedge \neg Q(\sigma_0) \implies \neg W(\sigma_2)$.
  From the left-hand side (LHS) of S_2, we have $\sigma_0 \rightarrow^2_{\mathsf{e}} \sigma_2$,
  meaning $\sigma_0 \rightarrow_{\mathsf{e}} \sigma_1 \rightarrow_{\mathsf{e}} \sigma_2$.
  We have the transition from $\sigma_0 \rightarrow_{\mathsf{e}} \sigma_1$; therefore, we have A_1 and B_1.
  Now, let consider a transition from $\sigma_1 \rightarrow_{\mathsf{e}} \sigma_2$.
  Combining $\sigma_1 \rightarrow_{\mathsf{e}} \sigma_2$ and A, B, and the right-hand side (RHS) of A_1, we have
  $P(\sigma_1) \wedge (\sigma_1 \rightarrow_{\mathsf{e}} \sigma_2) \wedge \neg Q(\sigma_1) \implies P(\sigma_2) \wedge \neg Q(\sigma_2)$, and
  $P(\sigma_1) \wedge (\sigma_1 \rightarrow_{\mathsf{e}} \sigma_2) \wedge \neg Q(\sigma_1) \implies \neg W(\sigma_2)$.
  From that, we can conclude
  $A_2 := P(\sigma_0) \wedge (\sigma_0 \rightarrow^2_{\mathsf{e}} \sigma_2) \wedge \neg Q(\sigma_0) \implies P(\sigma_2) \wedge \neg Q(\sigma_2)$,
  and $B_2 := P(\sigma_0) \wedge (\sigma_0 \rightarrow^2_{\mathsf{e}} \sigma_2) \wedge \neg Q(\sigma_0) \implies \neg W(\sigma_2)$.
  Similarly, S_2 is shown from B_2.

- **Assume the case** $n = k$ **is true**, we have
  $S_k := P(\sigma_0) \wedge (\sigma_0 \rightarrow^k_{\mathsf{e}} \sigma_k) \wedge \neg Q(\sigma_0) \implies \neg W(\sigma_k)$.

  Since the formula is true with the path from $\sigma_0 \rightarrow_{\mathsf{e}} \sigma_1 \rightarrow_{\mathsf{e}} \sigma_2 \rightarrow_{\mathsf{e}} \cdots \rightarrow_{\mathsf{e}} \sigma_k$,
  we also have $A_k := P(\sigma_0) \wedge (\sigma_0 \rightarrow^k_{\mathsf{e}} \sigma_k) \wedge \neg Q(\sigma_0) \implies P(\sigma_k) \wedge \neg Q(\sigma_k)$, and
  $B_k := P(\sigma_0) \wedge (\sigma_0 \rightarrow^k_{\mathsf{e}} \sigma_k) \wedge \neg Q(\sigma_0) \implies \neg W(\sigma_k)$.
  We need to prove with the case $n = k + 1$.

- **Case** $n = k + 1$**.**

  We need to prove that $S_{k+1} := P(\sigma_0) \wedge (\sigma_0 \rightarrow^{k+1}_{\mathsf{e}} \sigma_{k+1}) \wedge \neg Q(\sigma_0) \implies \neg W(\sigma_{k+1})$.
  There is a transition from $\sigma_k \rightarrow_{\mathsf{e}} \sigma_{k+1}$.
  Combining $\sigma_k \rightarrow_{\mathsf{e}} \sigma_{k+1}$ and A, B, and RHS of A_k, we have
  $P(\sigma_k) \wedge (\sigma_k \rightarrow_{\mathsf{e}} \sigma_{k+1}) \wedge \neg Q(\sigma_k) \implies P(\sigma_{k+1}) \wedge \neg Q(\sigma_{k+1})$, and
  $P(\sigma_k) \wedge (\sigma_k \rightarrow_{\mathsf{e}} \sigma_{k+1}) \wedge \neg Q(\sigma_k) \implies \neg W(\sigma_{k+1})$.

  From that, we can conclude
  $A_{k+1} := P(\sigma_0) \wedge (\sigma_0 \rightarrow^{k+1}_{\mathsf{e}} \sigma_{k+1}) \wedge \neg Q(\sigma_0) \implies P(\sigma_{k+1}) \wedge \neg Q(\sigma_{k+1})$, and
  $B_{k+1} := P(\sigma_0) \wedge (\sigma_0 \rightarrow^{k+1}_{\mathsf{e}} \sigma_{k+1}) \wedge \neg Q(\sigma_0) \implies \neg W(\sigma_{k+1})$.
  Similarly, S_{k+1} is shown from B_{k+1}.

As a result, we have $A \wedge B \vdash S$. □

Thanks to Theorem 1, we only need to reason about the predicate A and B. Now, we rewrite Formula 4.8 as the form of the predicate A in Formula 4.9 as follows.

$$
\begin{gathered}
\forall \texttt{w,m}.\,(\texttt{M;M}',\sigma \models \texttt{w:Wallet} \;\wedge \\
\texttt{M;M}',\sigma \models (\texttt{w.balance = m}) \wedge (\exists \sigma'.(\texttt{M;M}',\sigma \rightarrow_{\mathsf{e}} \sigma')) \wedge \\
\forall \texttt{o}.\big(\texttt{M;M}',\sigma \models (\texttt{o Access w.secret}) \implies \texttt{M;M}',\sigma \models \texttt{Internal(o)}\big) \\
\implies \texttt{M;M}',\sigma \lhd \sigma' \models \texttt{w:Wallet} \;\wedge \texttt{M;M}',\sigma \lhd \sigma' \models (\texttt{w.balance = m}) \wedge \\
\forall \texttt{o}.\big(\texttt{M;M}',\sigma \lhd \sigma' \models (\texttt{o Access w.secret}) \implies \texttt{M;M}',\sigma \lhd \sigma' \models \texttt{Internal(o)}\big))
\end{gathered}
\tag{4.9}
$$

We also rewrite Formula 4.8 as the form of the predicate B in Formula 4.10 below.

$$
\begin{aligned}
&\forall \mathtt{w,m}.\ (\mathtt{M;M'},\sigma \models \mathtt{w:Wallet}\ \land \\
&\mathtt{M;M'},\sigma \models (\mathtt{w.balance\ =\ m}) \land (\exists \sigma'.(\mathtt{M;M'},\sigma \rightarrow_{\mathtt{e}} \sigma')) \land \\
&\forall \mathtt{o}.\big(\mathtt{M;M'},\sigma \models (\mathtt{o\ Access\ w.secret}) \implies \mathtt{M;M'},\sigma \models \mathtt{Internal(o)}\big) \\
&\implies \mathtt{M;M'},\sigma \lhd \sigma' \models (\mathtt{w.balance} = \mathtt{m})
\end{aligned}
\tag{4.10}
$$

Note that the runtime configuration σ' in Formula 4.9 and 4.10 is different from the runtime configuration σ' in Formula 4.8.

From Formula 4.9 and 4.10, we set out lemmas with goals to support reasoning these specifications. In a capability system, all access to a capability must derive from pre-existing access. In short, "only connectivity begets connectivity" [22]. This thesis states and proves the properties of "only connectivity begets connectivity" (Theorem 2 and Theorem 3). We also form some technical lemmas aiding to prove them. While we have finished proving Theorem 2, we have already proved a partial of Theorem 3.

Before going to the details of theorems, we want to remember high-level terms and some interpretations throughout the upcoming sections. To be convenient, we frequently shift among notations.

- Frames ϕ are mappings from identifiers $x$to values, where values include addresses.
- Heaps χ are mappings from addresses α to objects.
- Configurations σ are pairs (ψ, χ) where ψ is a list of stack frames ϕ and χ is the heap.
- Lookup of identifier x in runtime configuration σ is written $\lfloor x \rfloor_{\sigma}$. Lookup of identifier x in the stack frame ϕ is written $\lfloor x \rfloor_{\phi}$. Both $\lfloor x \rfloor_{\sigma}$ and $\lfloor o \rfloor_{\phi}$ is defined as same as $\phi(x)$.
- Lookup of field f from x object in runtime configuration σ is written $\lfloor x.f \rfloor_{\sigma}$. Result of $\lfloor x.f \rfloor_{\sigma}$ is a value v and $\chi(\phi(x)) = (C, fldMap)$, where $fldMap(f) = v$ with $fldMap$ is a mapping from field name to values.
- The operator $\lhd$ denotes adaptation between two runtime configurations, formalized in Section 3.7.5.
- The term σ.cont is used to obtain the continuation cont from runtime configuration σ.

Theorem 2 says that an object has a direct path to another object if and only if there is a direct path on the same objects in the next configuration. Note that the lookup of objects is the same as the lookup of other objects, in which the current configuration or the next configuration is a part.

Theorem 2. *Let σ' be a next configuration such that $M; M', \sigma \rightarrow_e \sigma'$.*
Let o_1, o_2 be identifiers such that $\lfloor o_1 \rfloor_\sigma = \lfloor \mathtt{this} \rfloor_\sigma$ and $\lfloor o_2 \rfloor_{\sigma'} = \lfloor \mathtt{this} \rfloor_{\sigma'}$.
Let $o_3,\ o_4$ be identifiers such that $\lfloor o_3 \rfloor_\sigma,\ \lfloor o_4 \rfloor_\sigma \notin \{\lfloor o_1 \rfloor_\sigma,\ \lfloor o_2 \rfloor_{\sigma'}\}$.
Also, let heaps be χ and χ' such that $\chi(\mathtt{this}) \notin dom(M)$ and $\chi'(\mathtt{this}) \notin dom(M)$.
Show that $\sigma \models \mathtt{Next}\ \ (o_3\ \ \mathtt{access}\ \ o_4) \iff \sigma \models (o_3\ \ \mathtt{Access}\ \ o_4)$.

Proof. To prove the lemma, we have two cases to consider as follows.

1. $\sigma \models \mathtt{Next}\ \ (o_3\ \ \mathtt{Access}\ \ o_4) \implies \sigma \models (o_3\ \ \mathtt{Access}\ \ o_4)$. We prove it in Lemma 3.

2. $\sigma \models (o_3\ \ \mathtt{Access}\ \ o_4) \implies \sigma \models \mathtt{Next}\ \ (o_3\ \ \mathtt{Access}\ \ o_4)$. We prove it in Lemma 4.

As a result, we have $\sigma \models \mathtt{Next}\ \ (o_3\ \ \mathtt{Access}\ \ o_4) \iff \sigma \models (o_3\ \ \mathtt{Access}\ \ o_4)$. □

Theorem 3 says that if an object has a direct path to another object in the next configuration, then there is a direct path on the same objects in the current configuration. In this case, the lookup of the first object is the same as the lookup of other objects. The current configuration or the next configuration is a part, and the second object is in the current configuration.

Theorem 3. *Let σ' be a next configuration such that $M; M', \sigma \rightarrow_e \sigma'$.*
Let o_i, where $i = 1, 2$ be identifers such that $\lfloor o_1 \rfloor_\sigma = \lfloor \mathtt{this} \rfloor_\sigma$, $\lfloor o_2 \rfloor_{\sigma'} = \lfloor \mathtt{this} \rfloor_{\sigma'}$, $o_2 \in dom(\sigma)$ and $o_k \in dom(\sigma)$.
Also, let heaps be χ and χ' such that $\chi(\mathtt{this}) \notin dom(M)$ and $\chi'(\mathtt{this}) \notin dom(M)$.
Show that $\sigma \models \mathtt{Next}\ \ (o_i\ \ \mathtt{access}\ \ o_k) \implies (\sigma \models (o_1\ \ \mathtt{Access}\ \ o_k)) \lor (\sigma \models (o_2\ \ \mathtt{Access}\ \ o_k))$.

Note that we have already proved a partial of Theorem 3. Therefore, we put the proofs in Appendix A.7.

Lemma 3 gives the “forwards” proof of Theorem 2, presented as follows.

Lemma 3. *Let σ' be a next configuration such that $M; M', \sigma \rightarrow_e \sigma'$.*
Let o_1, o_2 be identifiers such that $\lfloor o_1 \rfloor_\sigma = \lfloor this \rfloor_\sigma$ and $\lfloor o_2 \rfloor_{\sigma'} = \lfloor this \rfloor_{\sigma'}$.
Let o_3, o_4 be identifiers such that $\lfloor o_3 \rfloor_\sigma$, $\lfloor o_4 \rfloor_\sigma \not\in \{\lfloor o_1 \rfloor_\sigma, \lfloor o_2 \rfloor_{\sigma'}\}$.
Also, let heaps be χ and χ' such that $\chi(this) \not\in dom(M)$ and $\chi'(this) \not\in dom(M)$.
Show that $\sigma \models \mathtt{Next}\ (o_3\ \mathtt{Access}\ o_4) \implies \sigma \models (o_3\ \mathtt{Access}\ o_4)$.

Proof. To be convenient, we call LHS of the implication is $\sigma \models \mathtt{Next}\ (o_3\ \mathtt{Access}\ o_4)$, and RHS is $\sigma \models (o_3\ \mathtt{Access}\ o_4)$.
From LHS of the implication of the formula, $\sigma \models \mathtt{next}\ (o_3\ \mathtt{Access}\ o_4)$ holds if

$$(\sigma \triangleleft \sigma') \models (o_3\ \mathtt{Access}\ o_4) \quad (1)$$

Using the definition of Permission in [8] (formalized in Section 3.7.2 as well), we rewrite (1) as an equivalent form as follows.
$(\lfloor o_3 \rfloor_{\sigma \triangleleft \sigma'} = \lfloor o_4 \rfloor_{\sigma \triangleleft \sigma'}) \vee (\lfloor o_3.f \rfloor_{\sigma \triangleleft \sigma'} = \lfloor o_4 \rfloor_{\sigma \triangleleft \sigma'}) \vee$

$[(\lfloor o_3 \rfloor_{\sigma \triangleleft \sigma'} = \lfloor \mathtt{this} \rfloor_{\sigma \triangleleft \sigma'}) \wedge (\lfloor o_4 \rfloor_{\sigma \triangleleft \sigma'} = \lfloor \mathtt{z}' \rfloor_{\sigma \triangleleft \sigma'})]$, where $\mathtt{z}'$ appears in $(\sigma \triangleleft \sigma').\mathtt{cont}'$.

- **Case** $\lfloor o_3 \rfloor_{\sigma \triangleleft \sigma'} = \lfloor o_4 \rfloor_{\sigma \triangleleft \sigma'}$.

  We have $\lfloor o_3 \rfloor_{\sigma \triangleleft \sigma'}$ and $\lfloor o_4 \rfloor_{\sigma \triangleleft \sigma'}$ are defined in $\sigma \triangleleft \sigma'$.
  Also, we have $\lfloor o_3 \rfloor_{\sigma \triangleleft \sigma'} = \lfloor o_3 \rfloor_\sigma$, and $\lfloor o_4 \rfloor_{\sigma \triangleleft \sigma'} = \lfloor o_4 \rfloor_\sigma$, since o_3 and o_4 are defined in σ (See Lemma 7). Therefore, we have $\lfloor o_3 \rfloor_\sigma = \lfloor o_4 \rfloor_\sigma$ in σ, and it implies $o_3\ \mathtt{Access}\ o_4$ in σ. From that, we have RHS of the implication of the formula.

- **Case** $\lfloor o_3.f \rfloor_{\sigma \triangleleft \sigma'} = \lfloor o_4 \rfloor_{\sigma \triangleleft \sigma'}$, with some field f.
  The configuration $\sigma \triangleleft \sigma'$ and σ' use the same heap χ' (from Definition of Adaptation in [8]). We rewrite $\lfloor o_3.f \rfloor_{\sigma \triangleleft \sigma'} = v$ for some v, and $\chi'(\phi(o_3)) = (C, fldMap)$, where $fldMap(f) = v$. On the other hand, we have $\lfloor o_4 \rfloor_{\sigma \triangleleft \sigma'} = \lfloor o_4 \rfloor_\sigma$.
  Now we need to show that $\lfloor o_3.f \rfloor_\sigma = \lfloor o_3.f \rfloor_{\sigma \triangleleft \sigma'}$.
  Thus, we have $\chi'(\phi(o_3)) = \chi(\phi(o_3))$, since Lemma 6 and $\lfloor o_3 \rfloor_\sigma \neq \lfloor \mathtt{this} \rfloor_\sigma$. Therefore, $\lfloor o_3.f \rfloor_\sigma$ and $\lfloor o_3.f \rfloor_{\sigma \triangleleft \sigma'}$ are evaluated by the same value v.
  So, we have $\lfloor o_3.f \rfloor_\sigma = \lfloor o_4 \rfloor_\sigma$, and it implies $o_3\ \mathtt{Access}\ o_4$ in σ.
  Then, we also have RHS of the implication of the formula.

- **Case** $(\lfloor o_3 \rfloor_{\sigma \triangleleft \sigma'} = \lfloor \mathtt{this} \rfloor_{\sigma \triangleleft \sigma'}) \wedge (\lfloor o_4 \rfloor_{\sigma \triangleleft \sigma'} = \lfloor \mathtt{z}' \rfloor_{\sigma \triangleleft \sigma'})$, where $\mathtt{z}'$ appears in $(\sigma \triangleleft \sigma').\mathtt{cont}'$.
  Since $\sigma \triangleleft \sigma'$ has address $\phi'(\mathtt{this})$, we have $\lfloor \mathtt{this} \rfloor_{\sigma \triangleleft \sigma'} = \lfloor \mathtt{this} \rfloor_{\sigma'}$.

However, from $\lfloor o_3 \rfloor_\sigma$, $\lfloor o_4 \rfloor_\sigma \notin \{\lfloor o_1 \rfloor_\sigma, \lfloor o_2 \rfloor_{\sigma'}\}$, $\lfloor o_1 \rfloor_\sigma = \lfloor \texttt{this} \rfloor_\sigma$ and $\lfloor o_2 \rfloor_{\sigma'} = \lfloor \texttt{this} \rfloor_{\sigma'}$. There is a contradiction here.

As a result, we have $\sigma \models \texttt{Next}\ \ (o_3\ \ \texttt{Access}\ \ o_4) \implies \sigma \models (o_3\ \ \texttt{Access}\ \ o_4)$. □

Lemma 4 gives the "backwards" proof of Theorem 2, presented as follows.

Lemma 4. *Let σ' be a next configuration such that $M; M', \sigma \rightarrow_e \sigma'$.*
Let o_1, o_2 be identifiers such that $\lfloor o_1 \rfloor_\sigma = \lfloor this \rfloor_\sigma$ and $\lfloor o_2 \rfloor_{\sigma'} = \lfloor this \rfloor_{\sigma'}$.
Let o_3, o_4 be identifiers such that $\lfloor o_3 \rfloor_\sigma$, $\lfloor o_4 \rfloor_\sigma \notin \{\lfloor o_1 \rfloor_\sigma, \lfloor o_2 \rfloor_{\sigma'}\}$.
Also, let heaps be χ and χ' such that $\chi(this) \notin dom(M)$ and $\chi'(this) \notin dom(M)$.
Show that $\sigma \models (o_3\ \ Access\ \ o_4) \implies \sigma \models Next\ \ (o_3\ \ Access\ \ o_4)$.

Proof. To be convenient, we call LHS is $\sigma \models (o_3\ \ \texttt{Access}\ \ o_4)$,
and RHS is $\sigma \models \texttt{Next}\ \ (o_3\ \ \texttt{Access}\ \ o_4)$ of the implication.
From LHS of the implication of the formula, $\sigma \models (o_3\ \ \texttt{Access}\ \ o_4)$ holds if

$$\sigma \models (o_3\ \ \texttt{Access}\ \ o_4)$$

We rewrite the formula above as an equivalent form as follows.
$(\lfloor o_3 \rfloor_\sigma = \lfloor o_4 \rfloor_\sigma) \vee (\lfloor o_3.f \rfloor_\sigma = \lfloor o_4 \rfloor_\sigma) \vee [(\lfloor o_3 \rfloor_\sigma = \lfloor \texttt{this} \rfloor_{\sigma \lhd \sigma'}) \wedge (\lfloor o_4 \rfloor_\sigma = \lfloor \texttt{z}' \rfloor_\sigma)]$,
where $\texttt{z}'$ appears in $\sigma.\texttt{cont}$.

- **Case** $\lfloor o_3 \rfloor_\sigma = \lfloor o_4 \rfloor_\sigma$.
  We have $\lfloor o_3 \rfloor_{\sigma \lhd \sigma'} = \lfloor o_3 \rfloor_\sigma$, and $\lfloor o_4 \rfloor_{\sigma \lhd \sigma'} = \lfloor o_4 \rfloor_\sigma$, since o_3 and o_4 are defined in σ (See Lemma 7).
  Therefore, we have $\lfloor o_3 \rfloor_{\sigma \lhd \sigma'} = \lfloor o_4 \rfloor_{\sigma \lhd \sigma'}$ in $\sigma \lhd \sigma'$, and it implies o_3 $\texttt{Access}$ o_4 in $\sigma \lhd \sigma'$.
  From that, we have RHS of the implication of the formula.

- **Case** $\lfloor o_3.f \rfloor_\sigma = \lfloor o_4 \rfloor_\sigma$, with some field f.
  We rewrite $\lfloor o_3.f \rfloor_\sigma = v$ for some v, and $\chi(\phi(o_3)) = (C, fldMap)$,
  where $fldMap(f) = v$.
  On the other hand, since o_4 is defined in σ, we have $\lfloor o_4 \rfloor_\sigma = \lfloor o_4 \rfloor_{\sigma \lhd \sigma'}$.
  The configuration $\sigma \lhd \sigma'$ and σ' use the same heap χ'.
  Now we need to show that $\lfloor o_3.f \rfloor_\sigma = \lfloor o_3.f \rfloor_{\sigma \lhd \sigma'}$. Thus, we have $\chi(\phi(o_3)) = \chi'(\phi(o_3))$, since Lemma 6 and $\sigma(o_3) \neq \sigma(\texttt{this})$.
  Therefore, $\lfloor o_3.f \rfloor_\sigma$ and $\lfloor o_3.f \rfloor_{\sigma \lhd \sigma'}$ are evaluated by the same value v. So, we have $\lfloor o_3.f \rfloor_{\sigma \lhd \sigma'} = \lfloor o_4 \rfloor_{\sigma \lhd \sigma'}$, implying o_3 $\texttt{Access}$ o_4 in $\sigma \lhd \sigma'$. Then, we also have RHS of the implication of the formula.

- **Case** $(\lfloor o_3 \rfloor_\sigma = \lfloor \texttt{this} \rfloor_\sigma) \wedge (\lfloor o_4 \rfloor_\sigma = \lfloor \texttt{z}' \rfloor_\sigma)$, where $\texttt{z}'$ appears in σ.cont.

  We have $\lfloor o_3 \rfloor_\sigma \neq \lfloor \texttt{this} \rfloor_\sigma$ because $\lfloor o_3 \rfloor_\sigma \neq \lfloor o_1 \rfloor_\sigma$ and $\lfloor o_1 \rfloor_\sigma = \lfloor \texttt{this} \rfloor_\sigma$. There is a contradiction here.

As a result, we have $\sigma \models (o_3 \texttt{ Access } o_4) \implies \sigma \models \texttt{Next } (o_3 \texttt{ Access } o_4)$. □

Lemma 5, 6, 7, 8, and 9 are technical lemmas supporting to prove Theorem 2 and 3, showed as follows.

Lemma 5. *Let σ' be a next configuration such that $M; M', \sigma \rightarrow_e \sigma'$.*
Let $field\_update((C, fldMap), f, v) = (C, fldMap(f := v))$ be a function using to update a field.
Show that if $\chi'(\phi(this)) \neq \chi(\phi(this))$ and $\phi(this) \in dom(\chi)$,
then there exists a field f and an identifier x where identifier $x \in dom(\sigma)$ such that $\chi'(\phi(this)) = field\_update(\chi(\phi(this), f, \phi(x)))$.

Proof. By structural induction on operational semantics, we have

- **Case** `methCall_OS`**.** The heap χ is unchanged in the next configuration σ'.
  It means $\chi'(\phi(\texttt{this})) = \chi(\phi(\texttt{this}))$. Hence, the desired result is correct.

- **Case** `varAssgn_OS`**.** It is similar to the case of methCall_OS.

- **Case** `return_OS`**.** It is also similar to the case of methCall_OS.

- **Case** `fieldAssgn_OS`**.** The heap χ is only changed in σ' if there is an identifier x and a field f such that $\texttt{this}.f := x$ in σ.cont. In particular, if there is $\texttt{this}.f := x$ in σ.cont, the new heap χ' now is the heap χ at the address $\phi(\texttt{this})$, and the field f is now updated by $\phi(x)$.
  Formally, $\chi'(\phi(\texttt{this})) = \chi[\phi(\texttt{this}) \mapsto (C, fldMap[f \mapsto \phi(x)])]$. So the desired result is correct as well.

- **Case** `objCreate_OS`**.** The heap χ is also changed in σ'. However, the heap is only changed at a new address α, where $\chi(\alpha) = (C, f_1 \mapsto \phi(x_1), \ldots f_n \mapsto \phi(x_n))$. Since $a \notin dom(\chi)$, and $\phi(\texttt{this}) \in dom(\chi)$, we have $\alpha \neq \phi(\texttt{this})$. From that, we still have got $\chi'(\phi(\texttt{this})) = \chi(\phi(\texttt{this}))$. So, the desired result is correct.

As a result, the lemma is proved. □

Lemma 6. *Let $M, \sigma \rightarrow_e \sigma'$ be an execution. Also, let an address $a = \sigma(\mathtt{this})$. Show that if for all address a' such that $a' \neq a$, $a' \in dom(\chi)$, and $\phi(\mathtt{this}) \in dom(\chi)$, then $\chi'(a') = \chi(a')$.*

Proof. By structural induction on operational semantics,

- **Case** `methCall_OS`**.** The heap χ is unchanged in the next configuration σ'. It means $\chi'(a') = \chi(a')$. Hence, the desired result is correct.
- **Case** `varAssgn_OS`**.** It is similar to the case of methCall_OS.
- **Case** `return_OS`**:** It is also similar to the case of methCall_OS.
- **Case** `fieldAssgn_OS`**.** The heap χ only is updated if the field at the address $\phi(\mathtt{this})$ changed. However, we consider the address $a' \in dom(\chi)$ and $a' \neq \sigma(\mathtt{this})$. Therefore, there is no update in the heap here, and $\chi'(a') = \chi(a')$. So, the desired result is correct, as well.
- **Case** `objCreate_OS`**.** The heap χ is also changed in σ'. However, the heap is only changed at a new address α, where $\chi(\alpha) = (C, f_1 \mapsto \phi(x_1), \ldots f_n \mapsto \phi(x_n))$. Since α is new in the heap χ, we have $\alpha \notin dom(\chi)$. So, $a' \neq \alpha$. From that, we have $\chi'(a') = \chi(a')$. So, the desired result is also correct.

As a result, we have $\chi'(a') = \chi(a')$. □

Lemma 7. *Let w be an identifier such that $\lfloor w \rfloor_\sigma$ is defined. Show that $\lfloor w \rfloor_{\sigma \triangleleft \sigma'} = \lfloor w \rfloor_\sigma$.*

Proof. From Definition 8 of Adaptation on runtime configurations in [8], we have $\beta_2(\mathtt{zs}_1) = \beta_1(\mathtt{zs}_1)$, where β_1 and β_2 are variable maps from the runtime configuration σ_1 and σ_2 respectively, and $\mathtt{zs}_1$ is a set of identifiers defined in the current configuration σ.

Since w is defined in the current configuration σ, so we have $\lfloor w \rfloor_{\sigma \triangleleft \sigma'} = \lfloor w \rfloor_\sigma$. □

Lemma 8. *Let w be an identifier such that w is fresh in σ and σ'. Show that if $\exists v.\ \lfloor v \rfloor_{\sigma'}$ is defined, then $\lfloor w \rfloor_{\sigma \triangleleft \sigma'} = \lfloor v \rfloor_{\sigma'}$.*

Proof. From Definition 8 of Adaptation on runtime configurations in [8], we have $\beta_2(\mathtt{zs}') = \beta_2(\mathtt{zs}_2)$, where β_1 and β_2 are variable maps from the runtime configuration σ and σ' respectively, $\mathtt{zs}'$ is a set whose all identifiers are fresh in both variable maps β_1 and β_1, and $\mathtt{zs}_2$ is a set of identifiers defined in the current configuration σ'.

Since w is fresh in both the current configuration σ and the next configuration σ', so there exists an identifier v in $\texttt{zs}_2$ such that $\lfloor w \rfloor_{\sigma \triangleleft \sigma'} = \lfloor v \rfloor_{\sigma'}$. □

Lemma 9. *For any identifier $w \in dom(\beta'')$, where β'' is a variable map from $\sigma \triangleleft \sigma'$.*
Show that either
(1) If $w \in dom(\beta)$, then $\lfloor w \rfloor_{\sigma \triangleleft \sigma'} = \lfloor w \rfloor_{\sigma}$, or
(2) If $w \notin dom(\beta)$ and $w \notin dom(\beta')$, then $\exists v. v \in dom(\beta')$ such that $\lfloor w \rfloor_{\sigma \triangleleft \sigma'} = \lfloor v \rfloor_{\sigma'}$.

Proof. We obtain the proof with the use of Lemma 7 and Lemma 8. □

All technical lemmas mentioned at 5, 6, 7, 8, and 9 are shown in Isabelle/HOL in Section A.6 of the previous chapter.

Chapter 5

Conclusion and Future Work

5.1 Future Work

So far, we have a framework for holistic specifications formalized in Isabelle/HOL. The formalization can be a foundation to extend it or settle other parts built on top of it. In particular, we have some straightforward possible plans as follows.

- In this thesis, we have provided an Isabelle/HOL mechanization of the core of Chainmail. However, our formalization currently supports the only formal foundation of holistic specifications. Therefore, the immediate plan could provide a formal specification and verification in Isabelle/HOL some small case studies, i.e., `Wallet` example mentioned at the beginning of Chapter 4. Next, we want to provide holistic specifications and the reasoning of the sequence examples taken from the object-capability literature, such as the Bank/Account example [22] specified in Section 2.1.2 or attenuating the DOM (Domain Object Model). Then, we develop a full logic to prove their soundness in Isabelle/HOL.

- In a long-term plan, we plan to develop a technique of automated reasoning with that logic suggested above. Then, inspired by the initial Bank/Account work, we study the verification of capabilities in other programming languages, such as Javascript or Solidity.

5.2 Conclusion

In this thesis, we have presented the problem of holistic specifications, the background of Isabelle/HOL, and holistic specification, and a thorough survey of related work on the object-capability model and verification for object-capability programs in Chapter 2. The core of holistic specification language Chainmail is formalized in Isabelle/HOL mechanization written down in Chapter 3. We have constructed an Isabelle/HOL mechanization for the underlying language with proving the deterministic execution of this language, the syntax and semantics for Chainmail, and the proofs of properties of the underlying language, as well as Chainmail properties. We also have proposed lemmas and provided informal or formal proofs related to the properties of “only connectivity begets connectivity”, one of the two access principles enunciated in the object-capability literature in Chapter 4. It is considered preliminary results towards reasoning about the capability policies to be considered a fundamental means of verifying holistic specifications.

Appendix A

Auxiliary Functions, Lemmas in Isabelle/HOL, and Partial Proofs of Theorem 3

A.1 Auxiliary Functions supporting Operational semantics

```
theory Support
  imports Main

begin
lemma Greatest_eq [simp]:
  "(GREATEST x. x = (a::nat)) = a"
  by (simp add: Greatest_equality)

lemma sorted_head:
  "sorted xs ⟹ x ∈ set xs ⟹ hd xs ≤ x"
  by(induction xs arbitrary: x, auto)

lemma sorted_last:
  "sorted xs ⟹ x ∈ set xs ⟹ last xs ≥ x"
  by(induction xs arbitrary: x, auto)

lemma last_sorted_list_of_list_is_greatest:
  assumes fin: "finite A"
  assumes yin: "(y::nat) ∈ A"
  shows "y ≤ last (sorted_list_of_set A)"
  by (simp add: fin sorted_last yin)

lemma not_eq_a:
"finite A ⟹ Suc (last (sorted_list_of_set (insert a A))) ≠ a"
  by (metis Suc_n_not_le_n finite.insertI insertI1
      last_sorted_list_of_list_is_greatest)

lemma not_in_A:
"finite A ⟹ Suc (last (sorted_list_of_set (insert a A))) ∉ A"
  by (meson Suc_n_not_le_n finite.simps insertI2
      last_sorted_list_of_list_is_greatest)
end
```

A.2 Technical Lemmas supporting Deterministic

```
theory Deterministic_Aux
  imports Language
begin
lemma exec_det_aux:
  "M, σ →e σ' ⟹ M, σ →e σ'' ⟹ σ'' = σ'"
proof (induction arbitrary: σ'' rule: exec.induct)
  case (exec_method_call φ x y m params stmts a χ C paramValues M meth φ'' ψ)
  from ‹M, (φ # ψ, χ) →e σ''›
  show ?case
    apply(rule exec.cases)
    using exec_method_call
    by auto
next
  case (exec_var_assign φ x y stmts M ψ χ)
  from ‹M, (φ # ψ, χ) →e σ''›
  show ?case
    apply (rule exec.cases)
    by (auto simp: exec_var_assign)
next
  case (exec_field_assign φ y x stmts v χ χ' M ψ)
  from ‹M, (φ # ψ, χ) →e σ''›
  show ?case
    apply (rule exec.cases)
    using  exec_field_assign
    by auto
next
  case (exec_new φ x C params stmts paramValues M c obj' a χ χ' ψ)
  from ‹M, (φ # ψ, χ) →e σ''›
  show ?case
    apply (rule exec.cases)
    using exec_new
    by auto
next
  case (exec_return φ x stmts φ' x' stmts' M ψ χ)
  from ‹M, (φ # φ' # ψ, χ) →e σ''›
  show ?case
    apply (rule exec.cases)
    using exec_return
    by auto
qed

lemma internal_exec_rev_tr_nonempty':
  "internal_exec_rev M M' σ tr σ' ⟹ tr = [] ⟹ False"

  apply(induction rule: internal_exec_rev.induct, auto)
  done

lemma internal_exec_rev_tr_nonempty:
  "internal_exec_rev M M' σ [] σ' ⟹ False"
  using internal_exec_rev_tr_nonempty'
  by blast

lemma internal_exec_rev'_appI:
  "internal_exec_rev' M M' σ tr σ' ⟹
  internal_exec_rev' M M' σ' tr' σ'' ⟹
  internal_exec_rev' M M' σ (tr @ tr') σ''"

  apply(induct rule: internal_exec_rev'.induct)
  using internal_step
  by auto

lemma internal_exec_rev_appI:
  assumes "internal_exec_rev M M’ σ tr σ’’’"
  assumes "internal_exec_rev’ M M’ σ’’’ tr’ σ’’"
  shows "internal_exec_rev M M’ σ (tr @ tr’) σ’’"

  using assms
  apply(induction tr arbitrary: σ σ’’’ tr’ σ’’)
  using internal_exec_rev_tr_nonempty
   apply blast
  apply(erule internal_exec_rev.cases)
  by (simp add: internal_exec_rev’_appI internal_exec_rev_first_step)

lemma trace_rev1:
  "internal_exec M M’ σ tr σ’ ⟹ internal_exec_rev M M’ σ tr σ’"
proof (induction rule: internal_exec.induct)
  case (internal_exec_first_step σ σ’ c c’)
  thus ?case
    by (simp add: internal_exec_rev_first_step internal_refl)
next
   case (internal_exec_more_steps σ tr σ’ σ’’ c)
   note facts = internal_exec_more_steps
   thus ?case
   proof (cases rule: internal_exec.cases)
     case (internal_exec_first_step c c’)
     thus ?thesis
       using facts internal_exec_rev_first_step internal_refl internal_step
       by auto
   next
     case (internal_exec_more_steps tr’ σ’’’ c’)
     thus ?thesis
       using internal_exec_rev_appI facts internal_refl internal_step
       by blast
   qed
qed

lemma internal_exec_tr_nonempty:
  "internal_exec M M’ σ tr σ’ ⟹ tr = [] ⟹ False"

  by(induct rule: internal_exec.induct, auto)
```

```
lemma internal_exec_tr_nonempty' [simp]:
  "internal_exec M M' σ [] σ' = False"

  using internal_exec_tr_nonempty
  by blast

lemma internal_exec_appI:
  assumes "internal_exec_rev' M M' σ' tr σ''"
          "M;M', σ →ei⟨tr'⟩ σ'"
  shows "M;M',σ →ei⟨tr' @ tr⟩ σ''"
  using assms
proof (induction arbitrary:  tr' σ rule: internal_exec_rev'.induct )
  case (internal_refl σ)
  thus ?case by simp
next
  case (internal_step σ σ' c c' tr σ'')
  thus ?case
  proof -
    have "M;M',σ →ei⟨(tr' @ [σ']) @ tr⟩ σ''"
      by (meson internal_exec.simps internal_step internal_step)
    thus ?thesis
      by simp
  qed
qed

lemma internal_exec_appI1:
  assumes  "internal_exec M M' σ tr σ' "
           "internal_exec_rev' M M' σ' tr' σ'' "
  shows    "internal_exec_rev M M' σ (tr @ tr') σ''"

  using assms internal_exec_rev_appI trace_rev1
  by blast

lemma trace_rev2:
  "internal_exec_rev M M' σ tr σ' ⟹ internal_exec M M' σ tr σ' "
proof  (induction rule: internal_exec_rev.induct)
  case (internal_exec_rev_first_step σ σ' c c' tr σ'')
  note facts = internal_exec_rev_first_step
  from facts(7)
  show ?case
  proof(cases rule: internal_exec_rev'.cases)
    case internal_refl
    thus ?thesis
      using internal_exec_first_step facts
      by blast
  next
    case (internal_step σ''' c c' tr')
    thus ?thesis
      using internal_exec_appI internal_exec_first_step
            internal_exec_rev_first_step
      by (metis (full_types) append_Cons  self_append_conv2)
    qed
qed

lemma trace_rev:
  "internal_exec_rev M M' σ tr σ' = internal_exec M M' σ tr σ' "
  using trace_rev1 trace_rev2
  by blast

lemma internal_exec_rev'_det_prefix:
  "internal_exec_rev' M M' σ tr σ'   ⟹
   internal_exec_rev' M M' σ tr' σ'' ⟹
  (∃tr''. (tr = tr' @ tr'') ∨ (tr' = tr @ tr''))"
proof(induction arbitrary: tr' σ'' rule: internal_exec_rev'.induct)
  case (internal_refl σ)
  thus ?case by blast
next
  case (internal_step σ σ' c c' tr σ'')
  note facts = internal_step
  from internal_step(8)
  show ?case
  proof(cases rule: internal_exec_rev'.cases)
    case internal_refl
    thus ?thesis
      by blast
  next
    case (internal_step σ c c' tr)
    thus ?thesis
```

```
      using exec_det_aux facts
      by (metis append_Cons)
  qed
qed

lemma internal_exec_rev_det_prefix':
  "M;M', σ →eir⟨tr⟩ σ' ⟹
  M;M', σ →eir⟨tr'⟩ v ⟹
  (∃ tr''.(tr = (tr' @ tr'')) ∨ (tr' = tr @ tr''))"
proof(induction arbitrary: tr' v rule: internal_exec_rev.induct)
  case (internal_exec_rev_first_step  σ σ' c c' tr σ'')
  note facts = internal_exec_rev_first_step
  from ‹M;M',σ →eir⟨tr'⟩ v›
  show ?case
  proof (cases rule: internal_exec_rev.cases)
    case (internal_exec_rev_first_step σ''' c'' c''' tr'')
    have "σ''' = σ''"
      using facts(2) internal_exec_rev_first_step(3) exec_det_aux
      by blast
    with facts internal_exec_rev_first_step
    show ?thesis
      using internal_exec_rev'_det_prefix
      by auto
  qed
qed

lemma internal_exec_det_prefix':
  "M;M', σ →ei⟨tr⟩ σ' ⟹
  M;M', σ →ei⟨tr'⟩ v ⟹
  (∃ tr''.(tr = tr' @ tr'') ∨ (tr' = tr @ tr''))"
  using internal_exec_rev_det_prefix' trace_rev
  by blast

inductive_cases internal_exec_elim [elim!]: "internal_exec M M' σ tr σ''"
inductive_cases visible_exec_elim [elim!]: "visible_exec M M' σ σ''"

lemma internal_exec_det_aux:
 "M;M', σ →ei⟨tr⟩ σ' ⟹ M;M', σ →ei⟨tr⟩ v ⟹  v = σ'"
  by (metis internal_exec_elim last_ConsL last_snoc)
```

```
lemma internal_exec_is_internal:
  "internal_exec M M' σ tr σ' ⟹
   ∀σᵢ ∈ set tr.(∃ c.(this_class_lookup σᵢ = Some c ∧ c ∈ dom M))"
  by (induction rule: internal_exec.induct) simp+

lemma internal_exec_appD:
 assumes "M;M',σ →ₑi⟨tr'⟩ σ'"
 shows  "∀tr'' σᵢ. (M;M',σ →ₑi⟨tr' @ tr''⟩ σᵢ) ⟶
        (internal_exec_rev' M M' σ' tr'' σᵢ)"
  using assms
proof (induction tr' arbitrary: σ σ' rule: rev_induct)
  case Nil
  thus ?case by simp
next
  case (snoc x xs)
  note facts = internal_exec_more_steps
  fix tr'' σᵢ
  have x_eq [simp]: "x = σ'"
    using snoc.prems
    by auto
  hence "M;M',σ →ₑi⟨xs @ [σ']⟩ σ'"
    using snoc.prems
    by blast
  from ‹ M;M',σ →ₑi⟨xs @ [x]⟩ σ'›
  show ?case
  proof (cases rule: internal_exec.cases)
    case (internal_exec_first_step c c')
    hence xs_Nil [simp]: "xs = []"
      by simp
    show ?thesis
    proof(intro allI impI)
      fix tr'' σᵢ
      assume "M;M',σ →ₑi⟨(xs @ [x]) @ tr''⟩ σᵢ"
      hence "M;M',σ →ₑi⟨x # tr''⟩ σᵢ"
        by simp
      hence "internal_exec_rev M M' σ (x#tr'') σᵢ"
        using trace_rev
        by simp
      thus "M;M',σ' →eir1⟨tr''⟩ σi"
        apply (rule internal_exec_rev.cases)
        by simp
    qed
  next
    case (internal_exec_more_steps tr σ'' c)
    note facts = internal_exec_more_steps
    hence xs_tr [simp]: "xs = tr"
      by blast
    have internal_xs: "M;M',σ →ei⟨tr @ [σ']⟩ σ'"
      using snoc.prems
      by auto
    from this
    obtain σtr
      where "(M;M',σ →ei⟨tr⟩ σtr) ∧
             (internal_exec_rev' M M' σtr [σ'] σ')"
      using snoc(1) xs_tr internal_exec_more_steps
      by metis
    with snoc(1) have
      IH: "∀tr'' σi. (M;M',σ →ei⟨xs @ tr''⟩ σi)
           ⟶ (M;M',σtr →eir1⟨tr''⟩ σi)"
      by (metis xs_tr)
    show ?thesis
    proof(intro allI impI)
      fix tr'' σi
      assume "M;M',σ →ei⟨(xs @ [x]) @ tr''⟩ σi"
      hence "M;M',σ →ei⟨xs @ (x # tr'')⟩ σi"
        by simp
      with IH have "M;M',σtr →eir1⟨x # tr''⟩ σi"
        by metis
      thus " M;M',σ' →eir1⟨tr''⟩ σi"
        apply (cases rule: internal_exec_rev'.cases)
        using x_eq
        by blast
    qed
  qed
qed

lemma internal_exec_appD1:
  assumes "M;M',σ →ei⟨tr'⟩ σ'"
          "M;M',σ →ei⟨tr' @ tr''⟩ σi"
  shows   "internal_exec_rev' M M' σ' tr'' σi"

  using assms internal_exec_appD
  by metis

lemma visible_exec_det_aux:
  "M;M', σ →e σ' ⟹ link_wf M M' ⟹
   M;M', σ →e σ'' ⟹ σ'' = σ'"
proof(induction arbitrary: σ'' rule: visible_exec.induct)
  case (visible_exec_intro M M' σ tr σi σ' c)
  note facts = visible_exec_intro
  from ‹M;M',σ →e σ''›
  show ?case
  proof(cases rule: visible_exec.cases)
    case (visible_exec_intro tr' σi' c)
    note facts1 = visible_exec_intro
    from ‹M;M',σ →ei⟨tr⟩ σi› have
      "∀σi ∈ set tr.
       (∃ c. this_class_lookup σi = Some c ∧ c ∈ dom M)"
      using internal_exec_is_internal
      by auto
    from ‹M;M',σ →ei⟨tr'⟩ σi'› have
      "∀σi ∈ set tr'.
       (∃ c. this_class_lookup σi = Some c ∧ c ∈ dom M)"
      using internal_exec_is_internal
      by auto
    obtain tr'' where tr: "tr = tr' @ tr'' ∨ tr' = tr @ tr''"
      using facts(1) visible_exec_intro(1) internal_exec_det_prefix'
      by metis
    hence "tr'' = []"
    proof (rule disjE)
      show " tr = tr' @ tr'' ⟹ tr'' = []"
      proof (rule ccontr)
        assume tr_def: "tr = tr' @ tr''" "¬(tr'' = [])"
        from ‹M;M',σ →ei⟨tr⟩ σi›
        have "M;M',σ →ei⟨tr' @ tr''⟩ σi"
          using tr_def
          by simp
      from this and ‹M;M',σ →ei⟨tr'⟩ σi'›
      have inter_ex:  "internal_exec_rev' M M' σi' tr'' σi"
        using internal_exec_appD1
        by auto
      hence  "tr'' = []"
        proof (cases rule: internal_exec_rev'.cases)
          case internal_refl
          thus ?thesis
            by simp
        next
          case (internal_step σ' c' c'' tr)
          from facts1(2) internal_step(2)
          have a1: "σ'' = σ'"
            using exec_det_aux
            by auto
        from facts1 internal_step
        have a2: "this_class_lookup σ'' = Some c ∧ c ∈ dom M'" and
             a3: "this_class_lookup σ' = Some c'' ∧ c'' ∈ dom M"
           apply blast
          using local.internal_step(5) local.internal_step(6)
          by auto
        have a4: "c ∈ dom M' ∧ c'' ∈ dom M"
          by (simp add: local.internal_step(6)
              local.visible_exec_intro(4))
        from a1 a2 a3
        have a5: " c =  c''"
            by simp
          from a4 a5 show ?thesis
            using link_wf_def visible_exec_intro.prems(1)
            by auto
        qed
        thus False
          using ‹tr'' ≠ []›
          by auto
      qed
      show "tr' = tr @ tr'' ⟹ tr'' = []"
      proof (rule ccontr)
        assume tr_def1: "tr' = tr @ tr''" "¬(tr'' = [])"
```

```
        from ‹M;M',σ →ei⟨tr'⟩ σi'›
        have "M;M',σ →ei⟨tr @ tr''⟩ σi'"
          using tr_def1 by simp
        from this and ‹M;M',σ →ei⟨tr⟩ σi›
        have inter_ex1: "internal_exec_rev' M M' σi tr'' σi'"
          using internal_exec_appD1
          by auto
        hence  "tr'' = []"
        proof (cases rule: internal_exec_rev'.cases)
          case internal_refl
          thus ?thesis
            by simp
        next
          case (internal_step σ' c' c'' tr)
          thus ?thesis using facts exec_det_aux link_wf_def
            by (metis IntI empty_iff option.inject)
        qed
        thus False
          using ‹tr'' ≠ []› by auto
      qed
    qed
    thus ?thesis
      using facts1 tr exec_det_aux append.right_neutral
        internal_exec_det_aux visible_exec_intro.hyps(1)
        visible_exec_intro.hyps(2)
      by metis
  qed
qed
end
```

A.3 Technical Lemmas supporting Linking module preserving execution

```
theory LinkingPreserve_Aux
  imports Deterministic
begin

lemma link_exec_aux:
  "⟦ M, σ →e σ'; link_wf M M' ⟧ ⟹ (M ∘l M'), σ →e σ'"
  unfolding link_wf_def moduleLinking_def
            moduleAux_def dom_def build_call_frame_def
proof (induction  rule: exec.induct)
  case (exec_method_call φ x y m params stmts α χ C paramValues M meth φ'' ψ)
  thus ?case
    by (simp add: exec.exec_method_call ident_lookup.induct
        ℳ_def split: option.splits)
next
  case (exec_var_assign φ x y stmts M ψ χ)
  thus ?case
    apply (simp add: this_field_lookup.induct)
    using exec.exec_var_assign by auto
next
  case (exec_field_assign φ y x stmts v χ χ' M ψ)
  thus ?case
  apply (simp add: this_field_lookup.induct ident_lookup.induct)
    by (simp add: exec.exec_field_assign)
next
  case (exec_new φ x C params stmts paramValues M c obj' α χ χ' ψ)
  thus ?case
    by (simp add: ident_lookup.induct exec.exec_new)
next
  case (exec_return φ x stmts φ' x' stmts' M ψ χ)
  thus ?case
    apply (simp add: ident_lookup.induct)
    using exec.exec_return by auto
qed

lemma internal_linking_1_aux:
  "⟦M;M',σ →ei⟨tr⟩ σ'; link_wf_3M M M' M''⟧ ⟹
        M; (M' ∘l M''), σ →ei⟨tr⟩ σ'"
proof (induction rule: internal_exec.induct)
  case (internal_exec_first_step  σ σ' c c')
  have a:  "link_wf M (M' ∘l M'')"
    using ‹link_wf_3M M M' M''› link_wf_def link_wf_3M_def
    by fastforce
  have wf: "link_wf M M'"
    using ‹link_wf_3M M M' M''›
    by simp
  have  "((M ∘l M') ∘l M''), σ →e σ'"
    apply(rule link_exec_aux)
     apply(rule ‹(M ∘l M'), σ →e σ'›)
    using ‹(link_wf_3M M M' M'')› link_wf_3M_dest
    by blast
  from this wf link_assoc
  have b: "(M ∘l (M' ∘l M'')), σ →e σ'"
    by metis
  have "dom (M' ∘l M'') = dom M' ∪ dom M''"
    by (simp add: ‹link_wf_3M M M' M''›)
  hence c: "this_class_lookup σ = Some c ∧ c ∈ dom (M' ∘l M'')"
    using internal_exec_first_step
    by blast
  from a b c internal_exec_first_step
  show ?case
    using internal_exec.internal_exec_first_step
    by blast
next
  case (internal_exec_more_steps  σ tr σ' σ'' c)
  have a: "M;M' ∘l M'',σ →ei⟨tr⟩ σ' "
    using ‹link_wf_3M M M' M'' ⟹ M;M' ∘l M'',σ →ei⟨tr⟩ σ'›
          and ‹link_wf_3M M M' M''›
    by simp
  have asm: "M ∘l (M' ∘l M'') = (M ∘l M') ∘l M''"
    by (metis internal_exec_more_steps.prems
        link_assoc link_wf_3M_dest(1))
  have "((M ∘l M') ∘l M''), σ' →e σ'' "
    apply (rule link_exec_aux)
     apply (rule ‹(M ∘l M'), σ' →e σ''›)
    using ‹link_wf_3M M M' M''› link_wf_3M_dest
    by blast
  from this and asm
  have b: "(M ∘l (M' ∘l M'')), σ' →e σ''"
    by simp
  have c: "this_class_lookup σ'' = Some c ∧ (c ∈ dom M)"
    using internal_exec_more_steps
    by simp
  from a b c internal_exec.internal_exec_more_steps
  show ?case
    by simp
qed

lemma internal_linking_2_aux:
  "M;M',σ →ei⟨tr⟩ σ' ⟹ link_wf_3M M M' M''⟹
  (M ∘l M''); M', σ →ei⟨tr⟩ σ'"
proof (induction rule: internal_exec.induct)
  case (internal_exec_first_step σ σ' c c')
  have a: "link_wf (M ∘l M'') M'"
    using ‹link_wf_3M M M' M''›
    by (fastforce simp add: link_wf_def link_wf_3M_def)
  have " M∘l M'' = M''∘l M "
    using ‹link_wf_3M M M' M''› link_commute
    by blast
  hence "(M∘l M'') ∘l M' =  M''∘l (M ∘l M')"
    using ‹link_wf_3M M M' M''› link_wf_3M_def link_wf_def
        internal_exec_first_step.hyps(1)
        internal_exec_first_step.prems
        link_assoc link_commute link_wf_3M_dest(2)
        link_wf_3M_dest(3) link_wf_3M_dest(4)
    by metis
  hence "(M∘l M'') ∘l M' = (M ∘l M') ∘lM''"
    by (simp add: ‹link_wf_3M M M' M''›)
  hence asm: "(M ∘l M') ∘l M'' = (M∘l M'') ∘l M'"
    by simp
  have "((M ∘l M') ∘l M''), σ →e σ'"
    apply (rule link_exec_aux)
     apply (rule ‹(M ∘l M'), σ →e σ'›)
    using internal_exec_first_step
    by blast
  from this and asm
  have b: "((M ∘l M'') ∘l M'), σ →e σ'"
    by simp
  have c: "this_class_lookup σ = Some c ∧ (c ∈ dom M') "
    using internal_exec_first_step
    by simp
  have d: "this_class_lookup σ' = Some c' ∧
           (c' ∈ dom (M ∘l M''))"
    using internal_exec_first_step
    by simp
  from a b c d  internal_exec_first_step
  show ?case
    using internal_exec.internal_exec_first_step
    by simp
next
  case (internal_exec_more_steps  σ tr σ' σ'' c)
  from ‹(link_wf_3M M M' M'') ⟹
       (M ∘l M'');M',σ →ei⟨tr⟩ σ'›
       and ‹(link_wf_3M M M' M'')›
  have a: "(M ∘l M'');M',σ →ei⟨tr⟩ σ'"
    by simp
  have "M ∘l M'' = M''∘l M "
    using ‹link_wf_3M M M' M''› link_commute
    by blast
  hence "(M∘l M'') ∘l M' =  M''∘l (M ∘l M')"
    using ‹link_wf_3M M M' M''› link_wf_3M_def link_wf_def
    by (metis Int_Un_distrib Un_Int_crazy Un_Int_distrib
        Un_commute distrib_imp2 inf.right_idem inf_commute
        inf_sup_absorb internal_exec_more_steps.prems link_assoc
        link_commute link_dom link_wf_3M_dest(3) link_wf_3M_dest(4)
        sup.left_commute sup_assoc sup_bot.left_neutral
        sup_bot.right_neutral sup_idem sup_inf_distrib1)
  hence "(M∘l M'') ∘l M' = (M ∘l M') ∘lM''"
    by (simp add: ‹link_wf_3M M M' M''›)
  hence asm: "(M ∘l M') ∘l M'' = (M∘l M'') ∘l M'"
    by simp
  have "((M ∘l M') ∘l M''), σ' →e σ''"
    apply (rule link_exec_aux)
     apply (rule ‹(M ∘l M'), σ' →e σ''›)
    using ‹link_wf_3M M M' M''›
    by blast
  from this and asm
  have b: "((M ∘l M'') ∘l M'), σ' →e σ''"
    by simp
  have c: "this_class_lookup σ'' = Some c ∧
          (c ∈ dom (M ∘l M''))"
    using internal_exec_more_steps
    by simp
  from a b c internal_exec.internal_exec_more_steps
  show ?case
    by simp
qed


lemma visible_exec_linking_1_aux:
  "⟦(M;M',σ →e σ'); (link_wf_3M M M' M'')⟧ ⟹
    M; (M' ∘l M''), σ →e σ'"
proof (induction rule: visible_exec.induct)
  case (visible_exec_intro M M' σ tr σ' σ'' c)
  have a: "M; (M' ∘l M''), σ →ei⟨tr⟩ σ'"
    using ‹ M; M',σ →ei⟨tr⟩ σ'›
    by (simp add: visible_exec_intro internal_linking_1_aux)
  have asm: "M ∘l (M' ∘l M'') = (M ∘l M') ∘l M''"
    by (metis link_assoc link_wf_3M_dest(1)
          visible_exec_intro.prems)
  have "((M ∘l M') ∘l M''), σ' →e σ'' "
    apply (rule link_exec_aux)
     apply (rule ‹(M ∘l M'), σ' →e σ''›)
    using ‹link_wf_3M M M' M''› link_wf_3M_dest
    by blast
  from this and asm
  have b: "(M ∘l (M' ∘l M'')), σ' →e σ''" by simp
  have c: "this_class_lookup σ'' = Some c ∧
          ( c ∈ dom (M' ∘l M''))"
    using ‹this_class_lookup σ'' = Some c› and ‹c ∈ dom M'›
    by simp
  from a b c show ?case
    using visible_exec.visible_exec_intro
    by simp
qed


lemma visible_exec_linking_2_aux:
  "⟦(M;M',σ →e σ'); (link_wf_3M M M' M'')⟧ ⟹
    (M ∘l M''); M', σ →e σ'"
proof (induction rule: visible_exec.induct)
```

```
  case (visible_exec_intro M M' σ tr σ' σ'' c)
  have a: "(M ∘l M'');M',σ →ei⟨tr⟩ σ'"
    using ‹M;M',σ →ei⟨tr⟩ σ'›
    by (simp add: ‹link_wf_3M M M' M''› internal_linking_2_aux)
  have "M ∘l M'' = M'' ∘l M"
    using link_commute visible_exec_intro
    by auto
  hence "(M∘l M'') ∘l M' =  M''∘l (M ∘l M')"
    using ‹link_wf_3M M M' M''› link_wf_3M_def link_wf_def
    by auto
  hence "(M∘l M'') ∘l M' = (M ∘l M') ∘lM''"
    by (simp add: ‹link_wf_3M M M' M''›)
  hence asm:  "(M ∘l M') ∘l M'' = (M∘l M'') ∘l M'"
    by simp
  have "((M ∘l M') ∘l M''), σ' →e σ''"
    apply (rule link_exec_aux)
     apply (rule ‹(M ∘l M'), σ' →e σ''›)
    using ‹link_wf_3M M M' M''›
    by blast
  from this and asm
  have b: "((M ∘l M'') ∘l M'), σ' →e σ''"
    by simp
  have c: "this_class_lookup σ'' = Some c ∧ c ∈ dom M'"
    using ‹this_class_lookup σ'' = Some c› and ‹c ∈ dom M'›
    by simp
  from a b c visible_exec.visible_exec_intro
  show ?case
    by simp
qed
```

A.4 Technical Lemmas supporting Adaptation

```
definition
ident_subst :: "Identifier ⇒ Identifier  ⇒ Identifier ⇒ Identifier"
  where
"ident_subst x y v = (if v = x then y else v)"

fun
```

```
stmt_subst :: "Stmt ⇒ Identifier ⇒ Identifier ⇒ Stmt"
  where
"stmt_subst (AssignToField f v) x y =
                (AssignToField f (ident_subst x y v))" |
"stmt_subst (ReadFromField v f) x y =
                (ReadFromField (ident_subst x y v) f)" |
"stmt_subst (MethodCall v v' m vs) x y =
    (MethodCall (ident_subst x y v) (ident_subst x y v') m (map (ident_subst x
y) vs))" |
"stmt_subst (NewObject v c vs) x y =
    (NewObject (ident_subst x y v) c (map (ident_subst x y) vs))" |
"stmt_subst (Return v) x y =
    (Return (ident_subst x y v))"
```

fun

```
stmt_idents :: "Stmt ⇒ Identifier set"
  where
"stmt_idents (AssignToField f v) = {v}" |
"stmt_idents (ReadFromField v f) = {v}" |
"stmt_idents (MethodCall v v' m vs) = {v,v'} ∪ (set vs)" |
"stmt_idents (NewObject v c vs) = {v} ∪ (set vs)" |
"stmt_idents (Return v) = {v}"
```

lemma *stmt_subst_idents:*

```
  "stmt_idents (stmt_subst s x y) =
   ((stmt_idents s - {x}) ∪ (if x ∈ stmt_idents s then {y} else {}))"
  by (induction rule: stmt_idents.induct,
      auto simp: ident_subst_def split: if_splits)
```

fun

```
stmts_subst :: "Stmts ⇒ Identifier ⇒ Identifier ⇒ Stmts"
  where
"stmts_subst (SingleStmt s) x y =
             (SingleStmt (stmt_subst s x y))" |
"stmts_subst (Seq s1 s2) x y =
             (Seq (stmt_subst s1 x y) (stmts_subst s2 x y))"
```

fun

```
stmts_idents :: "Stmts ⇒ Identifier set"
```

```
  where
"stmts_idents (SingleStmt s) = (stmt_idents s)" |
"stmts_idents (Seq s1 s2) = (stmt_idents s1 ∪ (stmts_idents s2))"

lemma stmts_subst_idents:
  "stmts_idents (stmts_subst s x y) =
   ((stmts_idents s - {x}) ∪ (if x ∈ stmts_idents s then {y} else {}))"
  by (induction rule: stmts_subst.induct,
      auto simp: stmt_subst_idents)

fun
stmts_subst_list :: "Stmts ⇒ Identifier list ⇒ Identifier list ⇒ Stmts"
  where
"stmts_subst_list s (v#vs) (v'#vs') =
              (stmts_subst_list (stmts_subst s v v') vs vs')" |
"stmts_subst_list s (v#vs) [] = undefined" |
"stmts_subst_list s [] (v'#vs') = undefined" |
"stmts_subst_list s [] [] = s"


definition
stmts_subst_list_wf :: "Stmts ⇒ Identifier list ⇒ Identifier list ⇒ bool"
  where
"stmts_subst_list_wf s vs vs' ≡  (length vs = length vs') "

lemma stmts_list_idents:
 "stmts_subst_list_wf s vs vs'  ⟹
  (stmts_idents (stmts_subst_list s vs vs') ⊆
  ((stmts_idents s - (set vs)) ∪ (set vs')))"
proof (induction rule: stmts_subst_list.induct)
  case (1 s v vs v' vs')
  then
  have assm1: "stmts_subst_list_wf (stmts_subst s v v') vs vs'"
    and assm2: "stmts_subst_list_wf (stmts_subst s v v') vs vs' ⟹
                stmts_idents (stmts_subst_list (stmts_subst s v v') vs vs')
                ⊆ stmts_idents (stmts_subst s v v') - set vs ∪ set vs'"
     apply (simp add: stmts_subst_list_wf_def)
    by (simp add: "1.IH")
  from assm1 and assm2
  have assm3: "stmts_idents (stmts_subst_list (stmts_subst s v v') vs vs')
              ⊆ stmts_idents (stmts_subst s v v') - set vs ∪ set vs'"
    by blast
  hence assm4: "stmts_idents (stmts_subst_list s (v # vs) (v' # vs')) =
       stmts_idents (stmts_subst_list (stmts_subst s v v') vs vs')"
    by auto
  from assm1 assm2 assm3 assm4
  have assm5: "stmts_idents (stmts_subst s v v') - set vs ∪ set vs'
              ⊆ (stmts_idents s - set (v # vs) ∪ set (v' # vs'))"
    using stmts_subst_idents
    by auto
  from  assm1 assm2 assm3 assm4 assm5
  have assm6: "stmts_idents (stmts_subst_list s (v # vs) (v' # vs'))
              ⊆ (stmts_idents s - set (v # vs) ∪ set (v' # vs'))"
    by blast
  thus ?case
    by auto
next
  case (2 s v vs)
  thus ?case
    by ( auto simp add: stmts_subst_list_wf_def)
next
  case (3 s v' vs')
  thus ?case
    by ( auto simp add: stmts_subst_list_wf_def)
next
  case (4 s)
  thus ?case
    by simp
qed
```

The `fresh_idents X xs` is used to generate a list of fresh identifiers where none of the new identifiers appear in `X` or `xs`.

```
primrec
fresh_idents :: "Identifier set ⇒ Identifier list ⇒ Identifier list"
  where
"fresh_idents X [] = []" |
"fresh_idents X (x#xs)  = (let v = fresh_nat (X ∪ (set (x#xs))) in
                                    (v # (fresh_idents (X ∪ {v}) xs)))"

lemma fresh_idents_length [simp]:
"length (fresh_idents X xs) = length xs"
  apply(induction xs arbitrary: X)
   apply clarsimp+
  by(metis length_Cons)


lemma fresh_ident_greater:
  "finite X ⟹ X ≠ {} ⟹ fresh_nat X > Max X"
  unfolding Max_def If_def
  by (metis Max_def Max_less_iff fresh_nat_def
      last_sorted_list_of_list_is_greatest le_imp_less_Suc)


lemma fresh_idents_greater:
  "finite X  ⟹  xs ≠ [] ⟹ (∀x ∈ set (fresh_idents X xs). x > Max (X ∪ set
xs))"
  apply(induction xs arbitrary: X)
   apply simp
  apply (clarsimp simp: Let_def)
  apply (rule conjI)
  using fresh_ident_greater apply simp
  apply(rule conjI)
  using fresh_ident_greater apply auto[1]
  apply clarsimp
  apply(subgoal_tac "(insert a (X ∪ set xs)) ≠ {} ∧ finite (insert a (X ∪ set
xs))")
  using fresh_ident_greater
proof -
  fix a :: nat and xsa :: "nat list"
      and Xa :: "nat set" and x :: nat
  assume a1:
    "x ∈ set (fresh_idents (insert (fresh_nat (insert a (Xa ∪ set xsa))) Xa) xsa)"
  assume a2:
    "⋀X. ⟦finite X; xsa ≠ []⟧ ⟹ ∀x∈set (fresh_idents X xsa). ∀a∈X ∪ set xsa.
a < x"
  assume a3: "finite Xa"
  assume a4:
    "insert a (Xa ∪ set xsa) ≠ {} ∧ finite (insert a (Xa ∪ set xsa))"
  have f5:
    "∀n. n ∉ set (fresh_idents (insert (fresh_nat (insert a (Xa ∪ set xsa))) Xa)
xsa) ∨
        (∀na. na ∉ insert (fresh_nat (insert a (Xa ∪ set xsa))) Xa ∪ set xsa
∨ na < n)"
    using a3 a2
    by (metis finite.insertI fresh_idents.simps(1) insert_not_empty
        mk_disjoint_insert set_empty2)
  have f6: "∀n N na. (n::nat) ∉ N ∧ n ≠ na ∨ n ∈ insert na N"
by force
  hence "a < fresh_nat (insert a (Xa ∪ set xsa))"
    using a4 by (metis (no_types) Max_less_iff fresh_ident_greater)
  hence "a < x"
    using f5 a1 by fastforce
  thus "a < x ∧ (∀n∈Xa ∪ set xsa. n < x)"
    using f6 f5 a1 by (metis (no_types) Un_insert_left)
next
  fix a xs X x
  show
  "⟦⋀X. ⟦finite X; xs ≠ []⟧ ⟹ ∀x∈set (fresh_idents X xs).
                          ∀a∈X ∪ set xs. a < x; finite X;
     x ∈ set (fresh_idents (insert (fresh_nat (insert a (X ∪ set xs))) X) xs);
        ⋀X. ⟦finite X; X ≠ {}⟧ ⟹ Max X < fresh_nat X⟧
      ⟹ insert a (X ∪ set xs) ≠ {} ∧ finite (insert a (X ∪ set xs))"
    by blast
qed

lemma fresh_idents_empty:
  "finite X ⟹ (set (fresh_idents X xs) ∩ (X ∪ set xs)) = {}"
proof-
  assume "finite X "
  hence "(∀x∈set (fresh_idents X xs). x > Max (X ∪ set xs))"
    by (metis empty_iff empty_set fresh_idents.simps(1) fresh_idents_greater)
  then
  show "(set (fresh_idents X xs) ∩ (X ∪ set xs)) = {}"
    by (meson Int_emptyI List.finite_set Max_ge ‹finite X› finite_UnI not_le)
qed
```

```
lemma fresh_idents_distinct [simp]:
  "finite X ⟹ distinct (fresh_idents X xs)"
proof(induction xs arbitrary: X)
  case Nil
  thus ?case
    by clarsimp
next
  case (Cons a xs)
  show ?case
  proof (clarsimp simp: Let_def, rule conjI)
    show "fresh_nat (insert a (X ∪ set xs))
    ∉ set (fresh_idents (insert (fresh_nat (insert a (X ∪ set xs))) X) xs)"
      using fresh_idents_empty Cons.prems contra_subsetD
      by blast
  next
    show
      "distinct (fresh_idents (insert (fresh_nat (insert a (X ∪ set xs))) X) xs)"
      using Cons fresh_idents_empty
      by simp
  qed
qed

fun
ident_subst_list ::
  "Identifier ⇒ Identifier list ⇒ Identifier list ⇒ Identifier"
  where
"ident_subst_list x (v#vs) (v'#vs') = (ident_subst v v' x)" |
"ident_subst_list x (v#vs) [] = undefined" |
"ident_subst_list x [] (v'#vs') = undefined" |
"ident_subst_list x [] [] = x"
end
```

A.5 Technical Lemmas supporting Lemmas

```
theory Lemmas_Aux
  imports Adaptation
begin
```

Properties of classical logic

```
lemma aComplement_1:
  "Assert_wf A M M' σ ⟹
    (aAnd A  (aNot A) M M' σ) = afalse M M' σ"
  unfolding Assert_wf_def aAnd_def aNot_def afalse_def bopt_def
  by auto

lemma aComplement_2:
  "Assert_wf A M M' σ ⟹
    (aOr A  (aNot A) M M' σ) = atrue M M' σ"
  unfolding Assert_wf_def aNot_def aOr_def atrue_def bopt_def
  by auto

lemma aCommutative_1:
  "⟦Assert_wf A M M' σ;  Assert_wf A' M M' σ ⟧ ⟹
    (aOr A  A' M M' σ) = (aOr A' A M M' σ)"
  unfolding Assert_wf_def aOr_def bopt_def
  by auto

lemma aCommutative_2:
  "⟦Assert_wf A M M' σ;  Assert_wf A' M M' σ ⟧ ⟹
    (aAnd A A' M M' σ) = (aAnd A' A M M' σ)"
  unfolding Assert_wf_def aAnd_def bopt_def
  by auto

lemma aAssociative_1:
  "⟦Assert_wf A M M' σ; Assert_wf A' M M' σ ⟧ ⟹
   (aOr (aOr A  A') A'' M M' σ) = (aOr A (aOr A' A'') M M' σ)"
  unfolding aOr_def bopt_def option.case_eq_if
  by simp

lemma aAssociative_2:
  "⟦Assert_wf A M M' σ; Assert_wf A' M M' σ ⟧ ⟹
   (aAnd (aAnd A  A') A'' M M' σ) = (aAnd A (aAnd A' A'') M M' σ)"
  unfolding aAnd_def bopt_def option.case_eq_if
  by simp

lemma aDistributive_1:
  "⟦Assert_wf A M M' σ; Assert_wf A' M M' σ; Assert_wf A'' M M' σ ⟧ ⟹
   (aAnd (aOr A  A') A'' M M' σ) = (aOr (aAnd A A'') (aAnd A' A'') M M' σ)"
  unfolding aAnd_def aOr_def bopt_def option.case_eq_if
  by auto

lemma aDistributive_2:
  "⟦Assert_wf A M M' σ;  Assert_wf A' M M' σ; Assert_wf A'' M M' σ ⟧ ⟹
   (aOr (aAnd A  A') A'' M M' σ) = (aAnd (aOr A A'') (aOr A' A'') M M' σ)"
  unfolding aAnd_def aOr_def bopt_def option.case_eq_if
  by auto

lemma aDeMorgan_1:
  "⟦Assert_wf A M M' σ;  Assert_wf A' M M' σ ⟧ ⟹
    (aNot (aAnd A  A')  M M' σ) = (aOr (aNot A) (aNot A') M M' σ)"
  unfolding aAnd_def aOr_def aNot_def bopt_def option.case_eq_if
  by auto

lemma aDeMorgan_2:
  "⟦Assert_wf A M M' σ; Assert_wf A' M M' σ ⟧ ⟹
    (aNot (aOr A  A') M M' σ) = (aAnd (aNot A) (aNot A') M M' σ)"
  unfolding aAnd_def aOr_def aNot_def bopt_def option.case_eq_if
  by auto

lemma aUniversal_existential_1:
  "Assert_wf A M M' σ ⟹
          aNot (aEx (λx. A  )) M M' σ = aAll (λx. aNot A) M M' σ"
  unfolding Assert_wf_def aNot_def aEx_def aAll_def
  by(auto split: option.splits)

lemma aUniversal_existential_2:
  "Assert_wf A M M' σ ⟹
          aNot (aAll (λx. A)) M M' σ = aEx (λx. aNot A) M M' σ"
  unfolding Assert_wf_def aNot_def aEx_def aAll_def
  using option.case_eq_if option.simps
  by auto

lemma aImplication:
  "⟦Assert_wf A M M' σ; Assert_wf A' M M' σ⟧  ⟹
        (aAnd A (aImp A A'))  M M' σ = Some True ⟹
```

```
      A' M M' σ = Some True"
  unfolding Assert_wf_def bopt_def aImp_def aAnd_def aImp_def
  by auto

lemma aNeverHold:
  "Assert_wf A M M' σ ⟹
      (aAnd A (aNot A) M M' σ) = Some False"
  unfolding Assert_wf_def aAnd_def aNot_def bopt_def
  by auto
```

Some lemmas, which could be useful for reasoning about holistic specifications, are formed and proved in this section. We make the proof details of these lemmas in Appendix A.5.

```
lemma object_Unchange_Aux:
  "M, σ →e σ' ⟹ σ = (φ#ψ,χ) ⟹
  σ' = (φ'#ψ',χ') ⟹ finite (dom χ) ⟹
  ∀a1.((a1 ≠ (this φ) ∧ a1 ∈ dom χ ∧ (this φ) ∈ dom χ)⟶ χ a1 = χ' a1)"
proof (induction rule: exec.induct)
  case (exec_method_call φ x y m params stmts a χ C
                          paramValues M meth φ'' ψ)
  thus ?case
    by simp
next
  case (exec_var_assign φ x y stmts M ψ χ)
  thus ?case
    by simp
next
  case (exec_field_assign φ y x stmts v χ χ' M ψ)
  thus ?case
    by auto
next
  case (exec_new φ x C params stmts
                paramValues M c obj' a χ χ' ψ)
  hence "a ∉ dom χ"
    by simp
  thus ?case
    using exec_new
    by fastforce
```

```
next
  case (exec_return φ x stmts φ' x' stmts' M ψ χ)
  thus ?case
    by simp
qed

lemma Changed_FieldAssign_Aux:
  "M, σ →e σ' ⟹ σ = (φ#ψ,χ) ⟹
  σ' = (φ'#ψ',χ') ⟹ finite (dom χ) ⟹
   χ' (this φ) ≠ χ (this φ) ⟹ (this φ) ∈ dom χ ⟹
   ∃f x. x ∈ dom (vars φ) ⟶
   Some χ' = this_field_update φ χ f (the (vars φ x))"
proof (induction rule: exec.induct)
  case (exec_method_call φ x y m params stmts a χ C
                              paramValues M meth φ'' ψ)
  thus ?case
    by simp
next
  case (exec_var_assign φ x y stmts M ψ χ)
  thus ?case
    by simp
next
  case (exec_field_assign φ y x stmts v χ χ' M ψ)
  thus ?case
    using ident_lookup.elims Pair_inject
          list.inject option.sel
    by metis
next
  case (exec_new φ x C params stmts paramValues
                  M c obj' a χ χ' ψ)
  hence "a ∉ dom χ"
    by simp
  thus ?case
    using exec_new
    by fastforce
next
  case (exec_return φ x stmts φ' x' stmts' M ψ χ)
  thus ?case
    by simp
```

qed

```
lemma adapt_to_config_Aux:
  "finite (dom (vars φ)) ⟹
   φ'' = adapt_frame φ φ' ⟹
   w ∈ dom (vars φ'') ⟹
   w ∈ dom (vars φ) ⟹
   (vars φ'' w) = (vars φ w)"

  unfolding adapt_frame_def Let_def
  using Frame.select_convs(2) UnCI
        disjoint_iff_not_equal
        fresh_idents_empty
        map_upds_apply_nontin
  by metis

lemma adapt_to_config'_Aux:
  "finite (dom (vars φ)) ⟹
   φ'' = adapt_frame φ φ' ⟹
   w ∉ dom (vars φ) ⟹
   w ∈ dom (vars φ'')  ⟹
   ∃v. (v ∈ dom (vars φ') ⟶
  (vars φ'' w = vars φ' v) ∧ w ∉ dom (vars φ'))"

  unfolding adapt_frame_def Let_def
  using Frame.select_convs(2)
        fresh_nat_is_fresh
        list.simps(8)
        map_upds_Nil2
        sorted_list_of_set.infinite
  by metis
```

A.6 Lemmas aiding for Holistic assertions in Isabelle/HOL

The *object_Unchange* is the formalized proof of Lemma 6 in Section 4.1.

```
lemma object_Unchange:
  "M, σ →e σ' ⟹ σ = (φ#ψ,χ) ⟹
  σ' = (φ'#ψ',χ') ⟹ finite (dom χ) ⟹
  ∀a1.((a1 ≠ (this φ) ∧ a1 ∈ dom χ ∧
    (this φ) ∈ dom χ)⟶ χ a1 = χ' a1)"
  by (simp add: object_Unchange_Aux)
```

The `Changed_FieldAssign` is the formalized proof of Lemma 5 in Section 4.1.

```
lemma Changed_FieldAssign:
  "M, σ →e σ' ⟹ σ = (φ#ψ,χ) ⟹
  σ' = (φ'#ψ',χ') ⟹ finite (dom χ) ⟹
   χ' (this φ) ≠ χ (this φ) ⟹ (this φ) ∈ dom χ ⟹
   ∃f x. x ∈ dom (vars φ) ⟶
    Some χ' = this_field_update φ χ f (the (vars φ x))"
  using Changed_FieldAssign_Aux by blast
```

Here is a formalized proof of Lemma 7 mentioned in Section 4.1.

```
lemma adapt_to_config:
  "finite (dom (vars φ)) ⟹
   φ'' = adapt_frame φ φ' ⟹
   w ∈ dom (vars φ'') ⟹
   w ∈ dom (vars φ) ⟹
   (vars φ'' w) = (vars φ w)"
  using adapt_to_config_Aux by blast
```

Here is a formalized proof of Lemma 8 discussed in Section 4.1.

```
lemma adapt_to_config':
  "finite (dom (vars φ)) ⟹
   φ'' = adapt_frame φ φ' ⟹
   w ∉ dom (vars φ) ⟹
   w ∈ dom (vars φ'')  ⟹
   ∃v. (v ∈ dom (vars φ') ⟶
    (vars φ'' w = vars φ' v) ∧ w ∉ dom (vars φ'))"
  by (simp add: adapt_to_config'_Aux)
```

Lemma 9 is also stated as a formalized proof here.

```
lemma adapt_to_config_config':
  "finite (dom (vars φ)) ⟹
```

```
   φ'' = adapt_frame φ φ' ⟹
   w ∈ dom (vars φ'') ⟹
  (w ∉ dom (vars φ) ∧ (∃v. v ∈ dom (vars φ') ⟶
  vars φ'' w =  vars φ' v ∧ w ∉ dom (vars φ')) ∨
  (w ∈ dom (vars φ) ∧ (vars φ'' w = vars φ w)))"
  using adapt_to_config' adapt_to_config
  by meson
end
```

A.7 Partial Proofs of Theorem 3

Theorem 3. *Let σ' be a next configuration such that $M; M', \sigma \rightarrow_e \sigma'$.*
Let o_i, where $i = 1, 2$ be identifers such that $\lfloor o_1 \rfloor_\sigma = \lfloor \mathtt{this} \rfloor_\sigma$, $\lfloor o_2 \rfloor_{\sigma'} = \lfloor \mathtt{this} \rfloor_{\sigma'}$, $o_2 \in dom(\sigma)$ and $o_k \in dom(\sigma)$.
Also, let heaps be χ and χ' such that $\chi(\mathtt{this}) \notin dom(M)$ and $\chi'(\mathtt{this}) \notin dom(M)$.
Show that $\sigma \models \mathtt{Next}\ (o_i\ \mathtt{Access}\ o_k) \implies (\sigma \models (o_1\ \mathtt{Access}\ o_k)) \vee (\sigma \models (o_2\ \mathtt{Access}\ o_k))$.

Proof. We consider two cases of o_i, including $o_i = o_1$ and $o_i = o_2$.
In each case, we also call $o_k = o_3$, and $o_3 \in dom(\sigma)$.
To be convenient, we call LHS of the implication is $\sigma \models \mathtt{Next}\ (o_i\ \mathtt{Access}\ o_k)$,
and RHS is $(\sigma \models (o_1\ \mathtt{Access}\ o_k)) \vee (\sigma \models (o_2\ \mathtt{Access}\ o_k))$.

- **Case** $o_i = o_1$. We need to show
  $\sigma \models \mathtt{Next}\ (o_1\ \mathtt{Access}\ o_3) \implies (\sigma \models (o_1\ \mathtt{Access}\ o_3)) \vee (\sigma \models (o_2\ \mathtt{Access}\ o_3))$.
  It is proved in Lemma 10.

- **Case** $o_i = o_2$. We need to show
  $\sigma \models \mathtt{Next}\ (o_2\ \mathtt{Access}\ o_3) \implies (\sigma \models (o_1\ \mathtt{Access}\ o_3)) \vee (\sigma \models (o_2\ \mathtt{Access}\ o_3))$.
  It is proved in Lemma 11.

As a result, we have
$\sigma \models \mathtt{Next}\ (o_i\ \mathtt{Access}\ o_k) \implies (\sigma \models (o_1\ \mathtt{Access}\ o_k)) \vee (\sigma \models (o_2\ \mathtt{Access}\ o_k))$. □

Lemma 10. *Let σ' be a next configuration such that $M; M', \sigma \rightarrow_e \sigma'$.*
Let o_1 and o_2 be identifers such that $\lfloor o_1 \rfloor_\sigma = \lfloor \mathtt{this} \rfloor_\sigma$, $\lfloor o_2 \rfloor_{\sigma'} = \lfloor \mathtt{this} \rfloor_{\sigma'}$, $o_2 \in dom(\sigma)$, and $o_3 \in dom(\sigma)$, $\chi(\mathtt{this}) \notin dom(M)$ and $\chi'(\mathtt{this}) \notin dom(M)$.
Show that $\sigma \models \mathtt{Next}\ (o_1\ \mathtt{Access}\ o_3) \implies (\sigma \models (o_1\ \mathtt{Access}\ o_3)) \vee (\sigma \models (o_2\ \mathtt{Access}\ o_3))$.

Proof. To be convenient, we call LHS of the implication is $\sigma \models \texttt{Next}\ (o_1\ \texttt{Access}\ o_3)$, and the RHS is $(\sigma \models (o_1\ \texttt{Access}\ o_3)) \vee (\sigma \models (o_2\ \texttt{Access}\ o_3))$.
From LHS of the implication of the formula, $\sigma \models \texttt{Next}\ (o_1\ \texttt{Access}\ o_3)$ holds if

$$(\sigma \lhd \sigma') \models (o_1\ \texttt{Access}\ o_3).$$

It is equivalent to
$(\lfloor o_1 \rfloor_{\sigma\lhd\sigma'} = \lfloor o_3 \rfloor_{\sigma\lhd\sigma'}) \vee (\lfloor o_1.f \rfloor_{\sigma\lhd\sigma'} = \lfloor o_3 \rfloor_{\sigma\lhd\sigma'}) \vee$
$[(\lfloor o_1 \rfloor_{\sigma\lhd\sigma'} = \lfloor \texttt{this} \rfloor_{\sigma\lhd\sigma'}) \wedge (\lfloor o_3 \rfloor_{\sigma\lhd\sigma'} = \lfloor \texttt{z}' \rfloor_{\sigma\lhd\sigma'})]$,
where $\texttt{z}'$ appears in $(\sigma \lhd \sigma').\texttt{cont}'$.

- **Case** $\lfloor o_1 \rfloor_{\sigma\lhd\sigma'} = \lfloor o_3 \rfloor_{\sigma\lhd\sigma'}$.
  We have $\lfloor o_1 \rfloor_{\sigma\lhd\sigma'}$ and $\lfloor o_3 \rfloor_{\sigma\lhd\sigma'}$ are defined in $\sigma \lhd \sigma'$.
  Also, $\lfloor o_1 \rfloor_{\sigma\lhd\sigma'} = \lfloor o_1 \rfloor_{\sigma}$ and $\lfloor o_3 \rfloor_{\sigma\lhd\sigma'} = \lfloor o_3 \rfloor_{\sigma}$, since o_1 and o_3 are defined in σ (See Lemma 7).
  Therefore, we have $\lfloor o_1 \rfloor_{\sigma} = \lfloor o_3 \rfloor_{\sigma}$ in σ that implies $o_1\ \texttt{Access}ø_3$ in σ.
  So, we have RHS of the implication of the formula.

- **Case** $\lfloor o_1.f \rfloor_{\sigma\lhd\sigma'} = \lfloor o_3 \rfloor_{\sigma\lhd\sigma'}$, with some field f.
  The configuration $\sigma \lhd \sigma'$ and σ' use the same heap χ'.
  We can rewrite $\lfloor o_1.f \rfloor_{\sigma\lhd\sigma'} = v$ for some v, and $\chi'(\phi(\texttt{this})) = (C, fldMap)$, where $fldMap(f) = v$, since $\lfloor o_1 \rfloor_{\sigma\lhd\sigma'}$ is defined, $\lfloor o_1 \rfloor_{\sigma\lhd\sigma'} = \lfloor o_1 \rfloor_{\sigma}$, and $\lfloor o_1 \rfloor_{\sigma} = \lfloor \texttt{this} \rfloor_{\sigma}$.
  On the other hand, $\lfloor o_3 \rfloor_{\sigma\lhd\sigma'} = \lfloor o_3 \rfloor_{\sigma}$. Hence, $v = \lfloor o_3 \rfloor_{\sigma}$.
  We consider two cases which are $\chi'(\phi(\texttt{this})) = \chi(\phi(\texttt{this}))$ and $\chi'(\phi(\texttt{this})) \neq \chi(\phi(\texttt{this}))$.

  – Case $\chi'(\phi(\texttt{this})) = \chi(\phi(\texttt{this}))$. We have $\lfloor o_1.f \rfloor_{\sigma} = \lfloor o_3 \rfloor_{\sigma}$ obviously.

  – Case $\chi'(\phi(\texttt{this})) \neq \chi(\phi(\texttt{this}))$.
  From the Lemma 5, there exists a field f and an identifier x such that $x \in dom(\sigma)$ such that $\chi'(\phi(\texttt{this})) = \texttt{field\_update}(\chi(\phi(\texttt{this}), f, \phi(x)))$. Therefore, $fldMap(f) = \phi(x)$ and we also have $\phi(x) = \phi(o_3)$. We have o_1 Access o_3 in σ. It implies the RHS of the implication of the formula.

- **Case** $(\lfloor o_1 \rfloor_{\sigma\lhd\sigma'} = \lfloor \texttt{this} \rfloor_{\sigma\lhd\sigma'}) \wedge (\lfloor o_3 \rfloor_{\sigma\lhd\sigma'} = \lfloor \texttt{z} \rfloor_{\sigma\lhd\sigma'})$, where $\texttt{z}'$ appears in $(\sigma \lhd \sigma').\texttt{cont}'$.
  Note that: We have not finished this case.

□

Lemma 11. *Let σ' be a next configuration such that $M; M', \sigma \to_e \sigma'$.*
Let o_1 and o_2 be identifers such that $\lfloor o_1 \rfloor_\sigma = \lfloor this \rfloor_\sigma$, $\lfloor o_2 \rfloor_{\sigma'} = \lfloor this \rfloor_{\sigma'}$, $o_2 \in dom(\sigma)$, and $o_3 \in dom(\sigma)$, $\chi(this) \notin dom(M)$ and $\chi'(this) \notin dom(M)$.
Show that $\sigma \models \mathtt{Next}\ (o_2\ \mathtt{Access}\ o_3) \implies (\sigma \models (o_1\ \mathtt{Access}\ o_3)) \vee (\sigma \models (o_2\ \mathtt{Access}\ o_3))$.

Proof. To be convenient, we call LHS of the implication is $\sigma \models \mathtt{Next}\ (o_2\ \mathtt{Access}\ o_3)$, and RHS is
$(\sigma \models (o_1\ \mathtt{Access}\ o_3)) \vee (\sigma \models (o_2\ \mathtt{Access}\ o_3))$.

From LHS of the implication of the formula, $\sigma \models \mathtt{next}\ (o_2\ \mathtt{Access}\ o_3)$ holds if

$$(\sigma \lhd \sigma') \models (o_2\ \mathtt{Access}\ o_3) \quad (1)$$

We rewrite (1) as an equivalent form as follows.
$(\lfloor o_2 \rfloor_{\sigma\lhd\sigma'} = \lfloor o_3 \rfloor_{\sigma\lhd\sigma'}) \vee (\lfloor o_2.f \rfloor_{\sigma\lhd\sigma'} = \lfloor o_3 \rfloor_{\sigma\lhd\sigma'}) \vee$
$[(\lfloor o_2 \rfloor_{\sigma\lhd\sigma'} = \lfloor \mathtt{this} \rfloor_{\sigma\lhd\sigma'}) \wedge (\lfloor o_3 \rfloor_{\sigma\lhd\sigma'} = \lfloor \mathtt{z}' \rfloor_{\sigma\lhd\sigma'})]$,
where $\mathtt{z}'$ appears in $(\sigma \lhd \sigma').\mathtt{cont}'$.

- **Case** $\lfloor o_2 \rfloor_{\sigma\lhd\sigma'} = \lfloor o_3 \rfloor_{\sigma\lhd\sigma'}$.
  We have $\lfloor o_2 \rfloor_{\sigma\lhd\sigma'}$ and $\lfloor o_3 \rfloor_{\sigma\lhd\sigma'}$ are defined in $\sigma \lhd \sigma'$.
  Also, we have $\lfloor o_2 \rfloor_{\sigma\lhd\sigma'} = \phi(o_2)$, and $\lfloor o_3 \rfloor_{\sigma\lhd\sigma'} = \phi(o_3)$, since o_2 and o_3 are defined in σ (See Lemma 7). Therefore, we have $\lfloor o_2 \rfloor_\sigma = \lfloor o_3 \rfloor_\sigma$ in σ and it implies $o_2\ \mathtt{Access}\ o_3$ in σ. From that, we have RHS of the implication of the formula.

- **Case** $\lfloor o_2.f \rfloor_{\sigma\lhd\sigma'} = \lfloor o_3 \rfloor_{\sigma\lhd\sigma'}$, with some field f.
  Moreover, $\sigma \lhd \sigma'$ and σ' use the same heap χ'. We can rewrite $\lfloor o_2.f \rfloor_{\sigma\lhd\sigma'} = v$ for some v, and $\chi'(\phi(o_2)) = (C, fldMap)$, where $fldMap(f) = v$, since $\lfloor o_2 \rfloor_{\sigma\lhd\sigma'}$ is defined, and $\phi'(o_2) = \phi(o_2)$.
  Also, $\lfloor o_3 \rfloor_{\sigma\lhd\sigma'} = \phi(o_3)$. Hence, $v = \phi(o_3)$.
  Here, we consider two cases. These are $\phi(o_2) \neq \phi(o_1)$ and $\phi(o_2) = \phi(o_1)$.

  - Case $\phi(o_2) \neq \phi(o_1)$. It means $\phi(o_2) \neq \phi(\mathtt{this})$ and from the Lemma 6, we also have $\chi'(\phi(o_2)) = \chi(\phi(o_2))$.
    Hence, $\chi(\phi(o_2)) = \phi(o_3)$ in σ, since $v = \phi(o_3)$. From that, we have $o_2\ \mathtt{Access}\ o_3$. It implies RHS of the implication of the formula.

  - Case $\phi(o_2) = \phi(o_1)$. It means $\phi(o_2) = \phi(\mathtt{this})$. We consider two cases which are $\chi'(\phi(\mathtt{this})) = \chi(\phi(\mathtt{this}))$ and $\chi'(\phi(\mathtt{this})) \neq \chi(\phi(\mathtt{this}))$.

    * Case $\chi'(\phi(\mathtt{this})) = \chi(\phi(\mathtt{this}))$. We have $\lfloor o_2.f \rfloor_\sigma = \lfloor o_3 \rfloor_\sigma$ obviously.

  * Case $\chi'(\phi(\texttt{this})) \neq \chi(\phi(\texttt{this}))$.
    From the Lemma 5, there exists a field f and an identifier x such that $x \in dom(\sigma)$ such that $\chi'(\phi(\texttt{this})) = \texttt{field\_update}(\chi(\phi(\texttt{this}), f, \phi(x)))$. Therefore, $fldMap(f) = \phi(x)$ and we also have $\phi(x) = \phi(o_3)$. We have o_2 `Access` o_3 in σ. It implies RHS of the implication of the formula.

- **Case** $(\lfloor o_2 \rfloor_{\sigma\triangleleft\sigma'} = \lfloor \texttt{this} \rfloor_{\sigma\triangleleft\sigma'}) \wedge (\lfloor o_3 \rfloor_{\sigma\triangleleft\sigma'} = \lfloor \texttt{z}' \rfloor_{\sigma\triangleleft\sigma'})$, where $\texttt{z}'$ appears in $(\sigma\triangleleft\sigma').\texttt{cont}'$.
  Note that: We have not finished this case.

$\square$

Bibliography

[1] Mike Barnett, K Rustan M Leino, and Wolfram Schulte. The spec# programming system: An overview. In *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 49–69. Springer, 2004.

[2] Andrew P Black, Kim B Bruce, Michael Homer, and James Noble. Grace: the absence of (inessential) difficulty. In *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*, pages 85–98, 2012.

[3] Gilad Bracha. *The Dart Programming Language*. Addison-Wesley Professional, 2015.

[4] Dominique Devriese, Lars Birkedal, and Frank Piessens. Reasoning about object capabilities with logical relations and effect parametricity. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 147–162. IEEE, 2016.

[5] Sophia Drossopoulou and James Noble. The need for capability policies. In *Proceedings of the 15th Workshop on Formal Techniques for Java-like Programs*, pages 1–7, 2013.

[6] Sophia Drossopoulou, James Noble, and Mark S Miller. Swapsies on the internet: First steps towards reasoning about risk and trust in an open world. In *Proceedings of the 10th ACM Workshop on Programming Languages and Analysis for Security*, pages 2–15, 2015.

[7] Sophia Drossopoulou, James Noble, Toby Murray, and Mark S Miller. Reasoning about risk and trust in an open world. 2015.

[8] Sophia Drossopoulou, James Noble, Julian Mackay, and Susan Eisenbach. Holistic specifications for robust programs. In *FASE*, pages 420–440, 2020.

[9] Christoph Jentzsch. Decentralized autonomous organization to automate governance. *White paper, November*, 2016.

[10] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. *ACM SIGPLAN Notices*, 50(1):637–650, 2015.

[11] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, pages 256–269, 2016.

[12] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. The essence of higher-order concurrent separation logic. In *European Symposium on Programming*, pages 696–723. Springer, 2017.

[13] Gary T Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M Zimmerman, et al. Jml reference manual, 2008.

[14] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer, 2010.

[15] K Rustan M Leino and Wolfram Schulte. Using history invariants to verify observers. In *European Symposium on Programming*, pages 80–94. Springer, 2007.

[16] Sergio Maffeis, John C Mitchell, and Ankur Taly. Object capabilities and isolation of untrusted web applications. In *2010 IEEE Symposium on Security and Privacy*, pages 125–140. IEEE, 2010.

[17] Darya Melicher, Yangqingwei Shi, Alex Potanin, and Jonathan Aldrich. A capability-based module system for authority control. In *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[18] Adrian Mettler, David A Wagner, and Tyler Close. Joe-e: A security-oriented subset of java. In *NDSS*, volume 10, pages 357–374, 2010.

[19] Bertrand Meyer. Applying “Design by Contract”. *Computer*, 25(10):40–51, 1992.

[20] Bertrand Meyer. *Object-oriented software construction*, volume 2. Prentice hall Englewood Cliffs, 1997.

[21] M Miller. Robust composition: Towards a unified approach to access control and concurrency control 2006. *Johns Hopkins: Baltimore, MD*, page 302, 2006.

[22] Mark S Miller, Chip Morningstar, and Bill Frantz. Capability-based financial instruments. In *International Conference on Financial Cryptography*, pages 349–378. Springer, 2000.

[23] Mark S Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja: Safe active content in sanitized javascript, 2008. *Google white paper*, 2009.

[24] Mark S Miller, Tom Van Cutsem, and Bill Tulloh. Distributed electronic rights in javascript. In *European Symposium on Programming*, pages 1–20. Springer, 2013.

[25] Toby C Murray. *Analysing the security properties of object-capability patterns.* PhD thesis, University of Oxford, UK, 2010.

[26] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[27] James Noble and Sophia Drossopoulou. Rationally reconstructing the escrow example. In *Proceedings of 16th Workshop on Formal Techniques for Java-like Programs*, pages 1–6, 2014.

[28] David J Pearce and Lindsay Groves. Designing a verifying compiler: Lessons learned from developing whiley. *Science of Computer Programming*, 113:191–220, 2015.

[29] Alexander J Summers and Sophia Drossopoulou. Considerate reasoning and the composite design pattern. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 328–344. Springer, 2010.

[30] David Swasey, Deepak Garg, and Derek Dreyer. Robust and compositional verification of object capability patterns. *Proc. ACM Program. Lang.*, 1(OOPSLA):89–1, 2017.