A Unified Framework for Verification Techniques for Object Invariants (Full Version of ECOOP'08 Paper)

date: 24th April 2008
please check this site for updates and extensions of this paper

Sophia Drossopoulou   Adrian Francalanza
Alexander J. Summers
Imperial College
{S.Drossopoulou,A.Francalanza,Alexander.J.Summers}@imperial.ac.uk

Peter Müller
Microsoft Research, Redmond
mueller@microsoft.com

Abstract

Verification of object-oriented programs relies on object invariants which express consistency criteria of objects. The semantics of object invariants is subtle, mainly because of call-backs, multi-object invariants, and subclassing.

Several verification techniques for object invariants have been proposed. These techniques are complex and differ in restrictions on programs (e.g., which fields can be updated), restrictions on invariants (what an invariant may refer to), use of advanced type systems (such as Universe types or ownership), meaning of invariants (in which execution states are invariants assumed to hold), and proof obligations (when should an invariant be proven). As a result, it is difficult to understand whether/why these techniques are sound and to compare their expressiveness. This general lack of understanding also hampers the development of new approaches.

In this paper, we develop and formalise a unified framework to describe verification techniques for object invariants. We distil seven parameters, which characterise a verification technique, and identify sufficient conditions on these parameters under which a verification technique is sound. To illustrate the generality of our framework, we instantiate it with six verification techniques from the literature. We show how our framework facilitates the assessment and comparison of the soundness and expressiveness of these techniques.

1. Introduction

Object invariants play a crucial role in the verification of object-oriented programs, and have been an integral part of all major contract languages such as Eiffel [25], the Java Modeling Language JML [17], and Spec# [2]. Object invariants express consistency criteria for objects, which guarantee their correct working. These criteria range from simple properties of single objects (for instance, that a field is non-null) to complex properties of whole object structures (for instance, the sorting of a tree).

```
class C {                              class Client {
  int a, b;                              C c;
  invariant 0 <= a < b;                  invariant c.a <= 10;

  C() { a := 0; b := 3; }                /* methods omitted */
                                       }
  void m() {
    int k := 100 / (b − a);
    a := a + 3;
    n();                               class D extends C {
    b := (k + 4) ∗ b;                    invariant a <= 10;
  }
  void n() { m(); }                      /* methods omitted */
}                                      }
```

Figure 1. An example (adapted from [18]) illustrating the three main challenges for the verification of object invariants.

Most of the existing verification techniques expect object invariants to hold in the pre-state and post-state of method executions, often referred to as *visible states* [28]. Invariants may be violated temporarily between visible states. This semantics is illustrated by class C in Fig. 1. The invariant is established by the constructor. It may be assumed in the pre-state of method m. Therefore, the first statement in m's body can be proven not to cause a division-by-zero error. The invariant might temporarily be violated by the subsequent assignment to a, but it is later re-established by m's last statement; thus, the invariant holds in m's post-state.

While the basic idea of object invariants is simple, verification techniques for practical OO-programs face challenges. These challenges are made more daunting by the common expectation that classes should be verified without knowledge of their clients and subclasses:

Call-backs: Methods that are called while the invariant of an object o is temporarily broken might call back into o and find the object in an inconsistent state. In our example (Fig. 1), during execution of **new** C().m() the assignment to a violates the invariant, and the call-back via n() leads to a division by zero.

Multi-object invariants: When the invariant of an object p depends on the state of another object o, modifications of o potentially violate the invariant of p. In our example, a call $o.\mathsf{m}$ might break the invariant of a Client object p where $p.\mathsf{c} = o$. Aliasing makes the proof of preservation of p's invariant difficult. In particular, when verifying o, the invariant of p may not be known and, if not, cannot be expected to be preserved.

Subclassing: When the invariant of a subclass D refers to fields declared in the superclass C then methods of C potentially violate D's invariant by assigning to C's fields. In particular, when verifying a class, its subclass invariants are not known in general, and so cannot be expected to be preserved.

A number of verification techniques have been suggested to address some or all of these problems [1, 3, 14, 16, 18, 23, 26, 27, 28, 32]. These techniques share many commonalities, but differ in the following important aspects:

1. *Invariant semantics:* What invariants are expected to hold in which execution states? Some techniques require all invariants to hold in all visible states, whereas others address the multi-object invariant challenge by excluding certain invariants.
2. *Proof obligations:* What is required to be proven? Some techniques require proofs for invariants relating to the current active object whereas others require invariant proofs for all objects in the heap.
3. *Invariant restrictions:* What objects may invariants depend on? Some techniques use unrestricted invariants, whereas others address the subclassing challenge by preventing invariants from referring to inherited fields.
4. *Program restrictions:* What objects may be used as receivers of field updates and method calls? Some techniques permit arbitrary field updates, whereas others simplify verification by allowing updates to fields of the current receiver only.
5. *Type systems:* What syntactic information is used for reasoning? Some techniques are designed for arbitrary programs, whereas others use ownership types to facilitate verification.

These differences, together with the fact that most verification techniques are not formally specified, complicate the comparison of verification techniques, and hinder the understanding of why these techniques satisfy claimed properties such as soundness. For these reasons, it is hard to decide which technique to adopt, or to develop new sound techniques.

In this paper, we present a unified framework for verification techniques for object invariants. This framework formalises verification techniques in terms of seven parameters, which abstract away from differences pertaining to language features (type system, specification language, and logics) and highlight characteristics intrinsic to the techniques, thereby aiding comparisons. Subsets of these parameters describe aspects applicable to all verification techniques; for example, a generic definition of *soundness* is given in terms of two framework parameters, expressivity is captured by three other parameters.

We concentrate on techniques that require invariants to hold in the pre-state and post-state of a method execution (often referred to as *visible states* [28]) while temporary violations between visible states are permitted. These techniques constitute the vast majority of those described in the literature.

Contributions. The contributions of this paper are:

1. We present a unified formalism for object invariant verification techniques.
2. We identify conditions on the framework that guarantee soundness of a verification technique.
3. We separate type system concerns from verification strategy concerns.
4. We show how our framework describes some advanced verification techniques for visible state invariants.
5. We prove soundness for a number of techniques, and, guided by our framework, discover an unsoundness in one technique.

Our framework allows the extraction of comparable data from techniques that were presented using different concepts, terminology and styles. Comparative value judgements concerning the techniques are beyond the scope of our paper.

Outline. Sec. 2 gives an overview of our work, explaining the important concepts. Sec. 3 formalises program and invariant semantics. Sec. 4 describes our framework and defines soundness. Sec. 6 instantiates our framework with existing verification techniques. Sec. 5 presents sufficient conditions for a verification technique to be sound, and states a general soundness theorem. Sec. 7 discusses related work. Proofs and more details are in the companion report [8]. This paper follows our FOOL paper [7], but provides more explanations and examples.

2. Example and Approach

Example. Consider a scenario, in which a Person holds an Account, and has a salary. An Account has a balance, an interestRate and an associated DebitCard. This example will be used throughout the paper. We give the code in Fig. 2.

Account's interestRate is required to be zero when the balance is negative (I1). A further invariant (the two can be read as conjuncts of the full invariant for the class) ensures that the DebitCard associated with an account has a consistent reference back to the account (I2). A SavingsAccount is a special kind of Account, whose balance must be positive (I3). Person's invariant (I4) requires that the sum of salary and account's balance is positive. Finally, DebitCard's invariant (I5) requires dailyCharges not to exceed the balance of the associated account. Thus, I2, I4, and I5 are multi-object invariants.

To illustrate the challenges faced by verification techniques, suppose that p is an object of class Person, which holds the Account a with DebitCard d:

Call-backs: When p executes its method spend, this results in a call of withdraw on a, which (via a call to sendReport) eventually calls back notify on p; the call notify might reach p in a state where I4 does not hold.

Multi-object invariants: When a executes its method withdraw , it may temporarily break its invariant I1, since its balance is debited before any corresponding change is made to its interestRate . This violation is not important according to the visible state semantics; the **if** statement immediately afterwards ensures that the invariant is restored before the next visible state. However, by making an unrestricted reduction of the account balance, the method potentially breaks the invariants of other objects as well. In particular, p's invariant I4, and d's invariant I5 may be broken.

Subclassing: Further to the previous point, if a is a SavingsAccount , then calling the method withdraw may break the invariant I3, which was not necessarily known during the verification of class Account.

These points are addressed in the literature by striking various trade-offs between the differing aspects listed in the introduction.

Approach. Our framework uses seven parameters to capture the first four aspects in which verification techniques differ, i.e., invariant semantics, invariant restrictions, proof obligations and program

```
 class Account {
  Person holder;
  DebitCard card;
  int balance, interestRate;

  // invariant I1: balance < 0 ==>
       interestRate == 0;
  // invariant I2: card.acc == this;

  void withdraw(int amount) {
    balance -= amount;
    if (balance < 0) {
      interestRate = 0;
      this.sendReport();
    }
  }

  void sendReport()
    { holder.notify(); }
}

class SavingsAccount
             extends Account {
  // invariant I3: balance >= 0;
}

class Person {
  Account account;
  int salary;

  // invariant I4:
  //   account.balance + salary > 0;

  void spend(int amount)
    { account.withdraw(amount); }

  void notify()
    { ... }
}

class DebitCard {
  Account acc;
  int dailyCharges;

  // invariant I5:
  //   dailyCharges <= acc.balance;
}
```

Figure 2. An account example illustrating the main challenges for the verification of object invariants. We assume that fields hold non-null values.

restrictions. To describe these parameters we use two abstract notions, which we call *regions* and *properties*. A *region* (when interpreted semantically) describes a set of objects (e.g., those on which a method may be called), while a property describes a set of invariants (e.g., the invariants that have to be proven before a method call). We deal with the aspects identified in the previous section as follows:

1. *Invariant semantics:* The property $\mathbb{X}$ describes the invariants e$\underline{\text{x}}$pected to hold in visible states. The property $\mathbb{V}$ describes the invariants *$\underline{v}$ulnerable* to a given method, i.e., those which may be broken while the method executes.
2. *Invariant restrictions:* The property $\mathbb{D}$ describes the invariants that may $\underline{\text{d}}$epend on a given heap location. This also characterises indirectly the locations an invariant may depend on.
3. *Proof obligations:* The properties $\mathbb{B}$ and $\mathbb{E}$ describe the invariants that must be proven to hold $\underline{\text{b}}$efore a method call and at the $\underline{\text{e}}$nd of a method body, respectively.
4. *Program restrictions:* The regions $\mathbb{U}$ and $\mathbb{C}$ describe the permitted receivers for field $\underline{\text{u}}$pdates and method $\underline{\text{c}}$alls, respectively.
5. *Type systems:* We parameterise our framework by the type system. We state requirements on the type system, but leave abstract its concrete definition. We require that types are formed of a region-class pair so that we can handle types that express heap topologies (such as ownership types).
6. *Specification languages:* Rather than describing invariants concretely, we assume a judgement that expresses that an object satisfies the invariant of a class in a heap.
7. *Verification logics:* We express proof obligations via a special construct prv $\mathbb{p}$, which throws an exception if the invariants in property $\mathbb{p}$ cannot be proven, and has an empty effect otherwise. We leave abstract how the actual proofs are constructed and checked.

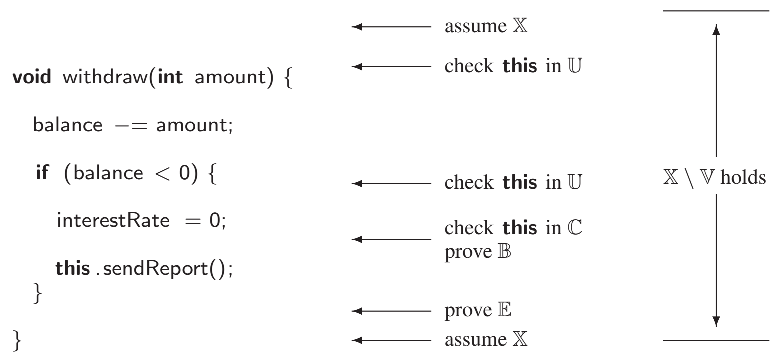
Fig. 3 illustrates the parameters of our framework by annotating the body of the method withdraw. $\mathbb{X}$ may be assumed to hold in the pre- and post-states of the method. Between these visible states, some object invariants may be broken (so long as they fall within $\mathbb{V}$), but $\mathbb{X} \setminus \mathbb{V}$ is known to hold throughout the method body. Field updates and method calls are allowed if the receiver object (here, **this**) is in $\mathbb{U}$ and $\mathbb{C}$, respectively. Before a method call, $\mathbb{B}$ must be proven. At the end of the method body, $\mathbb{E}$ must be proven. Finally, $\mathbb{D}$ (not shown in Fig. 3) constrains the effects of field updates on invariants. Thus, assignments to balance and interestRate affect at most $\mathbb{D}$.

Developing our framework was challenging because different verification techniques (1) use different type systems to restrict programs and invariants, and do not make a clear distinction between the type system and the verification technique, (2) use different specification languages to express invariants, and (3) use different verification logics. To deal with this diversity within one unified framework, we take the following approach:

1. We make a clear delineation between the framework and the type system and instead of describing one particular type system, we state requirements on the type systems used with our framework.
2. We assume a judgment that describes that an object satisfies the invariant of a class in a heap. We require that a field update preserves the invariant if it does not fall within $\mathbb{D}$.
3. We express proof obligations via a special construct prv $\mathbb{p}$, which throws an exception if the invariants in property $\mathbb{p}$ cannot be proven, and has an empty effect otherwise.

The number of parameters reflects the variety of concepts used by verification techniques, such as accessibility of fields, purity, helper methods, ownership, and effect specifications. For instance, $\mathbb{V}$ would be redundant if all methods were to re-establish the invariants they break; in such a setting, a method could break invariants only through field updates, and $\mathbb{V}$ could be derived from $\mathbb{U}$ and $\mathbb{D}$. However, in general, methods may break but not re-establish invariants.

The seven parameters capture concepts explicitly or implicitly found in all verification techniques, defined either through words [28, 14, 16, 32] or typing rules [23]. For example, $\mathbb{V}$ is implicit in [28], but is crucial for their soundness argument. $\mathbb{X}$ and $\mathbb{V}$ are explicit in [23], while $\mathbb{U}$ and $\mathbb{C}$ are implicitly expressed as constraints in their typing rules. Subsets of these seven parameters characterise



```
void withdraw(int amount) {
  balance −= amount;
  if (balance < 0) {
    interestRate = 0;
    this.sendReport();
  }
}
```

assume $\mathbb{X}$
check **this** in $\mathbb{U}$
check **this** in $\mathbb{U}$
check **this** in $\mathbb{C}$
prove $\mathbb{B}$
prove $\mathbb{E}$
assume $\mathbb{X}$
$\mathbb{X} \setminus \mathbb{V}$ holds

Figure 3. Role of framework parameters for method withdraw from Fig. 2.

verification technique concepts e.g., soundness (through $\mathbb{X}$ and $\mathbb{V}$), expressiveness ($\mathbb{D}$, $\mathbb{X}$ and $\mathbb{V}$), proof obligations ($\mathbb{B}$ and $\mathbb{E}$).

3. Invariant Semantics

We formalise invariant semantics through an operational semantics, defining at which execution points invariants are required to hold. In order to cater for the different techniques, the semantics is parameterised by properties to express proof obligations and which invariants are expected to hold. In this section, we focus on the main ideas of our semantics and relegate the less interesting definitions to App. **??**. We assume sets of identifiers for class names CLS, field names FLD, and method names MTHD, and use variables $c \in \text{CLS}$, $f \in \text{FLD}$ and $m \in \text{MTHD}$.

Runtime Structures. A *runtime structure* is a tuple consisting of a set of heaps HP, a set of addresses ADR, and a set of values $\text{VAL} = \text{ADR} \cup \{\text{null}\}$, using variables $h \in \text{HP}$, $\iota \in \text{ADR}$, and $v \in \text{VAL}$. A runtime structure provides the following operations. The operation $\text{dom}(h)$ represents the domain of the heap. $\text{cls}(h, \iota)$ yields the class of the object at address ι. The operation $\text{fld}(h, \iota, f)$ yields the value of a field f of the object at address ι. Finally, $\text{upd}(h, \iota, f, v)$ yields the new heap after a field update, and $\text{new}(h, \iota, t)$ yields the heap and address resulting from the creation of a new object of type t. We leave abstract how these operations work, but require properties about their behaviour, for instance that upd only modifies the corresponding field of the object at the given address, and leaves the remaining heap unmodified. See Def. 27 in App. **??** for details.

A stack frame $\sigma \in \text{STK} = \text{ADR} \times \text{ADR} \times \text{MTHD} \times \text{CLS}$ is a tuple of a receiver address, an argument address, a method identifier, and a class. The latter two indicate the method currently being executed and the class where it is defined.

Regions, Properties and Types. A region $\mathbb{r} \in \mathbf{R}$ is a syntactic representation for a set of objects; a property $\mathbb{p} \in \mathbf{P}$ is a syntactic representation for a set of assertions about particular objects. It is crucial that our syntax is parametric with the specific regions and properties; we use different regions and properties to model different verification techniques.[1]

We define a type $t \in \text{TYP}$, as a pair of a region and a class. The region allows us to cater for types that express the topology of the heap, without being specific about the underlying type system.

Expressions. In Fig. 4, we define source expressions $e \in \text{EXPR}$. In order to simplify our presentation (but without loss of generality), we restrict methods to always have exactly one argument. Besides the usual basic object-oriented constructs, we include proof annotations e prv $\mathbb{p}$. As we will see later, such a proof annotation

e ::=	this	(this)	| x	(variable)
|	null	(null)	| new t	(new object)
|	$e.f$	(access)	| $e.f := e$	(assignment)
|	$e.m(e)$	(method call)	| e prv $\mathbb{p}$	(proof annotat.)
e_r ::=	...	(as source exprs.)	| v	(value)
|	verfExc	(verif exc.)	| fatalExc	(fatal exc.)
|	$\sigma \cdot \text{e}_r$	(nested call)	| call e_r	(launch)
|	ret e_r	(return)		

Figure 4. Source and runtime expression syntax.

executes the expression e and then imposes a proof obligation for the invariants characterised by the property $\mathbb{p}$. To maintain generality, we avoid being precise about the actual syntax and checking of proofs.

In Fig. 4, we also define runtime expressions $\text{e}_r \in \text{REXPR}$. A runtime expression is a source expression, a value, a nested call with its stack frame σ, an exception, or a decorated runtime expression. A verification exception verfExc indicates that a proof obligation failed. A fatal exception fatalExc indicates that an expected invariant does not hold. Runtime expressions can be decorated with call e_r and ret e_r to mark the beginning and end of a method call, respectively.

In Def. 29 (App. **??**), we define evaluation contexts, $E[\cdot]$, which describe contexts within one activation record and extend these to runtime contexts, $F[\cdot]$, which also describe nested calls.

Programming Languages. We define a programming language as a tuple consisting of a set PRG of programs, a runtime structure, a set of regions, and a set of properties (see Def. 30 in App. **??**). Each $\text{P} \in \text{PRG}$ comes equipped with the following operations. $\mathcal{F}(c, f)$ yields the type of field f in class c as well as the class in which f is declared (c or a superclass of c). $\mathcal{M}(c, m)$ yields the type signature of method m in class c. $\mathcal{B}(c, m)$ yields the expression constituting the body of method m in class c as well as the class in which m is declared. Moreover, there are operators to denote subclasses and subtypes ($<:$), inclusion of regions ($\sqsubseteq$), and interpretation ($[\![\cdot]\!]$) of regions and properties.

The interpretation of a region produces a set of objects. We characterise each invariant by an object-class pair, with the intended meaning that the invariant specified in the class holds for the object.[2] Therefore, the interpretation of a property produces a set of object-class pairs, specifying all the invariants of interest. Regions and properties are interpreted wrt. a heap, and from the *viewpoint* of a "current object"; therefore, their definitions depend on heap and address parameters: $[\![\ldots]\!]_{h,\iota}$.

[1] For example, in Universe types, **rep** and **peer** are regions, while in ownership types, ownership parameters such as X, and also **this**, are regions (more in Sec. 6).

[2] An object may have different invariants for each of the classes it belongs to [18].

(rVarThis)
$$\frac{\sigma = (\iota, v, \_, \_)}{\begin{array}{c}\sigma\cdot\mathsf{this},\ h \longrightarrow \sigma\cdot\iota,\ h\\ \sigma\cdot x,\ h \longrightarrow \sigma\cdot v,\ h\end{array}}$$

(rNew)
$$\frac{\begin{array}{l}\sigma = (\iota, \_, \_, \_)\\ h', \iota' = \mathrm{new}(h, \iota, t)\end{array}}{\sigma\cdot\mathsf{new}\, t,\ h \longrightarrow \sigma\cdot\iota',\ h'}$$

(rDer)
$$\frac{v = \mathrm{fld}(h, \iota, f)}{\sigma\cdot\iota.f,\ h \longrightarrow \sigma\cdot v,\ h}$$

(rAss)
$$\frac{h' = \mathrm{upd}(h, \iota, f, v)}{\sigma\cdot\iota.f := v,\ h \longrightarrow \sigma\cdot v,\ h'}$$

(rCall)
$$\frac{\mathcal{B}(m, \mathrm{cls}(h, \iota)) = e, c \quad \sigma' = (\iota, v, c, m)}{\sigma\cdot\iota.m(v),\ h \longrightarrow \sigma\cdot\sigma'\cdot\mathsf{call}\, e,\ h}$$

(rCxtEval)
$$\frac{\sigma\cdot\mathrm{e}_r,\ h \longrightarrow \sigma\cdot\mathrm{e}_r',\ h'}{\sigma\cdot E[\mathrm{e}_r],\ h \longrightarrow \sigma\cdot E[\mathrm{e}_r'],\ h'}$$

(rCxtFrame)
$$\frac{\mathrm{e}_r,\ h \longrightarrow \mathrm{e}_r',\ h'}{\sigma\cdot\mathrm{e}_r,\ h \longrightarrow \sigma\cdot\mathrm{e}_r',\ h'}$$

(rLaunch)
$$\frac{\begin{array}{l}\sigma = (\iota, \_, c, m)\\ h \models \mathbb{X}_{c,m},\ \iota\end{array}}{\sigma\cdot\mathsf{call}\, e,\ h \longrightarrow \sigma\cdot\mathsf{ret}\, e,\ h}$$

(rLaunchEx)
$$\frac{\begin{array}{l}\sigma = (\iota, \_, c, m)\\ h \not\models \mathbb{X}_{c,m},\ \iota\end{array}}{\sigma\cdot\mathsf{call}\, e,\ h \longrightarrow \sigma\cdot\mathsf{fatalExc},\ h}$$

(rFrame)
$$\frac{\begin{array}{l}\sigma = (\iota, \_, c, m)\\ h \models \mathbb{X}_{c,m},\ \iota\end{array}}{\sigma\cdot\mathsf{ret}\, v,\ h \longrightarrow v,\ h}$$

(rFrameEx)
$$\frac{\begin{array}{l}\sigma = (\iota, \_, c, m)\\ h \not\models \mathbb{X}_{c,m},\ \iota\end{array}}{\sigma\cdot\mathsf{ret}\, v,\ h \longrightarrow \mathsf{fatalExc},\ h}$$

(rPrf)
$$\frac{\sigma = (\iota, \_, \_, \_) \quad h \models \mathbb{p},\ \iota}{\sigma\cdot v\ \mathsf{prv}\ \mathbb{p},\ h \longrightarrow \sigma\cdot v,\ h}$$

(rPrfEx)
$$\frac{\sigma = (\iota, \_, \_, \_) \quad h \not\models \mathbb{p},\ \iota}{\sigma\cdot v\ \mathsf{prv}\ \mathbb{p},\ h \longrightarrow \sigma\cdot\mathsf{verfExc},\ h}$$

Figure 5. Reduction rules of operational semantics.

Each program also comes with typing judgements $\Gamma \vdash e : t$ and $h \vdash e_r : t$ for source and runtime expressions, respectively. An environment $\Gamma \in$ ENV is a tuple of the class containing the current method, the method identifier, and the type of the sole argument.

Finally, the judgement $h \models \iota, c$ expresses that in heap h, the object at address ι satisfies the invariant declared in class c. We define that the judgement trivially holds if the object is not allocated ($\iota \notin \mathrm{dom}(h)$) or is not an instance of c ($\mathrm{cls}(h, \iota) \not\prec: c$). We say that the property $\mathbb{p}$ is *valid* in heap h wrt. address ι if all invariants in $[\![\mathbb{p}]\!]_{h,\iota}$ are satisfied. We denote validity of properties by $h \models \mathbb{p},\ \iota$:

$$h \models \mathbb{p},\ \iota \;\Leftrightarrow\; \forall(\iota', c) \in [\![\mathbb{p}]\!]_{h,\iota}.\ h \models \iota', c$$

Operational Semantics. The framework parameter $\mathbb{X}$ describes which invariants are expected to hold at visible states. Given a program P and a set of properties $\mathbb{X}_{c,m}$, each characterising the property that needs to hold at the beginning and end of a method m of class c, the *runtime semantics* is the relation $\longrightarrow \subseteq$ (REXPR $\times$ HP) $\times$ (REXPR $\times$ HP) defined in Fig. 5.

The first eight rules are standard for object-oriented languages. Note that in rNew, a new object is created using the function new, which takes a type as parameter rather than a class, thereby making the semantics parametric wrt. the type system: different type systems may use different regions and definitions of new to describe heap-topological information. Similarly, upd and fld do not fix a particular heap representation. Rule rCall describes method calls; it stores the class in which the method body is defined in the new stack frame σ, and introduces the "marker" call e_r at the beginning of the method body.

Our reduction rules abstract away from program verification and describe only its effect. Thus, rLaunch, rLaunchExc, rFrame, and rFrameExc check whether $\mathbb{X}_{c,m}$ is valid at the beginning and end of any execution of a method m defined in class c, and generate a fatal exception, fatalExc, if the check fails. This represents the visible state semantics discussed in the introduction. Proof obligations e prv $\mathbb{p}$ are verified once e reduces to a value (rPrf and rPrfExc); if $\mathbb{p}$ is not found to be valid, a verification exception verfExc is generated.

Verification using visible state semantics amounts to showing all proof obligations in some program logic, based on the assumption that expected invariants hold in visible states. Informally then, a specific verification technique described in our framework is sound if it guarantees that a fatalExc is never encountered. Verification technique soundness does allow verfExc to be generated, but this will never happen in a correctly verified program. We give a formal definition of soundness at the end of the next section.

This semantics allows us to be parametric wrt. the syntax of invariants and the logic of proofs. We also define properties that permit us to be parametric wrt. a sound type system (cf. Def. 32 in App. **??**). Thus, we can concentrate entirely on verification concerns.

4. Verification Techniques

In this section, we formalise verification techniques and their connection to programs. Moreover, we define what it means for a verification technique to be sound.

A verification technique is essentially a 7-tuple, where the *components* of the tuple provide instantiations for the seven parameters of our framework. These instantiations are expressed in terms of the regions and properties provided by the programming language. To allow the instantiations to refer to the program, for instance, to look up field declarations, we define a verification technique as a mapping from programs to 7-tuples.

Definition 1 (Verification Technique)**.** *A* verification technique $\mathcal{V}$ *for a* programming language *is a mapping from programs into a tuple:*

$$\begin{array}{l} \mathcal{V}\ :\ \mathrm{PRG} \to \mathrm{EXP} \times \mathrm{VUL} \times \mathrm{DEP} \times \mathrm{PRE} \times \mathrm{END} \times \mathrm{CLL} \times \mathrm{ASS}\\ \textit{where}\\ \begin{array}{rcl} \mathrm{EXP} & = & \mathrm{CLS} \times \mathrm{MTHD} \to \mathbf{P}\\ \mathrm{VUL} & = & \mathrm{CLS} \times \mathrm{MTHD} \to \mathbf{P}\\ \mathrm{DEP} & = & \mathrm{CLS} \to \mathbf{P}\\ \mathrm{PRE} & = & \mathrm{CLS} \times \mathrm{MTHD} \times \mathbf{R} \to \mathbf{P}\\ \mathrm{END} & = & \mathrm{CLS} \times \mathrm{MTHD} \to \mathbf{P}\\ \mathrm{CLL} & = & \mathrm{CLS} \times \mathrm{MTHD} \times \mathrm{CLS} \to \mathbf{R}\\ \mathrm{ASS} & = & \mathrm{CLS} \times \mathrm{MTHD} \times \mathrm{CLS} \times \mathrm{MTHD} \to \mathbf{R} \end{array} \end{array}$$

To describe a verification technique applied to a program, we write the application of the components to class, method names, etc., as $\mathbb{X}_{c,m}$, $\mathbb{V}_{c,m}$, $\mathbb{D}_c$, $\mathbb{B}_{c,m,\mathbb{r}}$, $\mathbb{E}_{c,m}$, $\mathbb{U}_{c,m,c'}$, $\mathbb{C}_{c,m,c',m'}$. The meaning of these components is:

$\mathbb{X}_{c,m}$**:** the property expected to be valid at the beginning and end of the body of method m in class c. The parameters c and m allow a verification technique to expect different invariants in the visible states of different methods. For instance, JML's helper methods [16, 17] do not expect any invariants to hold.

$\mathbb{V}_{c,m}$**:** the property vulnerable to method m of class c, that is, the property whose validity may be broken while control is inside m. Method m can break an invariant by updating a field or by calling a method that breaks, but does not re-establish the invariant (for instance, a helper method). The parameters c and m allow a verification technique to require that invariants of certain classes (for instance, c's subclasses) are not vulnerable.

$\mathbb{D}_c$**:** the property that depends on a field declared in class c. The parameter c is used, for instance, to prevent invariants from depending on fields declared in c's superclasses [16, 28].

$\mathbb{B}_{c,m,\mathbb{r}}$**:** the property whose validity has to be proven before calling a method on a receiver in region $\mathbb{r}$ from the execution of a method m in class c. The parameters allow a verification

$$\text{(vs-null)}\ \frac{}{\Gamma \vdash_{\mathcal{V}} \mathsf{null}} \qquad \text{(vs-Var)}\ \frac{}{\Gamma \vdash_{\mathcal{V}} x} \qquad \text{(vs-this)}\ \frac{}{\Gamma \vdash_{\mathcal{V}} \mathsf{this}} \qquad \text{(vs-new)}\ \frac{}{\Gamma \vdash_{\mathcal{V}} \mathsf{new}\, t} \qquad \text{(vs-fld)}\ \frac{\Gamma \vdash_{\mathcal{V}} e}{\Gamma \vdash_{\mathcal{V}} e.f}$$

$$\text{(vs-ass)}\ \frac{\Gamma \vdash e : \mathbb{r}\, c' \quad \mathcal{F}(c', f) = \_, c \quad \mathbb{r} \sqsubseteq \mathbb{U}_{\Gamma,c} \quad \Gamma \vdash_{\mathcal{V}} e \quad \Gamma \vdash_{\mathcal{V}} e'}{\Gamma \vdash_{\mathcal{V}} e.f := e'}$$

$$\text{(vs-call)}\ \frac{\Gamma \vdash e : \mathbb{r}\, c' \quad \mathcal{B}(c', m) = \_, c \quad \mathbb{r} \sqsubseteq \mathbb{C}_{\Gamma,c,m} \quad \Gamma \vdash_{\mathcal{V}} e \quad \Gamma \vdash_{\mathcal{V}} e'}{\Gamma \vdash_{\mathcal{V}} e.m(e'\ \mathsf{prv}\ \mathbb{B}_{\Gamma,\mathbb{r}})}$$

$$\text{(vs-class)}\ \frac{\left.\begin{array}{c}\mathcal{B}(c,m) = e, c \\ \mathcal{M}(c,m) = t, t'\end{array}\right\} \Rightarrow \begin{cases} e = e'\ \mathsf{prv}\ \mathbb{E}_{c,m} \\ c, m, t \vdash_{\mathcal{V}} e' \end{cases}}{\vdash_{\mathcal{V}} c}$$

Figure 6. Well-Verified source expressions and classes.

technique to impose proof obligations depending on the calling method and the ownership relation between caller and callee.

$\mathbb{E}_{c,m}$**:** the property whose validity has to be proven at the end of method m in class c. The parameters allow a verification technique to require different proofs for different methods to exclude subclass invariants or helper methods.

$\mathbb{U}_{c,m,c'}$**:** the region of allowed receivers for an update of a field in class c', within the body of method m in class c. The parameters allow a verification technique, for instance, to prevent field updates within pure methods.

$\mathbb{C}_{c,m,c',m'}$**:** the region of allowed receivers for a call to method m' of class c', within the body of method m of class c. The parameters allow a verification technique to permit calls depending on attributes (such as purity or effect specifications) of the caller and the callee.

Role of the Seven Components. The operational semantics uses a verification technique to specify the invariants expected in visible states, whereas the static analysis imposed by the verification technique describes program restrictions and proof obligations. More precisely, the operational semantics uses $\mathbb{X}$, to be checked at visible states; soundness requires that $\mathbb{X} \setminus \mathbb{V}$ holds during a method activation. Static analysis describes proof obligations using $\mathbb{B}$ and $\mathbb{E}$, and ensures program restrictions, through $\mathbb{U}$ and $\mathbb{C}$, are respected. Finally, $\mathbb{D}$ restricts invariants for well-verified programs (cf. Def. 2). Sec. 5 gives five conditions on these components that guarantee soundness.

It might be initially surprising that we need as many as seven components. This number is justified by the variety of concepts used by modern verification techniques, such as accessibility of fields, purity, helper methods, ownership, and effect specifications. Note for instance that $\mathbb{V}$ would be redundant if all methods were to re-establish the invariants they break; in such a setting, a method could break invariants only through field updates, and $\mathbb{V}$ could be derived from $\mathbb{U}$ and $\mathbb{D}$. However, in the presence of helper methods and ownership, methods may break but not re-establish invariants.

The class and method identifiers used as parameters to our components can be extracted from an environment Γ or a stack frame σ in the obvious way. Thus, for $\Gamma = (c, m, \_)$ or for $\sigma = (\iota, \_, c, m)$, we use $\mathbb{X}_\Gamma$ and $\mathbb{X}_\sigma$ as shorthands for $\mathbb{X}_{c,m}$; we also use $\mathbb{B}_{\Gamma,\mathbb{r}}$ and $\mathbb{B}_{\sigma,\mathbb{r}}$ as shorthands for $\mathbb{B}_{c,m,\mathbb{r}}$.

Well-Verified Programs. The judgement $\Gamma \vdash_{\mathcal{V}} e$ expresses that expression e is well-verified according to verification technique $\mathcal{V}$. The rules for this *well-verification judgement* are shown in Fig. 6.

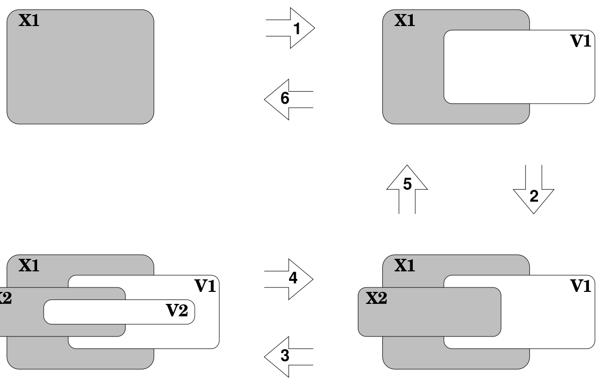


Figure 7. Open calls and valid invariants in a heap

The first five rules express that literals, variable lookup, object creation, and field lookup do not require proofs. The receiver of a field update must fall into $\mathbb{U}$ (vs-ass). The receiver of a call must fall into $\mathbb{C}$ (vs-call). Moreover, we require the proof of $\mathbb{B}$ before a call. Finally, a class is well-verified if the body of each of its methods is well-verified and ends with a proof obligation for $\mathbb{E}$ (vs-class). Note that we use the type judgement $\Gamma \vdash e : t$ without defining it; the definition is given by the underlying programming language, not by our framework.

Fig. 16 in App. **??** defines the judgement $h \vdash_{\mathcal{V}} \mathrm{e}_r$ for verified runtime expressions. The rules correspond to those from Fig. 6, with the addition of rules for values and nested calls.

A program P is well-verified wrt. $\mathcal{V}$, denoted as $\vdash_{\mathcal{V}}$ P, iff (1) all classes are well-verified and (2) all class invariants respect the dependency restrictions dictated by $\mathbb{D}$. That is, the invariant of an object ι' declared in a class c' will be preserved by an update of a field of an object of class c if it is not within $\mathbb{D}_c$.

Definition 2 (Well-Verified Programs)**.**

$$\vdash_{\mathcal{V}} \mathrm{P} \ \Leftrightarrow$$

$$\text{(W1)} \qquad \forall c \in \mathrm{P}.\ \vdash_{\mathcal{V}} c$$

$$\text{(W2)} \qquad \left.\begin{array}{l} \mathcal{F}(\mathrm{cls}(h,\iota), f) = \_, c \\ (\iota', c') \notin [\![\mathbb{D}_c]\!]_{h,\iota}, \\ h \models \iota', c' \end{array}\right\} \Rightarrow\ \mathrm{upd}(h, \iota, f, v) \models \iota', c'$$

Valid States. The properties $\mathbb{X}$ and $\mathbb{X} \setminus \mathbb{V}$ characterise the invariants that are known to hold in the visible states and between visible states of the current method execution, respectively. That is, they reflect the local knowledge of the current method, but do not describe globally all the invariants that need to hold in a given state.

For any state with heap h and execution stack $\overline{\sigma}$, the function $\mathrm{vi}(\overline{\sigma}, h)$ yields the set of *valid invariants*, that is, invariants that are expected to hold :

$$\mathrm{vi}(\overline{\sigma}, h) = \begin{cases} \emptyset & \text{if } \overline{\sigma} = \epsilon \\ (\mathrm{vi}(\overline{\sigma_1}, h) \cup [\![\mathbb{X}_\sigma]\!]_{h,\sigma}) \setminus [\![\mathbb{V}_\sigma]\!]_{h,\sigma} & \text{if } \overline{\sigma} = \overline{\sigma_1} \cdot \sigma \end{cases}$$

The call stack is empty at the beginning of program execution, at which point we expect the heap to be empty. For each additional stack frame σ, the corresponding method m may assume $\mathbb{X}_\sigma$ at the beginning of the call, therefore we add $[\![\mathbb{X}_\sigma]\!]_{h,\sigma}$ to the valid invariants. The method may break $\mathbb{V}_\sigma$ during the call, and so we remove $[\![\mathbb{V}_\sigma]\!]_{h,\sigma}$ from the valid invariants.

Fig. 7 depicts this mechanism of invariant violation and reestablishing for the execution of two consecutive calls. The properties $\mathbb{X}_1, \mathbb{V}_1$ denote the expected and vulnerable properties of the outer call and $\mathbb{X}_2, \mathbb{V}_2$ are the expected and vulnerable properties of the subcall. The first call violates $\mathbb{V}_1$ but $\mathbb{X}_1 \setminus \mathbb{V}_1$ hold throughout the call (1). Before making the subcall, it establishes all of $\mathbb{X}_2$ (2). The subcall violates $\mathbb{V}_2$ (3) but reestablishes all of $\mathbb{X}_2$ before returning (4); similarly, after the first call resumes control (5), it re-establishes $\mathbb{X}_1$ at the end of its execution (6).

A state with heap h and stack $\overline{\sigma}$ is *valid* iff:

(1) $\overline{\sigma}$ is a valid stack, denoted by $h \vdash_{\mathcal{V}} \overline{\sigma}$ (Def. **??** in App. **??**), and meaning that the receivers of consecutive method calls are within the respective $\mathbb{C}$ regions.

(2) The valid invariants $\text{vi}(\overline{\sigma}, h)$ hold.

(3) If execution is in a visible state with σ as the topmost frame of $\overline{\sigma}$, then the expected invariants $\mathbb{X}_\sigma$ hold additionally.

These properties are formalised in Def. 3. A state is determined by a heap h and a runtime expression e_r; the stack is extracted from e_r using function stack, given by Def. **??** in App. **??**.

Definition 3. *A state with heap h and runtime expression e_r is valid for a verification technique $\mathcal{V}$, $\text{e}_r \models_{\mathcal{V}} h$, iff:*

(1) $h \vdash_{\mathcal{V}} \text{stack}(\text{e}_r)$ (2) $h \models \text{vi}(\text{stack}(\text{e}_r), h)$

(3) $\text{e}_r = F[\sigma \cdot \mathit{call}\, e]$ *or* $\text{e}_r = F[\sigma \cdot \mathit{ret}\, v] \;\Rightarrow\; h \models \mathbb{X}_\sigma,\ \sigma$

Soundness. A verification technique is *sound* if verified programs only produce valid states and do not throw fatal exceptions. More precisely, a verification technique $\mathcal{V}$ is sound for a programming language PL iff for all well-formed and verified programs $\text{P} \in \text{PL}$, any well-typed and verified runtime expression e_r executed in a valid state reduces to another verified expression e'_r with a resulting valid state. Note that a verified e'_r contains no $\mathsf{fatalExc}$ (see Fig. 16).

Well-formedness of program P is denoted by $\vdash_{\mathbf{wf}}$ P (Def. 31, App. **??**). Well-typedness of runtime expression e_r is denoted by $h \vdash \text{e}_r : t$ and required as part of a sound type system in Def. 30, App. **??**. These requirement permits separation of concerns, whereby we can formally define verification technique soundness *in isolation*, assuming program well-formedness and a sound type system.

Definition 4. *A* verification technique $\mathcal{V}$ for a programming language *is a mapping from programs into a tuple:*

$$\mathcal{V} : \text{PRG} \rightarrow \text{EXP} \times \text{VUL} \times \text{DEP} \times \text{PRE} \times \text{END} \times \text{UPD} \times \text{CLL}$$

where

$$\begin{array}{lllll}
\mathbb{X} & \in & \text{EXP} & = & \text{CLS} \times \text{MTHD} \rightarrow \mathbf{P} \\
\mathbb{V} & \in & \text{VUL} & = & \text{CLS} \times \text{MTHD} \rightarrow \mathbf{P} \\
\mathbb{D} & \in & \text{DEP} & = & \text{CLS} \rightarrow \mathbf{P} \\
\mathbb{B} & \in & \text{PRE} & = & \text{CLS} \times \text{MTHD} \times \mathbf{R} \rightarrow \mathbf{P} \\
\mathbb{E} & \in & \text{END} & = & \text{CLS} \times \text{MTHD} \rightarrow \mathbf{P} \\
\mathbb{C} & \in & \text{CLL} & = & \text{CLS} \times \text{MTHD} \times \text{CLS} \rightarrow \mathbf{R} \\
\mathbb{U} & \in & \text{UPD} & = & \text{CLS} \times \text{MTHD} \times \text{CLS} \times \text{MTHD} \rightarrow \mathbf{R}
\end{array}$$

5. Well-Structured Verification Techniques

In this section, we identify conditions on the components of a verification technique that are sufficient for soundness.

Definition 5 (Well-Structured Verification Methodology)**.** *A verification technique is* well-structured *if, for all programs in the programming language:*

$$\begin{array}{ll}
\text{(S1)} & \mathbb{r} \sqsubseteq \mathbb{C}_{c,m,c'm'} \Rightarrow (\mathbb{r} \triangleright \mathbb{X}_{c',m'}) \setminus (\mathbb{X}_{c,m} \setminus \mathbb{V}_{c,m}) \;\subseteq\; \mathbb{B}_{c,m,\mathbb{r}} \\
\text{(S2)} & \mathbb{V}_{c,m} \cap \mathbb{X}_{c,m} \subseteq \mathbb{E}_{c,m} \\
\text{(S3)} & \mathbb{C}_{c,m,c',m'} \triangleright (\mathbb{V}_{c',m'} \setminus \mathbb{E}_{c',m'}) \;\subseteq\; \mathbb{V}_{c,m} \\
\text{(S4)} & \mathbb{U}_{c,m,c'} \triangleright \mathbb{D}_{c'} \;\subseteq\; \mathbb{V}_{c,m} \\
\text{(S5)} & c' <: c \;\Rightarrow \begin{cases} \mathbb{X}_{c',m} \subseteq \mathbb{X}_{c,m}, \\ \mathbb{V}_{c',m} \setminus \mathbb{E}_{c',m} \subseteq \mathbb{V}_{c,m} \setminus \mathbb{E}_{c,m} \end{cases}
\end{array}$$

In the above, the set theoretic symbols have the obvious interpretation in the domain of properties. For example (S2) is short for $\forall h, \iota \;:\; [\![\mathbb{V}_{c,m}]\!]_{h,\iota} \cap ([\![\mathbb{X}_c]\!]_{h,\iota} \subseteq [\![\mathbb{E}_{c,m}]\!]_{h,\iota}$. We use *viewpoint adaptation* $\mathbb{r} \triangleright \mathbb{p}$, defined as:

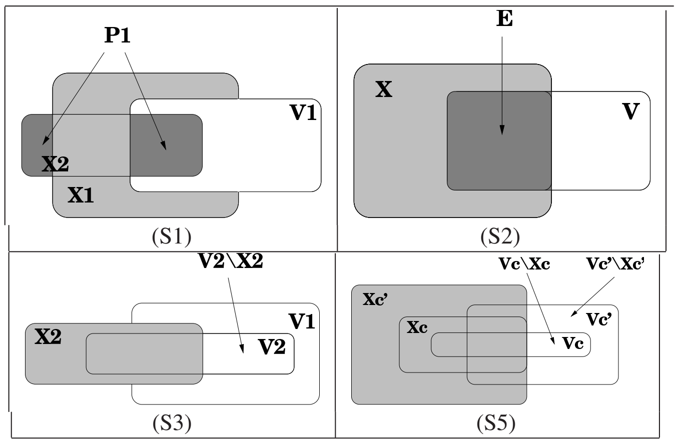


Figure 8. Well Structured Conditions

$$[\![\mathbb{r} \triangleright \mathbb{p}]\!]_{h,\iota} = \bigcup_{\iota' \in [\![\mathbb{r}]\!]_{h,\iota}} [\![\mathbb{p}]\!]_{h,\iota'}$$

meaning that the interpretation of a viewpoint-adapted property $\mathbb{r} \triangleright \mathbb{p}$ wrt. an address ι is equal to the union of the interpretations of $\mathbb{p}$ wrt. each object in the interpretation of $\mathbb{r}$.

The first two conditions relate proof obligations with expected invariants. *(S1)* ensures for a call within the permitted region that the expected invariants of the callee ($\mathbb{r} \triangleright \mathbb{X}_{c',m'}$) minus the invariants that hold throughout the calling method ($\mathbb{X}_{c,m} \setminus \mathbb{V}_{c,m}$) are included in the proof obligation for the call ($\mathbb{B}_{c,m,\mathbb{r}}$). (S2) ensures that the invariants that were broken during the execution of a method, but which are required to hold again at the end of the method ($\mathbb{V}_{c,m} \cap \mathbb{X}_{c,m}$) are included in the proof obligation at the end of the method ($\mathbb{E}_{c,m}$).

The third and fourth condition ensure that invariants that are broken by a method m of class c are actually in its vulnerable set. Condition (S3) deals with calls and therefore uses viewpoint adaptation for call regions ($\mathbb{C}_{c,m,c',m'} \triangleright \ldots$). It restricts the invariants that may be broken by the callee method m', but are not re-established by the callee through $\mathbb{E}$. These invariants must be included in the vulnerable invariants of the caller. Condition (S4) ensures for field updates within the permitted region that the invariants broken by updating a field of class c' are included in the vulnerable invariants of the enclosing method, m.

Finally, (S5) establishes conditions for subclasses. An overriding method m in a subclass c may expect fewer invariants than the overridden m in superclass c'. Moreover, the subclass method must leave less invariants broken than the superclass method.

To further motivate the well-structured requirements of Def. 5, let us refer back to Fig. 7. (S1) ensures that by proving $\mathbb{B}$ before making the subcall, we can safely make move (2) and reach a valid visible state at the beginning of the subcall. (S2) ensures that by proving $\mathbb{E}$ at the end of both method bodies we can make moves (4) and (6) and reach a valid visible states at the end of both calls. Constraint (S3) ensures that when the caller resumes control after the subcall, we can make move (5) and reach a state where $\mathbb{X}1 \setminus \mathbb{V}1$ holds in the heap. *(S4)* guarantees that $\mathbb{X} \setminus \mathbb{V}$ always hold for any call by ensuring that $\mathbb{V}$ is an adequate upper limit for the effect of a call. Finally, but crucially, (S5) permits static analysis of the relationship between $\mathbb{X}$ and $\mathbb{V}$ in the presence of dynamic dispatch of subclass methods because static property information of what is expected to hold at the visible states and what are the residue vulnerable invariants of the superclass imply the corresponding properties for the same method in the subclass.

Note that the five soundness conditions presented here are slightly weaker than those in the previous version of this work [7]. [3]

The five conditions from Def. 5 guarantee soundness, as stated in Def. **??**.

Theorem 6 (Soundness For Visible-State Verification Techniques)**.** *A well-structured verification technique built on top of a PL with a sound type system is sound.*

This theorem is one of our main results. It reduces the complex task of proving soundness of a verification technique to checking five fairly simple conditions.

5.1 Proof of Soundness Theorem

The proof of Theorem 6 uses a number of lemmas we briefly discuss here. For a start, we require to show the correspondence between well-verified source expressions (Fig. 6) and well-verified runtime expressions (Fig. 16).

Lemma 7 (Substitution/Instantiation)**.**

$$\Gamma \vdash_{\mathcal{V}} e,\;\; \Gamma \vdash h, \sigma \;\Rightarrow\; h \vdash_{\mathcal{V}} \sigma \cdot e$$

Proof. By induction on the derivation of $\Gamma \vdash_{\mathcal{V}} e$. ☐

We also require the following lemma stating that the adaptation operation is adequate and monotonic.

Lemma 8 (Adaptation Correspondence)**.**

1. $h \vdash \sigma \cdot \iota : \mathbb{r}, \_ \;\Rightarrow\; [\![\mathbb{p}]\!]_{h,\iota} \subseteq [\![\mathbb{r} \triangleright \mathbb{p}]\!]_{h,\sigma}$
2. $\mathbb{r}_1 \sqsubseteq \mathbb{r}_2 \;\Rightarrow\; \mathbb{r}_1 \triangleright \mathbb{p} \subseteq \mathbb{r}_2 \triangleright \mathbb{p}$
3. $\mathbb{p}_1 \sqsubseteq \mathbb{p}_2 \;\Rightarrow\; \mathbb{r} \triangleright \mathbb{p}_1 \subseteq \mathbb{r} \triangleright \mathbb{p}_2$

Proof. The first clause is straightforward from (T6) of Def. 32 and (P5) of Def. 30. The second and third clauses are also immediate as a result of (P5) of Def. 30. ☐

We also require a number of lemmas dealing with the reduction rules and heap validity. For instance, the following lemma states that heaps can only grow as a result of a reduction.

Lemma 9. $\mathrm{e}_r,\, h \longrightarrow \mathrm{e}'_r,\, h' \;\Rightarrow\; h \preceq h'$

Proof. By induction on the derivation of $\mathrm{e}_r,\, h \longrightarrow \mathrm{e}'_r,\, h'$ and Definition 27. ☐

The following lemma states that a well-verified runtime expression remains well-verified in an extended heap and also that well-verified values are independent of any guarding stack frame. The latter property is useful when we consider a return from a subcall in the main proof.

Lemma 10.

1. $h \vdash_{\mathcal{V}} \mathrm{e}_r,\; h \preceq h' \;\Rightarrow\; h' \vdash_{\mathcal{V}} \mathrm{e}_r$
2. $h \vdash_{\mathcal{V}} \sigma \cdot v \;\Rightarrow\; h \vdash_{\mathcal{V}} \sigma' \cdot v$

Heap validity depends solely on the stack frames of a runtime expression, thus evaluation contexts are non-influential.

Lemma 11 (Valid States)**.** *If* $\mathrm{stack}(\mathrm{e}_r) = \sigma_1 \cdot \ldots \cdot \sigma_n$ *then*

1. $\sigma' \cdot \mathrm{e}_r \models_{\mathcal{V}} h \;\Leftrightarrow\; \sigma' \cdot E[\mathrm{e}_r] \models_{\mathcal{V}} h$

2. $\sigma' \cdot \mathrm{e}_r \models_{\mathcal{V}} h \;\Leftrightarrow\; \begin{cases} h \models \mathbb{X}_{\sigma'} \setminus \mathbb{V}_{\sigma'} \setminus \mathbb{V}_{\sigma_1} \setminus \ldots \setminus \mathbb{V}_{\sigma_n} \\ h \vdash_{\mathcal{V}} \sigma' \cdot \sigma_1 \cdot \ldots \cdot \sigma_n \\ \mathrm{e}_r \models_{\mathcal{V}} h \end{cases}$

Proof. Immediate from Definition 3. ☐

In order to determine the effect of updates on valid invariants in a heap we require the following lemma.

Lemma 12 (Invariant Satisfaction Effect)**.**

$$\left.\begin{array}{l} \vdash_{\mathcal{V}} \mathrm{P} \\ h \models [\![\mathbb{p}]\!]_{h,\iota'} \\ \mathrm{cls}(h,\iota) <: c' \\ \mathcal{F}(c', f, =)_{\_}, c \\ h' = \mathrm{upd}(h, \iota, f, v) \end{array}\right\} \Rightarrow\; h' \models [\![\mathbb{p}]\!]_{h',\iota'} \setminus [\![\mathbb{D}_c]\!]_{h',\iota}$$

Proof. Immediate from (W2) of Def. 2 ☐

Thus for a well structured verification technique, we are guaranteed that certain invariants are unaffected by reductions.

Lemma 13 (Computation effects)**.** *For arbitrary invariant set* $s = \{(\iota_1, c_1), \ldots, (\iota_n, c_n)\}$*, if* $\mathcal{V}$ *is well-structured then:*

$$\left.\begin{array}{l} h \models s \\ \mathrm{e}_r,\, h \longrightarrow \mathrm{e}'_r,\, h' \\ \mathrm{stack}(\mathrm{e}'_r) = \sigma_1 \cdot \ldots \cdot \sigma_n \end{array}\right\} \Rightarrow\; h' \models s \setminus \mathbb{V}_{\sigma_1} \setminus \ldots \setminus \mathbb{V}_{\sigma_n}$$

Proof. By induction on the derivation of $\mathrm{e}_r,\, h \longrightarrow \mathrm{e}'_r,\, h'$, Lemma 12 and $(S4)$ of Definition 5. ☐

Finally, we restate the soundness theorem, Theorem 6, in full and prove the main cases.

Theorem 9 Soundness for Visible-State Verification Techniques *If* $\mathcal{V}$ *is well-structured, then:*

$$\left.\begin{array}{l} \vdash_{\mathit{wf}} \mathrm{P},\;\; h \vdash \mathrm{e}_r : t, \\ \vdash_{\mathcal{V}} \mathrm{P},\;\; \mathrm{e}_r \models_{\mathcal{V}} h,\;\; h \vdash_{\mathcal{V}} \mathrm{e}_r, \\ \mathrm{e}_r,\, h \longrightarrow \mathrm{e}'_r,\, h' \end{array}\right\} \;\Rightarrow\; \mathrm{e}'_r \models_{\mathcal{V}} h',\;\; h' \vdash_{\mathcal{V}} \mathrm{e}'_r$$

Proof. The proof is by induction on the derivation of $\mathrm{e}_r,\, h \longrightarrow \mathrm{e}'_r,\, h'$. As a shorthand, we find it convenient to write $h \models \mathbb{X}_\sigma$ and $h \models \mathbb{r} \triangleright \mathbb{X}_\sigma$ instead of $h \models [\![\mathbb{X}_\sigma]\!]_{h,\sigma}$ and $h \models [\![\mathbb{r} \triangleright \mathbb{X}_\sigma]\!]_{h,\sigma}$ respectively, and similarly for the framework component $\mathbb{V}_\sigma$. For convenience we also enumerate the premises of the Theorem as

$$\vdash_{\mathbf{wf}} \mathrm{P}, \tag{1}$$
$$h \vdash \mathrm{e}_r : t, \tag{2}$$
$$\vdash_{\mathcal{V}} \mathrm{P}, \tag{3}$$
$$\mathrm{e}_r \models_{\mathcal{V}} h, \tag{4}$$
$$h \vdash_{\mathcal{V}} \mathrm{e}_r, \tag{5}$$
$$\mathrm{e}_r,\, h \longrightarrow \mathrm{e}'_r,\, h' \tag{6}$$

We here focus on the main cases for the derivation of (6) and leave the remaining simpler cases for the interested reader.

rAss: From the conclusion and the premises of the rule we know

$$\mathrm{e}_r = \sigma \cdot \iota.f := v \tag{7}$$
$$\mathrm{e}'_r = \sigma \cdot v \tag{8}$$
$$h' = \mathrm{upd}(h, \iota, f, v) \tag{9}$$

[3] Namely, (S3) and (S5) are weaker, and thus less restrictive, here. In [7], instead of (S3) we required the stronger version $\mathbb{C}_{c,m,c',m'} \triangleright (\mathbb{V}_{c',m'} \setminus \mathbb{X}_{c',m'}) \subseteq \mathbb{V}_{c,m}$, and a similarly stronger version for (S5). However, the two versions are equivalent when $\mathbb{E}_{c,m}$ is the minimal set allowed by (S2), i.e., when $\mathbb{E}_{c,m} = \mathbb{V}_{c,m} \cap \mathbb{X}_{c,m}$ for all c and m. In all techniques presented here, $\mathbb{E}_{c,m}$ is minimal in the above sense.

From (7) we know (5) could only have been derived using vd-ass and from the premises of this rule we know

$$h \vdash \sigma\cdot\iota : \mathbb{r}\, c' \tag{10}$$
$$\mathcal{F}(c', f) = \_,\, c \tag{11}$$
$$\mathbb{r} \sqsubseteq \mathbb{U}_{\sigma,c} \tag{12}$$
$$h \vdash_{\mathcal{V}} \sigma\cdot\iota \tag{13}$$
$$h \vdash_{\mathcal{V}} \sigma\cdot v \tag{14}$$

From (9) and Def. 27 (H4) we know

$$h \simeq h' \text{ which implies } h \preceq h' \tag{15}$$

Thus by (14), (15) and Lemma 11.1 and then by (8) we derive that the resultant configuration is still well-verified, i.e.,

$$h' \vdash_{\mathcal{V}} \mathrm{e}'_r$$

We still need to show that (6) reduces to a valid state. From (4) and Def. 3 we know

$$h \models \mathbb{X}_\sigma \setminus \mathbb{V}_\sigma \tag{16}$$

Also, from (10) and Def. 32(T4) we know

$$\mathrm{cls}(h, \iota) <: c' \tag{17}$$

By (3), (16), (17), (11) (9) and Lemma 12 we obtain

$$h' \models \mathbb{X}_\sigma \setminus \mathbb{V}_\sigma \setminus [\![\mathbb{D}_c]\!]_{h,\iota} \tag{18}$$

By (10) and Lemma 8.1 we get

$$[\![\mathbb{D}_c]\!]_{h,\iota} \subseteq [\![\mathbb{r} \triangleright \mathbb{D}_c]\!]_{h,\sigma} \tag{19}$$

By (12) and Lemma 8.2 we get

$$[\![\mathbb{r} \triangleright \mathbb{D}_c]\!]_{h,\sigma} \subseteq [\![\mathbb{U}_{\sigma,c} \triangleright \mathbb{D}_c]\!]_{h,\sigma} \tag{20}$$

Since we assume that our verification technique $\mathcal{V}$ is well-structured, by 5(S4) we also get

$$[\![\mathbb{U}_{\sigma,c} \triangleright \mathbb{D}_c]\!]_{h,\sigma} \subseteq [\![\mathbb{V}_\sigma]\!]_{h,\sigma} \tag{21}$$

Thus, from (18), (19), (20), (21) and set inclusion transitivity we obtain

$$h' \models \mathbb{X}_\sigma \setminus \mathbb{V}_\sigma$$

which by (8) and Def. 3 means we get the valid state

$$\mathrm{e}'_r \models_{\mathcal{V}} h'$$

rCall: From the conclusion and the premises of the rule we know

$$\mathrm{e}_r = \sigma\cdot\iota.m(v) \tag{22}$$
$$\mathrm{e}'_r = \sigma\cdot\sigma'\cdot\mathsf{call}\, e_b \tag{23}$$
$$\mathcal{B}(\mathrm{cls}(h,\iota), m) = e_b,\, c \tag{24}$$
$$\sigma' = (\iota, v, c, m) \tag{25}$$
$$h' = h \tag{26}$$

From (22) we know that (5) could only have been derived using vd-call-2, and thus from the premises of this rule we get

$$h \vdash \sigma\cdot\iota : \mathbb{r}\, c' \tag{27}$$
$$\mathcal{B}(c', m) = \_,\, c'' \tag{28}$$
$$h \models \mathbb{B}_{\sigma,\mathbb{r}},\, \sigma \tag{29}$$
$$\mathbb{r} \sqsubseteq \mathbb{C}_{\sigma,c'',m} \tag{30}$$
$$h \vdash_{\mathcal{V}} \sigma\cdot\iota \tag{31}$$
$$h \vdash_{\mathcal{V}} \sigma\cdot v \tag{32}$$

From (3), Def. 2(W1) and vs-class we know

$$e_b = e\, \mathsf{prv}\, \mathbb{E}_{c,m} \tag{33}$$
$$(c, m, \_) \vdash_{\mathcal{V}} e \tag{34}$$

From (24) and Def. 30(P2) we know

$$\mathrm{cls}(h, \iota) <: c \tag{35}$$

This allows us to deduce that h, σ' is well-formed wrt. the environment $(c, m, \_)$ since by (25), (35) and Def. 31 we get

$$(c, m, \_) \vdash_{\mathsf{wf}} h, \sigma' \tag{36}$$

By (34), (36) and Lemma 7 we derive

$$h \vdash_{\mathcal{V}} \sigma'\cdot e \tag{37}$$

Hence, by (37) and vd-start we get

$$h \vdash_{\mathcal{V}} \sigma\cdot\sigma'\cdot\mathsf{call}\, e\, \mathsf{prv}\, \mathbb{E}_{\sigma'}$$

and by (26), (23), (33) and (25) we obtain

$$h' \vdash_{\mathcal{V}} \mathrm{e}'_r$$

We still need to show that (6) reduces to a valid state, that is $\mathrm{e}'_r \models_{\mathcal{V}} h'$. From (27), Def. 32(T4) we know

$$\mathrm{cls}(h, \iota) <: c' \tag{38}$$

and by (38), (28), (24), (1) and Def. 31(F4) we deduce

$$c <: c'' \tag{39}$$

Since $\mathcal{V}$ is well-structured, then by Def. 5(S5) and (39) we obtain

$$\mathbb{X}_{c,m} \subseteq \mathbb{X}_{c'',m} \tag{40}$$

which, by Lemma 8.2 yields

$$\mathbb{r} \triangleright \mathbb{X}_{c,m} \subseteq \mathbb{r} \triangleright \mathbb{X}_{c'',m} \tag{41}$$

Moreover from Def. 5(S1) and (30) we get

$$(\mathbb{r} \triangleright \mathbb{X}_{c'',m}) \setminus (\mathbb{X}_\sigma \setminus \mathbb{V}_\sigma) \subseteq \mathbb{B}_{\sigma,\mathbb{r}} \tag{42}$$

From (41) and (42) we obtain

$$(\mathbb{r} \triangleright \mathbb{X}_{c,m}) \setminus (\mathbb{X}_\sigma \setminus \mathbb{V}_\sigma) \subseteq \mathbb{B}_{\sigma,\mathbb{r}} \tag{43}$$

From (4) and Def. 3, and then from (29) we know

$$h \models \mathbb{X}_\sigma \setminus \mathbb{V}_\sigma \;\cup\; \mathbb{B}_{\sigma,\mathbb{r}} \tag{44}$$

and from (43) and (44) we obtain

$$h \models \mathbb{X}_\sigma \setminus \mathbb{V}_\sigma \;\cup\; (\mathbb{r} \triangleright \mathbb{X}_{c,m}) \tag{45}$$

From (25), (27) and Lemma 8.1 we know

$$[\![\mathbb{X}_{\sigma'}]\!]_{h,\sigma'} \subseteq [\![\mathbb{r} \triangleright \mathbb{X}_{c,m}]\!]_{h,\sigma} \tag{46}$$

and by (45) and (46) we obtain Def. 3(V2) and (V3), i.e.,

$$h \models \mathbb{X}_\sigma \setminus \mathbb{V}_\sigma \;\cup\; \mathbb{X}_{\sigma'} \tag{47}$$

Also by (25), (27), (30), (39) and Def. **??** we deduce Def. 3(V1), i.e.,

$$h \vdash_{\mathcal{V}} \sigma\cdot\sigma' \tag{48}$$

and by (48), (47), Def. 3, and by (23) and (26) we get, as required,

$$\mathrm{e}'_r \models_{\mathcal{V}} h' \tag{49}$$

rCxtFrame: From the conclusion and the premises of the rule we know

$$\mathrm{e}_r = \sigma\cdot\mathrm{e}^1_r \tag{50}$$
$$\mathrm{e}'_r = \sigma\cdot\mathrm{e}^2_r \tag{51}$$
$$\mathrm{e}^1_r,\, h \longrightarrow \mathrm{e}^2_r,\, h' \tag{52}$$

From (50) and (52)[4] we know that (5) could have been derived using either of the following three subcases:

1. vd-start: From the conclusion and premises of this rule we know

$$\mathrm{e}_r^1 = \sigma'\cdot\mathsf{call}\, e\, \mathsf{prv}\, \mathbb{E}_{\sigma'} \tag{53}$$

$$h \vdash_{\mathcal{V}} \sigma'\cdot e \tag{54}$$

As a result of (53), we know that (52) could have only been derived using either rLaunch or rLaunchEx. Moreover, because of (4), (50), (53), i.e., $\sigma\cdot\sigma'\cdot\mathsf{call}\, e\, \mathsf{prv}\, \mathbb{E}_{\sigma'} \models_{\mathcal{V}} h$, and 3, we also know

$$h \models \mathbb{X}_\sigma \setminus \mathbb{V}_\sigma \cup \mathbb{X}_{\sigma'} \tag{55}$$

$$h \vdash_{\mathcal{V}} \sigma\cdot\sigma' \tag{56}$$

Now (55), in particular $h \models \mathbb{X}_{\sigma'}$, rules out the use of rLaunchEx to derive (52). Thus, if (52) was derived using rLaunch, we know

$$\mathrm{e}_r^2 = \sigma'\cdot\mathsf{ret}\, e\, \mathsf{prv}\, \mathbb{E}_{\sigma'} \quad (\text{where } \mathrm{e}_r' = \sigma\cdot\mathrm{e}_r^2) \tag{57}$$

$$h' = h \tag{58}$$

By (54), (57), (58) and vd-frame we deduce

$$h' \vdash_{\mathcal{V}} \mathrm{e}_r'$$

and by (55), (56), (57), (58) and the fact that $\mathcal{V}$ is well-structured we deduce

$$\mathrm{e}_r' \models_{\mathcal{V}} h'$$

2. vd-frame: From the conclusion and premises of this rule we know

$$\mathrm{e}_r^1 = \sigma'\cdot\mathsf{ret}\, \mathrm{e}_r^3\, \mathsf{prv}\, \mathbb{E}_{\sigma'} \tag{59}$$

$$h \vdash_{\mathcal{V}} \sigma'\cdot\mathrm{e}_r^3 \tag{60}$$

From (59), we know (52) could have been derived using either of the following 3 subcases:

(a) rCxtEval with

$$E[\cdot] = \mathsf{ret}\, [\cdot]\, \mathsf{prv}\, \mathbb{E}_{\sigma'} \tag{61}$$

$$\sigma'\cdot\mathrm{e}_r^3,\, h \longrightarrow \sigma'\cdot\mathrm{e}_r^4,\, h' \tag{62}$$

$$\mathrm{e}_r^2 = \sigma'\cdot E[\mathrm{e}_r^4] \tag{63}$$

From (4), (50), (59), (61) and Lemma 11.1 and then Lemma 11.2 we obtain

$$\sigma'\cdot\mathrm{e}_r^3 \models_{\mathcal{V}} h \tag{64}$$

$$h \vdash_{\mathcal{V}} \sigma\cdot\sigma' \tag{65}$$

$$h \models \mathbb{X}_\sigma \setminus \mathbb{V}_\sigma \setminus \mathbb{V}_{\sigma'} \setminus \mathbb{V}_{\mathsf{stack}(\mathrm{e}_r^3)} \tag{66}$$

Also, from (2), (50), (59), (61) and 32(T6) we know

$$h \vdash \sigma'\cdot\mathrm{e}_r^3 : t \tag{67}$$

Thus by (1), (67), (3), (64), (60), (62) and inductive hypothesis we infer

$$h' \vdash_{\mathcal{V}} \sigma'\cdot\mathrm{e}_r^4 \tag{68}$$

$$\sigma'\cdot\mathrm{e}_r^4 \models_{\mathcal{V}} h' \tag{69}$$

By (68), (61), (63), (51) and vd-frame we derive

$$h' \vdash_{\mathcal{V}} \mathrm{e}_r'$$

By (69), (61), (63) and Lemma 11.1 we deduce

$$\mathrm{e}_r^2 \models_{\mathcal{V}} h' \tag{70}$$

From (66), (62) and Lemma 13 we get

$$h' \models \mathbb{X}_\sigma \setminus \mathbb{V}_\sigma \setminus \mathbb{V}_{\sigma'} \setminus \mathbb{V}_{\mathsf{stack}(\mathrm{e}_r^4)} \tag{71}$$

and by (69), (71), (63), (51) and Lemma 11.2 we obtain, as required,

$$\mathrm{e}_r' \models_{\mathcal{V}} h'$$

(b) rCxtEval, rPrf with

$$E[\cdot] = \mathsf{ret}\, [\cdot] \text{ and } \mathrm{e}_r^3 = v \tag{72}$$

$$\sigma'\cdot v\, \mathsf{prv}\, \mathbb{E}_{\sigma'},\, h \longrightarrow \sigma'\cdot v,\, h \tag{73}$$

$$\mathrm{e}_r^2 = \sigma'\cdot v \text{ and } h' = h \tag{74}$$

$$h \models \mathbb{E}_{\sigma'},\, \sigma' \tag{75}$$

From (72), (74), (51), (60), (75) and vd-end we obtain

$$h' \vdash_{\mathcal{V}} \mathrm{e}_r'$$

Since e_r' is a visible state, Def. 3 requires us to prove that more invariants hold for the resultant state to be valid. From $\mathcal{V}$being well-structured, 5(S2) and (75) we deduce

$$h \models \mathbb{X}_{\sigma'} \cap \mathbb{V}_{\sigma'} \tag{76}$$

and by (74), (72), (50), (51), (59), (4) we know

$$h \models ((\mathbb{X}_\sigma \setminus \mathbb{V}_\sigma) \cup \mathbb{X}_{\sigma'}) \setminus \mathbb{V}_{\sigma'} \tag{77}$$

And thus by (76), (77) and then Def. 3 we deduce $\mathrm{e}_r' \models_{\mathcal{V}} h'$

(c) rCxtEval, rPrfEx with

$$E[\cdot] = \mathsf{ret}\, [\cdot] \text{ and } \mathrm{e}_r^3 = v$$

$$\sigma'\cdot v\, \mathsf{prv}\, \mathbb{E}_{\sigma'},\, h \longrightarrow \sigma'\cdot\mathsf{verfExc},\, h$$

$$\mathrm{e}_r^2 = \sigma'\cdot\mathsf{verfExc} \text{ and } h' = h$$

$$h \not\models \mathbb{E}_{\sigma'},\, \sigma'$$

This case is similar to the previous case and is left for the interested reader.

3. vd-end: from the conclusion and the premises of the rule, we obtain

$$\mathrm{e}_r^1 = \sigma'\cdot\mathsf{ret}\, v \tag{78}$$

$$h \vdash_{\mathcal{V}} \sigma'\cdot v \tag{79}$$

$$h \models \mathbb{E}_{\sigma'},\, \sigma' \tag{80}$$

From (78) we know (52) could only have been derived using either rFrame or rFrameEx. However, from (4), (50) and (78) we know that e_r is in a visible state at σ' and thus

$$h \models \mathbb{X}_{\sigma'} \tag{81}$$

which rules out the possibility of using rFrameEx. Thus by rFrame we know

$$\mathrm{e}_r^2 = v \tag{82}$$

$$h' = h \tag{83}$$

By (51), (82), (83), (79) and Lemma 10.2 we obtain

$$h' \vdash_{\mathcal{V}} \mathrm{e}_r'$$

From (50), (78), (4) and Def. 3, Def. **??** and Def. **??** we know

$$h \models ((\mathbb{X}_\sigma \setminus \mathbb{V}_\sigma) \cup \mathbb{X}_{\sigma'}) \setminus \mathbb{V}_{\sigma'} \,\cup\, \mathbb{X}_{\sigma'} \tag{84}$$

$$\sigma' = (\iota, \_, c', m),\; h \vdash \sigma\cdot\iota : \mathtt{r}\,\_,\; c' <: c,\; \mathtt{r} \sqsubseteq \mathbb{C}_{\sigma,c,m} \tag{85}$$

We can rewrite (84) as

$$h \models (\mathbb{X}_\sigma \setminus \mathbb{V}_\sigma) \setminus (\mathbb{V}_{\sigma'} \setminus \mathbb{X}_{\sigma'}) \cup \mathbb{X}_{\sigma'} \tag{86}$$

[4] The fact that e_r^1 reduces means that e_r^1 contains at least one more stack frame, since all reduction rules in Fig. 5 are defined over runtime expressions of the form $\sigma\cdot\mathrm{e}_r$.

We can combine (86) with (80) to obtain

$$h \models (\mathbb{X}_\sigma \setminus \mathbb{V}_\sigma) \setminus (\mathbb{V}_{\sigma'} \setminus \mathbb{X}_{\sigma'}) \cup \mathbb{E}_{\sigma'} \cup \mathbb{X}_{\sigma'} \qquad (87)$$

By standard properties of set theory, we observe that

$$\begin{array}{rl} & (\mathbb{X}_\sigma \setminus \mathbb{V}_\sigma) \setminus (\mathbb{V}_{\sigma'} \setminus \mathbb{E}_{\sigma'}) \\ = & ((\mathbb{X}_\sigma \setminus \mathbb{V}_\sigma) \setminus (\mathbb{V}_{\sigma'})) \cup ((\mathbb{X}_\sigma \setminus \mathbb{V}_\sigma) \cap (\mathbb{E}_{\sigma'})) \\ \subseteq & ((\mathbb{X}_\sigma \setminus \mathbb{V}_\sigma) \setminus (\mathbb{V}_{\sigma'})) \cup \mathbb{E}_{\sigma'} \\ \subseteq & ((\mathbb{X}_\sigma \setminus \mathbb{V}_\sigma) \setminus (\mathbb{V}_{\sigma'} \setminus \mathbb{X}_{\sigma'})) \cup \mathbb{E}_{\sigma'} \end{array}$$

i.e., we have

$$(\mathbb{X}_\sigma \setminus \mathbb{V}_\sigma) \setminus (\mathbb{V}_{\sigma'} \setminus \mathbb{E}_{\sigma'}) \subseteq (\mathbb{X}_\sigma \setminus \mathbb{V}_\sigma) \setminus (\mathbb{V}_{\sigma'} \setminus \mathbb{X}_{\sigma'}) \cup \mathbb{E}_{\sigma'} \qquad (88)$$

From (87) and (88), we can obtain

$$h \models (\mathbb{X}_\sigma \setminus \mathbb{V}_\sigma) \setminus (\mathbb{V}_{\sigma'} \setminus \mathbb{E}_{\sigma'}) \cup \mathbb{X}_{\sigma'} \qquad (89)$$

By (85) and Lemma 8.1 we have

$$[\![\mathbb{V}_{\sigma'} \setminus \mathbb{E}_{\sigma'}]\!]_{h,\sigma'} \subseteq [\![\mathbb{r} \triangleright (\mathbb{V}_{\sigma'} \setminus \mathbb{E}_{\sigma'})]\!]_{h,\sigma} \qquad (90)$$

Also by (85) and Lemma 8.2 we have

$$\mathbb{r} \triangleright (\mathbb{V}_{\sigma'} \setminus \mathbb{E}_{\sigma'}) \subseteq \mathbb{C}_{\sigma,c,m} \triangleright (\mathbb{V}_{\sigma'} \setminus \mathbb{E}_{\sigma'}) \qquad (91)$$

Since we assume our verification technique $\mathcal{V}$to be well-structured, then by 5(S3) we know

$$\mathbb{C}_{\sigma,c,m} \triangleright (\mathbb{V}_{c,m} \setminus \mathbb{E}_{c,m}) \subseteq \mathbb{V}_\sigma \qquad (92)$$

and by (S5) and (85) we also know

$$(\mathbb{V}_{\sigma'} \setminus \mathbb{E}_{\sigma'}) \subseteq (\mathbb{V}_{c,m} \setminus \mathbb{E}_{c,m}) \qquad (93)$$

and by (93) and Lemma 8.3 we get

$$\mathbb{C}_{\sigma,c,m} \triangleright (\mathbb{V}_{\sigma'} \setminus \mathbb{E}_{\sigma'}) \subseteq \mathbb{C}_{\sigma,c,m} \triangleright (\mathbb{V}_{c,m} \setminus \mathbb{E}_{c,m}) \qquad (94)$$

By (90), (91), (94) and (92) we can rewrite (89) as

$$h \models (\mathbb{X}_\sigma \setminus \mathbb{V}_\sigma) \cup \mathbb{X}_{\sigma'} \qquad (95)$$

and by (95), (83), (82) and (51) we obtain

$$\mathrm{e}'_r \models_{\mathcal{V}} h'$$

as required.

□

6. Instantiations

In this section, we instantiate our framework to describe six verification techniques from the literature, and compare their expressiveness. We also prove their soundness using Def. 5 and Theorem 6 from Sec. 5.

An optimal verification technique would allow maximal expressivity of the invariants (i.e., large $\mathbb{D}$), impose as few program restrictions as possible (i.e., large $\mathbb{U}$ and $\mathbb{C}$), and require as few proof obligations as possible (i.e., small $\mathbb{B}$ and $\mathbb{E}$). Obviously, these are contradictory goals, and some trade-offs need to be struck.

The first three techniques use information about classes to improve the tradeoff, whereas the latter three also use information about the topology of the heap. We call them *unstructured heap* and *structured heap* techniques, respectively.

6.1 Verification Techniques for Unstructured Heaps

Unstructured heap techniques make trade-offs by using information about classes, visibility, and access paths used in definitions of invariants. The instantiations are summarised in Fig. 9 whereby the keyword all denotes the set of all object invariants.

	Poetzsch-Heffter	Huizing & Kuiper	Leavens & Müller
$\mathbb{X}_c$	any	any	any
$\mathbb{V}_{c,m}$	any	vul$\langle c\rangle$	any$\langle c\rangle$
$\mathbb{D}_c$	any	vul$\langle c\rangle$	self$\langle c\rangle$
$\mathbb{B}_{c,m,\mathbb{r}}$	any	vul$\langle c\rangle$	any$\langle c\rangle$
$\mathbb{E}_{c,m,c'}$	any	vul$\langle c\rangle$	any$\langle c\rangle$
$\mathbb{U}_{c,m,c'}$	any	self	any if visF(c', c) emp otherwise
$\mathbb{C}_{c,m,c',m'}$	any	any	any

Figure 9. Verification techniques for unstructured heaps.

6.1.1 Poetzsch-Heffter

Poetzsch-Heffter [32] devised the first verification technique that is sound for call-backs and multi-object invariants. His technique neither restricts programs nor invariants. To deal with this generality, it requires extremely strong proof obligations.

The absence of restrictions is reflected by the regions and properties needed to model Poetzsch-Heffter's technique. We define a singleton region set $\mathbf{R} = \{\mathsf{any}\}$ and a singleton property set $\mathbf{P} = \{\mathsf{any}\}$ with interpretations $[\![\mathsf{any}]\!]_{h,\iota} = \mathrm{dom}(h)$ and $[\![\mathsf{any}]\!]_{h,\iota} = \mathrm{all}$. As shown in Fig. 9, this technique requires all invariants to hold in visible states. It does not restrict invariants; $\mathbb{D}$ allows a field update to affect any invariant. $\mathbb{U}$ and $\mathbb{C}$ permit arbitrary receivers for field updates and method calls. Consequently, any invariant is vulnerable to each method. This requires proof obligations for all invariants before method calls (to handle call-backs) and at the end of the method.

6.1.2 Huizing & Kuiper

Huizing and Kuiper's technique [14] is almost as liberal as Poetzsch-Heffter's, but imposes fewer proof obligations. It achieves this by determining syntactically for each field the set of invariants that are potentially invalidated by updating the field. Proof obligations are imposed only for those vulnerable invariants.

We define the region set $\mathbf{R} = \{\mathsf{self}, \mathsf{any}\}$ with the interpretation $[\![\mathsf{self}]\!]_{h,\iota} = \{\iota\}$ and $[\![\mathsf{any}]\!]_{h,\iota} = \mathrm{dom}(h)$. The region self is used to restrict the receivers of field updates to **this** (see Fig. 9). The concept of vulnerability is captured by the property set $\mathbf{P} = \{\mathsf{vul}\langle\mathbf{c}\rangle, \mathsf{any}\}$ with the following interpretation:

$$\begin{array}{rl} [\![\mathsf{vul}\langle c\rangle]\!]_{h,\iota} = & \{(\iota', c') \mid \text{the invariant of } c' \text{ contains an expression} \\ & \textbf{this}\,.g_1 \ldots g_n.f\ (n \geq 0) \text{ where } \mathcal{F}(c, f) = \_, \_ \wedge \\ & \mathrm{fld}(h, \mathrm{fld}(h, \mathrm{fld}(h, \iota', g_1), \ldots), g_n) = \iota\} \cup \\ & \{(\iota, c') \mid \mathrm{cls}(h, \iota) <: c'\} \\ [\![\mathsf{any}]\!]_{h,\iota} = & \mathrm{all} \end{array}$$

Given an address ι and a class c, the set of vulnerable invariants contains the invariants of all client objects ι' of ι that refer to a field f of c via an access path $g_1 \ldots g_n$, as well as all invariants of ι. The interpretation shows that this technique inspects client invariants syntactically to determine whether they are vulnerable or not.

As shown in Fig. 9, this technique requires all invariants to hold in visible states. It does not restrict invariants; therefore, $\mathbb{D}$ describes exactly the set of vulnerable invariants. These invariants are vulnerable to each method and must be proven before method calls and at the end of each method.

Formalizing Huizing and Kuiper's technique in our framework reveals that it is very similar to Poetzsch-Heffter's. The main difference is that the former technique uses a syntactic analysis and restricts field updates to reduce proof obligations.

6.1.3 Leavens & Müller

Leavens and Müller [16] studied information hiding in interface specifications, based on the notion of visibility defined by access control of the programming language. For instance in Java, private field are visible only within their class. Their technique allows classes to declare several invariants and to specify the visibility of these invariants.

Since our formalization does not cover the visibility of fields and assumes exactly one invariant per class, we model a special case of Leavens and Müller's technique. We assume that all fields of a class have the same visibility. The predicate $\mathsf{visF}(c', c)$ yields whether the fields declared in class c' are visible in class c. We assume that each class declares exactly one invariant and specifies its visibility. The predicate $\mathsf{visI}(c', c)$ yields whether the invariant declared in class c' is visible in class c. A generalization is possible, but does not provide any deeper insights.

We define the region set $\mathbf{R} = \{\mathsf{emp}, \mathsf{any}\}$ with the interpretation $[\![\mathsf{emp}]\!]_{h,\iota} = \emptyset$ and $[\![\mathsf{any}]\!]_{h,\iota} = \mathrm{dom}(h)$. This technique permits field updates on arbitrary receivers as long as the field is visible in the method performing the update (see Fig. 9). Method calls are not restricted

The visibility of invariants is captured by the property set $\mathbf{P} = \{\mathsf{any}, \mathsf{self}\langle\mathbf{c}\rangle, \mathsf{any}\langle\mathbf{c}\rangle\}$ with the following interpretation:

$$[\![\mathsf{any}]\!]_{h,\iota} = \mathsf{all} \qquad [\![\mathsf{any}\langle c\rangle]\!]_{h,\iota} = \{(\iota', c') \mid \mathsf{visI}(c', c)\}$$
$$[\![\mathsf{self}\langle c\rangle]\!]_{h,\iota} = \{(\iota, c) \mid \forall c'.\mathsf{visF}(c, c') \Leftrightarrow \mathsf{visI}(c, c')\}$$

$\mathbb{D}$ allows invariants to depend on fields of the same object declared in the same class, provided that the invariant is visible wherever the field is. This requirement enforces that any method that potentially breaks an invariant can see it and, thus, re-establish it. This requirement is very restrictive, as it disallows multi-object invariants and prevents invariants from depending on inherited fields.

The technique guarantees that only visible invariants are vulnerable; therefore, only visible invariants need to be proven before method calls and at the end of methods. It also supports helper methods, which we omit here for brevity

6.1.4 Comparison

We compare invariant restrictions, program restrictions, and proof obligations.

Invariant Restrictions ($\mathbb{D}$). Poetzsch-Heffter allows invariants to depend on arbitrary locations, in particular, his technique supports multi-object invariants. Huizing and Kuiper require for multi-object invariants the existence of an access path from the object containing the invariant to the object it depends on. This excludes, for instance, universal quantifications over objects. Leavens and Müller focus on invariants of single objects, and address the subclass challenge by disallowing dependencies on inherited fields.

Program Restrictions ($\mathbb{U}$ and $\mathbb{C}$). All three techniques permit arbitrary method calls. Huizing and Kuiper restrict field updates to the receiver **this**. Leavens and Müller require the updated field to be visible; a requirement enforced by the type system anyway, thus they are not limiting expressiveness.

Proof Obligations ($\mathbb{B}$ and $\mathbb{E}$). Both Poetzsch-Heffter and Huizing and Kuiper impose proof obligations for invariants of essentially all classes of a program (even though Huizing and Kuiper use a syntactic analysis to exclude invariants that are not vulnerable). This makes both techniques highly non-modular. Leavens and Müller's technique requires proof obligations only for visible invariants, which makes this technique modular.

Lemma 14 (Well-Structuredness of Verification Techniques for Unstructured Heaps)**.** *The Poetzsch-Heffter, Huizing and Kuiper and Leavens and Müller are well-structured.*

Proof. We here outline the proof for Poetzsch-Heffter and leave the proof of the remaining two for the interested reader. According to Def. 5, the components of the Poetzsch-Heffter verification technique, given earlier in Fig. 9, have to satisfy the following 5 criteria:

(S1): From $(\mathbb{r} \triangleright \mathbb{X}_{c',m'}) \setminus (\mathbb{X}_{c,m} \setminus \mathbb{V}_{c,m}) \subseteq \mathbb{B}_{c,m,\mathbb{r}}$ we get:

$$\begin{aligned} \mathsf{any} \triangleright \mathsf{any} \setminus (\mathsf{any} \setminus \mathsf{any}) &\subseteq \mathsf{any} \\ \mathsf{any} \triangleright \mathsf{any} \setminus \emptyset &\subseteq \mathsf{any} \\ \mathsf{any} &\subseteq \mathsf{any} \end{aligned}$$

(S2): From $\mathbb{V}_{c,m} \cap \mathbb{X}_{c,m} \subseteq \mathbb{E}_{c,m}$ we get:

$$\mathsf{any} \cap \mathsf{any} \subseteq \mathsf{any}$$
$$\mathsf{any} \subset \mathsf{any}$$

(S3): From $\mathbb{C}_{c,m,c',m'} \triangleright (\mathbb{V}_{c',m'} \setminus \mathbb{E}_{c',m'}) \subseteq \mathbb{V}_{c,m}$ we get:

$$\begin{aligned} \mathsf{any} \triangleright (\mathsf{any} \setminus \mathsf{any}) &\subseteq \mathsf{any} \\ \mathsf{any} \triangleright \emptyset &\subseteq \mathsf{any} \\ \emptyset &\subseteq \mathsf{any} \end{aligned}$$

(S4): From $\mathbb{U}_{c,m,c'} \triangleright \mathbb{D}_{c'} \subseteq \mathbb{V}_{c,m}$ we get:

$$\mathsf{any} \triangleright \mathsf{any} \subseteq \mathsf{any}$$
$$\mathsf{any} \subseteq \mathsf{any}$$

(S5): For $c' <: c$, from $\mathbb{X}_{c',m} \subseteq \mathbb{X}_{c,m}$ we get

$$\mathsf{any} \subseteq \mathsf{any}$$

and from $\mathbb{V}_{c',m} \setminus \mathbb{E}_{c',m} \subseteq \mathbb{V}_{c,m} \setminus \mathbb{E}_{c,m}$ we get

$$\mathsf{any} \setminus \mathsf{any} \subseteq \mathsf{any} \setminus \mathsf{any}$$

□

6.2 Verification Techniques for Structured Heaps

We consider three techniques which strike a better trade-off by using the heap topology enforced by ownership types, and summarise them in Fig. 10.

To sharpen our discussion wrt. structured heaps, we will be adding annotations to the example from Fig. 2, to obtain a topology where the Person p owns the Account a and the DebitCard d.

6.2.1 Müller et al.

Müller, Poetzsch-Heffter, and Leavens [28] present two techniques for multi-object invariants, called ownership technique and visibility technique (OT and VT for short). Both techniques utilise the hierarchic heap topology enforced by Universe types [6, 27]. Universe types associate reference types with ownership modifiers, which specify ownership relative to the current object. The modifier **rep** expresses that an object is owned by the current object; **peer** expresses that an object has the same owner as the current object; **any** expresses that an object may have any owner.

```
class Account {                  class Person {
  peer DebitCard card;             rep Account account;
  any Person holder;               ...
  ...                            }
}

class DebitCard {
  peer Account acc;
  ...
}
```

Figure 11. Universe modifiers for the Account example from Fig. 2.

	Müller et al. (OT)	Müller et al. (VT)	Lu et al.(Oval)
$\mathbb{X}_{c,m}$	$\mathsf{own}\,;\mathsf{rep}^{+}$	$\mathsf{own}\,;\mathsf{rep}^{+}$	$\mathsf{I}\,;\mathsf{rep}^{*}$
$\mathbb{V}_{c,m}$	$\mathsf{super}\langle c\rangle \sqcup \mathsf{own}^{+}$	$\mathsf{peer}\langle c\rangle \sqcup \mathsf{own}^{+}$	$\mathsf{E}\,;\mathsf{own}^{*}$
$\mathbb{D}_{c}$	$\mathsf{self}\langle c\rangle \sqcup \mathsf{own}^{+}$	$\mathsf{peer}\langle c\rangle \sqcup \mathsf{own}^{+}$	$\mathsf{self}\,;\mathsf{own}^{*}$
$\mathbb{B}_{c,m,\mathbb{r}}$	$\mathsf{super}\langle c\rangle$ if $\mathsf{intrsPeer}(\mathbb{r})$ emp otherwise	$\mathsf{peer}\langle c\rangle$ if $\mathsf{intrsPeer}(\mathbb{r})$ emp otherwise	emp
$\mathbb{E}_{c,m}$	$\mathsf{super}\langle c\rangle$	$\mathsf{peer}\langle c\rangle$	self if $\mathsf{I}=\mathsf{E}$ emp otherwise
$\mathbb{U}_{c,m,c'}$	self	peer	self if $\mathsf{I}=\mathsf{E}$ emp otherwise
$\mathbb{C}_{c,m,c',m'}$	$\mathsf{rep}\langle c\rangle \sqcup \mathsf{peer}$	$\mathsf{rep}\langle c\rangle \sqcup \mathsf{peer}$	$\bigsqcup_{\mathbb{r},\text{ with } \mathsf{SC}(\mathsf{I},\mathsf{E},\mathsf{I}',\mathsf{E}',\mathcal{O}_{\mathbb{r},c})} \mathbb{r}$

Figure 10. Components of verification techniques. For Oval, $\mathcal{O}_{\mathbb{r},c}$ is the owner of $\mathbb{r}$; we use shorthands $\mathsf{I} = \mathsf{I}(c,m)$, and $\mathsf{E} = \mathsf{E}(c,m)$, and $\mathsf{I}' = \mathbb{r}\,;\mathsf{I}(c',m')$, and $\mathsf{E}' = \mathbb{r}\,;\mathsf{E}(c',m')$.

Both OT and VT forbid fields f and g declared in different classes c_f and c_g, of the same object o to reference the same object. This *subclass separation* is formalised elegantly by using an ownership model where each object is owned by an object-class pair [18]. In this model, the object referenced from $o.f$ is owned by (o, c_f), whereas the object referenced from $o.g$ is owned by (o, c_g). Since they have different owners, these objects must be different.

We assume a heap operation that yields the owner of an object in a heap: $\text{ownr} : \text{HP} \times \text{ADR} \to \text{ADR} \times \text{CLS}$. The set of areas is:

$$\mathbb{r} \in \mathbf{R} ::= \mathsf{emp} \mid \mathsf{self} \mid \mathsf{rep}\langle \mathbf{c}\rangle \mid \mathsf{peer} \mid \mathsf{any} \mid \mathbb{r} \sqcup \mathbb{r}$$

with the following interpretation:

$$\llbracket \mathsf{self} \rrbracket_{h,\iota} = \{\iota\} \qquad \llbracket \mathsf{any} \rrbracket_{h,\iota} = \text{dom}(h) \qquad \llbracket \mathsf{emp} \rrbracket_{h,\iota} = \emptyset$$
$$\llbracket \mathsf{rep}\langle c\rangle \rrbracket_{h,\iota} = \{\iota' \mid \text{ownr}(h,\iota') = \iota\, c\}$$
$$\llbracket \mathsf{peer} \rrbracket_{h,\iota} = \{\iota' \mid \text{ownr}(h,\iota') = \text{ownr}(h,\iota)\}$$
$$\llbracket \mathbb{r}_1 \sqcup \mathbb{r}_2 \rrbracket_{h,\iota} = \llbracket \mathbb{r}_2 \rrbracket_{h,\iota} \cup \llbracket \mathbb{r}_2 \rrbracket_{h,\iota}$$

In our framework, Universe modifiers intuitively correspond to regions, since they describe areas of the heap. For example, peer describes all objects which share the owner (object-class pair) with the current object. However, because of the subclass separation described above, it is useful to employ richer regions of the form $\mathsf{rep}\langle c\rangle$, describing all objects owned by the current object *and* class c. For regions (and properties) we also include the "union" of two regions (properties). The predicate $\mathsf{intrsPeer}(\mathbb{r})$ checks whether a region intersects the peer region.

The two techniques require a rather rich set of properties to deal with the various aspects of ownership and subclassing:

$$\mathbb{p} \in \mathbf{P} ::= \mathsf{emp} \mid \mathsf{self}\langle c\rangle \mid \mathsf{super}\langle c\rangle \mid \mathsf{peer}\langle c\rangle \mid \mathsf{rep} \mid \mathsf{own} \mid \mathsf{rep}^{+} \mid \mathsf{own}^{+} \mid \mathbb{p};\mathbb{p}$$

with the following interpretations:

$$\llbracket \mathsf{emp} \rrbracket_{h,\iota} = \emptyset \qquad \llbracket \mathsf{self}\langle c\rangle \rrbracket_{h,\iota} = \{(\iota, c) \mid \text{cls}(h,\iota) <: c\}$$
$$\llbracket \mathsf{super}\langle c\rangle \rrbracket_{h,\iota} = \{(\iota, c') \mid c <: c'\}$$
$$\llbracket \mathsf{peer}\langle c\rangle \rrbracket_{h,\iota} = \{(\iota', c') \mid \text{ownr}(h,\iota') = \text{ownr}(h,\iota) \wedge \mathsf{vis}(c', c)\}$$
$$\llbracket \mathsf{rep} \rrbracket_{h,\iota} = \{(\iota', c') \mid \text{ownr}(h,\iota') = \iota\,\_\} \qquad \llbracket \mathsf{own} \rrbracket_{h,\iota} = \{\text{ownr}(h,\iota)\}$$
$$\llbracket \mathbb{p}_1;\mathbb{p}_2 \rrbracket_{h,\iota} = \bigcup_{(\iota',c)\in\llbracket \mathbb{p}_1 \rrbracket_{h,\iota}} \llbracket \mathbb{p}_2 \rrbracket_{h,\iota'}$$
$$\llbracket \mathsf{rep}^{+} \rrbracket_{h,\iota} = \llbracket \mathsf{rep} \rrbracket_{h,\iota} \cup \llbracket \mathsf{rep};\mathsf{rep}^{+} \rrbracket_{h,\iota}$$
$$\llbracket \mathsf{own}^{+} \rrbracket_{h,\iota} = \llbracket \mathsf{own} \rrbracket_{h,\iota} \cup \llbracket \mathsf{own};\mathsf{own}^{+} \rrbracket_{h,\iota}$$

Here we exploit that owners and object invariants both are object-class pairs. Therefore, we can use the owner (o, c) of an object to denote the object invariant for object o declared in class c.

For properties, $\mathsf{self}\langle c\rangle$ represents the singleton set containing a pair of the current object with the class c. The property $\mathsf{super}\langle c\rangle$ represents the set of pairs of the current object with all its classes that are superclasses of c. The property $\mathsf{peer}\langle c\rangle$ represents all the objects (paired with their classes) that share the owner with the current object, provided their class is visible in c. There are also properties to describe the invariants of an object's owned objects, its owner, its transitively owned objects, and its transitive owners. A property of the form $\mathbb{p}_1;\mathbb{p}_2$ denotes a composition of properties, which behaves similarly to function composition when interpreted.

Ownership Technique. As shown in Fig. 10, OT requires that in visible states, all objects owned by the owner of **this** must satisfy their invariants ($\mathbb{X}$).

Invariants are allowed to depend on fields of the object itself (at the current class), as in I1 in Fig. 2, and all its **rep** objects, as in I2. Other client invariants such as I4 and I5) and subclass invariants that depend on inherited fields (such as I3) are not permitted. Therefore, a field update potentially affects the invariants of the modified object and of all its (transitive) owners ($\mathbb{D}$).

A method may update fields of **this** ($\mathbb{U}$). Since an updated field is declared in the enclosing class or a superclass, the invariants potentially affected by the update are those of **this** (for the enclosing class and its superclasses, which addresses the subclass challenge) as well as the invariants of the (transitive) owners of **this** ($\mathbb{V}$).

OT handles multi-object invariants by allowing invariants to depend on fields of owned objects ($\mathbb{D}$). Therefore, methods may break the invariants of the transitive owners of **this** ($\mathbb{V}$). For example, the invariant I2 of Person (Fig. 2) is legal only because account is a **rep** field (Fig. 11). Account's method withdraw need not preserve Person's invariant. This is reflected by the definition of $\mathbb{E}$: only the invariants of **this** are proven at the end of the method, while those of the transitive owners may remain broken; it is the responsibility of the owners to re-establish them, which addresses the multi-object challenge. As an example, the method spend has to re-establish Person's invariant after the call to account.withdraw.

Since the invariants of the owners of **this** might not hold, OT disallows calls on references other than **rep** and **peer** references ($\mathbb{C}$). For instance, the call holder . notify () in method sendReport is not permitted because holder is in an ancestor ownership context.

The proof obligations for method calls ($\mathbb{B}$) must cover those invariants expected by the callee that are vulnerable to the caller. This intersection contains the invariant of the caller, if the caller and the callee are peers because the callee might call back; it is otherwise empty (**rep**s cannot callback their owners).

Visibility Technique. VT relaxes the restrictions of OT in two ways. First, it permits invariants of a class c to depend on fields of peer objects, provided that these invariants are visible in c ($\mathbb{D}$). Thus, VT can handle multi-object structures that are not organised hierarchically. For instance, in addition to the invariants permitted by OT, VT permits invariants I4 and I5 in Fig. 2. Visibility is transitive, thus, the invariant must also be visible wherever fields of c are updated. Second, VT permits field updates on peers of **this** ($\mathbb{U}$).

These relaxations make more invariants vulnerable. Therefore, $\mathbb{V}$ includes additionally the invariants of the peers of **this**. This addition is also reflected in the proof obligations before peer calls ($\mathbb{B}$) and before the end of a method ($\mathbb{E}$). For instance, method withdraw must be proven to preserve the invariant of the associated DebitCard, which does not in general succeed in our example.

Lemma 15. OT *and* VT *are well-structured.*

Proof. The proof assumes the following definition of region inclusion for the universe type system, defined as the least relation characterised by the rules below. It is not hard to see that this definition satisfies constraint (P4) of Def. 30. Other definitions for universe set inclusion are possible.

$$\text{(u-emp)}\ \frac{}{\mathsf{emp} \sqsubseteq \mathbb{r}} \qquad \text{(u-any)}\ \frac{}{\mathbb{r} \sqsubseteq \mathsf{any}} \qquad \text{(u-self)}\ \frac{}{\mathsf{self} \sqsubseteq \mathsf{peer}}$$

$$\text{(u-union)}\ \frac{}{\mathbb{r}_1 \sqsubseteq \mathbb{r}_1 \sqcup \mathbb{r}_2 \quad \mathbb{r}_2 \sqsubseteq \mathbb{r}_1 \sqcup \mathbb{r}_2} \qquad \text{(u-relf)}\ \frac{}{\mathbb{r} \sqsubseteq \mathbb{r}} \qquad \text{(u-trans)}\ \frac{\mathbb{r}_1 \sqsubseteq \mathbb{r}_2 \quad \mathbb{r}_2 \sqsubseteq \mathbb{r}_3}{\mathbb{r}_1 \sqsubseteq \mathbb{r}_3}$$

We start by showing that OT is well-structured from the components given in Fig. 10.

(S1): There are a number of regions that satisfy $\mathbb{r} \sqsubseteq \mathbb{C}_{c,m,c'm'}$. We here give the proof for the main two cases, i.e., peer and rep$\langle c\rangle$. For $\mathbb{r} = \mathsf{peer}$ we have to show:

$$(\mathsf{peer} \triangleright \mathsf{own}\,;\mathsf{rep}^+) \setminus (\mathsf{own}\,;\mathsf{rep}^+ \setminus \mathsf{super}\langle c\rangle \sqcup \mathsf{own}^+) \subseteq \mathsf{super}\langle c\rangle$$

When we adapt own ; rep$^+$, i.e., everything beneath the owner of the current receiver, by peer, i.e., peer $\triangleright$ own ; rep$^+$, we still get own ; rep$^+$ since peers share the same owner. Thus we get

$$\mathsf{own}\,;\mathsf{rep}^+ \setminus (\mathsf{own}\,;\mathsf{rep}^+ \setminus \mathsf{super}\langle c\rangle \sqcup \mathsf{own}^+) \subseteq \mathsf{super}\langle c\rangle$$

Using the set identity

$$A \setminus (B \setminus C) = (A \cap C) \cup (A \setminus B) \tag{96}$$

on the left hand of the inclusion we get:

$$(\mathsf{own}\,;\mathsf{rep}^+ \cap \mathsf{super}\langle c\rangle \sqcup \mathsf{own}^+) \cup (\mathsf{own}\,;\mathsf{rep}^+ \setminus \mathsf{own}\,;\mathsf{rep}^+) \subseteq \mathsf{super}\langle c\rangle$$

and thus

$$(\mathsf{own}\,;\mathsf{rep}^+ \cap \mathsf{super}\langle c\rangle \sqcup \mathsf{own}^+) \cup \emptyset \subseteq \mathsf{super}\langle c\rangle \tag{97}$$

From the interpretations given, we can show that

$$\mathsf{own}\,;\mathsf{rep}^+ \cap \mathsf{super}\langle c\rangle \sqcup \mathsf{own}^+ = \mathsf{super}\langle c\rangle$$

and as a result, from (97) we obtain

$$\mathsf{super}\langle c\rangle \subseteq \mathsf{super}\langle c\rangle$$

For the second case, i.e., $\mathbb{r} = \mathsf{rep}\langle c\rangle$ we have

$$(\mathsf{rep}\langle c\rangle \triangleright \mathsf{own}\,;\mathsf{rep}^+) \setminus (\mathsf{own}\,;\mathsf{rep}^+ \setminus \mathsf{super}\langle c\rangle \sqcup \mathsf{own}^+) \subseteq \mathsf{emp}$$

When we adapt own ; rep$^+$ by rep$\langle c\rangle$ we get all objects transitively owned by the current receiver, namely rep$^+$ and thus we get

$$\mathsf{rep}^+ \setminus (\mathsf{own}\,;\mathsf{rep}^+ \setminus \mathsf{super}\langle c\rangle \sqcup \mathsf{own}^+) \subseteq \mathsf{emp}$$

At this point we apply the set identity (96) from the previous case and get

$$(\mathsf{rep}^+ \cap \mathsf{super}\langle c\rangle \sqcup \mathsf{own}^+) \cup (\mathsf{rep}^+ \setminus \mathsf{own}\,;\mathsf{rep}^+) \subseteq \mathsf{emp}$$

Since rep$^+$ does not include the current receiver, we know $\mathsf{rep}^+ \cap \mathsf{super}\langle c\rangle \sqcup \mathsf{own}^+ = \emptyset$. Also since $\mathsf{rep}^+ \subseteq \mathsf{own}\,;\mathsf{rep}^+$, we also know $\mathsf{rep}^+ \setminus \mathsf{own}\,;\mathsf{rep}^+ = \emptyset$ and hence we get

$$\emptyset \cup \emptyset \subseteq \mathsf{emp}$$

(S2): We require

$$\mathsf{super}\langle c\rangle \sqcup \mathsf{own}^+ \cap \mathsf{own}\,;\mathsf{rep}^+ \subseteq \mathsf{super}\langle c\rangle \tag{98}$$

From the interpretations, it is not hard to show directly that

$$\mathsf{super}\langle c\rangle \cap \mathsf{own}\,;\mathsf{rep}^+ = \mathsf{super}\langle c\rangle \tag{99}$$

$$\mathsf{own}^+ \cap \mathsf{own}\,;\mathsf{rep}^+ = \emptyset \tag{100}$$

from which (98) follows.

(S3): Instantiating the components of Fig. 10 we need to show

$$\mathsf{rep}\langle c'\rangle \sqcup \mathsf{peer} \triangleright ((\mathsf{super}\langle c\rangle \sqcup \mathsf{own}^+) \backslash \mathsf{super}\langle c\rangle) \subseteq \mathsf{super}\langle c'\rangle \sqcup \mathsf{own}^+$$

We highlight the different roles played by the classes c and c' in the above statement. It is easy to show that

$$\mathsf{own}^+ \cap \mathsf{super}\langle c\rangle = \emptyset \tag{101}$$

from which we obtain

$$(\mathsf{super}\langle c\rangle \sqcup \mathsf{own}^+) \setminus \mathsf{super}\langle c\rangle = \mathsf{own}^+ \tag{102}$$

Therefore, it suffices to show

$$\mathsf{rep}\langle c'\rangle \sqcup \mathsf{peer} \triangleright (\mathsf{own}^+) \subseteq \mathsf{super}\langle c'\rangle \sqcup \mathsf{own}^+$$

From the interpretations we derive the identities:

$$\mathsf{rep}\langle c'\rangle \triangleright \mathsf{own}^+ = \mathsf{self}\langle c'\rangle \sqcup \mathsf{own}^+ \tag{103}$$

$$\mathsf{peer} \triangleright \mathsf{own}^+ = \mathsf{own}^+ \tag{104}$$

and thus, from the direct interpretation of $\sqcup$ we obtain

$$\mathsf{self}\langle c'\rangle \cup \mathsf{own}^+ \subseteq \mathsf{super}\langle c'\rangle \cup \mathsf{own}^+$$

which is immediately true since, from the interpretations we know $\mathsf{self}\langle c'\rangle \subseteq \mathsf{super}\langle c'\rangle$

(S4): Once again we have two cases.

- For pure methods we have

$$\mathsf{emp} \triangleright \mathsf{self}\langle c\rangle \sqcup \mathsf{own}^+ \subseteq \mathsf{super}\langle c\rangle \sqcup \mathsf{own}^+$$

  Since the interpretation of emp is $\emptyset$, anything adapted by the viewpoint emp given emp and thus

$$\mathsf{emp} \subseteq \mathsf{super}\langle c\rangle \sqcup \mathsf{own}^+$$

  which is trivially true.

- For non-pure methods we have

$$\mathsf{self} \triangleright \mathsf{self}\langle c\rangle \sqcup \mathsf{own}^+ \subseteq \mathsf{super}\langle c\rangle \sqcup \mathsf{own}^+$$

  Any adaptation by self acts as the identity and thus we obtain

$$\mathsf{self}\langle c\rangle \sqcup \mathsf{own}^+ \subseteq \mathsf{super}\langle c\rangle \sqcup \mathsf{own}^+$$

  which is true by the same reasons we gave for case **(S3)** above.

(S5): For $c <: c'$ we have to show

1. $$\mathsf{own}\,;\mathsf{rep}^+ \subseteq \mathsf{own}\,;\mathsf{rep}^+$$
2. $$\begin{pmatrix} \mathsf{super}\langle c\rangle \sqcup \mathsf{own}^+ \backslash \\ \mathsf{super}\langle c\rangle \end{pmatrix} \subseteq \begin{pmatrix} \mathsf{super}\langle c'\rangle \sqcup \mathsf{own}^+ \backslash \\ \mathsf{super}\langle c\rangle \end{pmatrix}$$

The first statement is trivially true whereas the second statement is true because (102) is true for any c, so substituting the right hand side of (102) gives us own$^+$ on both sides.

The proof of well-structuredness of the VT is similar to that of the OT:

(S1): As before, the two cases for $\mathbb{r}$ we consider are peer and $\mathsf{rep}\langle c\rangle$. For peer we have the proof

$$\begin{array}{rl}
\mathsf{peer} \triangleright \mathsf{own}\,;\mathsf{rep}^+ \setminus \begin{pmatrix} \mathsf{own}\,;\mathsf{rep}^+ \setminus \\ \mathsf{peer}\langle c\rangle \sqcup \mathsf{own}^+ \end{pmatrix} & \subseteq \mathsf{peer}\langle c\rangle \\
\mathsf{own}\,;\mathsf{rep}^+ \setminus \begin{pmatrix} \mathsf{own}\,;\mathsf{rep}^+ \setminus \\ \mathsf{peer}\langle c\rangle \sqcup \mathsf{own}^+ \end{pmatrix} & \subseteq \mathsf{peer}\langle c\rangle \\
\begin{pmatrix} \mathsf{own}\,;\mathsf{rep}^+ \cap \\ \mathsf{peer}\langle c\rangle \sqcup \mathsf{own}^+ \end{pmatrix} \cup \begin{pmatrix} \mathsf{own}\,;\mathsf{rep}^+ \setminus \\ \mathsf{own}\,;\mathsf{rep}^+ \end{pmatrix} & \subseteq \mathsf{peer}\langle c\rangle \\
\begin{pmatrix} \mathsf{own}\,;\mathsf{rep}^+ \cap \\ \mathsf{peer}\langle c\rangle \sqcup \mathsf{own}^+ \end{pmatrix} \cup \emptyset & \subseteq \mathsf{peer}\langle c\rangle
\end{array}$$

Here we use the property

$$\mathsf{own}\,;\mathsf{rep}^+ \cap \mathsf{peer}\langle c\rangle \sqcup \mathsf{own}^+ = \mathsf{peer}\langle c\rangle \qquad (105)$$

derived directly from the interpretations and get

$$\mathsf{peer}\langle c\rangle \cup \emptyset \subseteq \mathsf{peer}\langle c\rangle$$

For the case of $\mathsf{rep}\langle c\rangle$ we have the proof:

$$\begin{array}{rl}
\mathsf{rep}\langle c\rangle \triangleright \mathsf{own}\,;\mathsf{rep}^+ \setminus \begin{pmatrix} \mathsf{own}\,;\mathsf{rep}^+ \setminus \\ \mathsf{peer}c \sqcup \mathsf{own}^+ \end{pmatrix} & \subseteq \mathsf{emp} \\
\mathsf{rep}^+ \setminus \begin{pmatrix} \mathsf{own}\,;\mathsf{rep}^+ \setminus \\ \mathsf{peer}c \sqcup \mathsf{own}^+ \end{pmatrix} & \subseteq \mathsf{emp} \\
(\mathsf{rep}^+ \cap \mathsf{peer}\langle c\rangle \sqcup \mathsf{own}^+) \cup (\mathsf{rep}^+ \setminus \mathsf{own}\,;\mathsf{rep}^+) & \subseteq \mathsf{emp} \\
\emptyset \cup \emptyset & \subseteq \mathsf{emp}
\end{array}$$

(S2): There are two cases.

- For pure methods we have

$$\begin{array}{rl}
\mathsf{emp} \cap \mathsf{own}\,;\mathsf{rep}^+ & \sqsubseteq \mathsf{emp} \\
\mathsf{emp} & \sqsubseteq \mathsf{emp}
\end{array}$$

- For non-pure methods we have

$$\mathsf{peer}\langle c\rangle \sqcup \mathsf{own}^+ \cap \mathsf{own}\,;\mathsf{rep}^+ \sqsubseteq \mathsf{peer}\langle c\rangle$$

which is true from (105) earlier.

(S3): There are four cases to consider here, depending on the purity of the two methods. We here give the proof for two cases.

- If m is pure and m' is non-pure we have

$$\begin{array}{ll}
\mathsf{emp} \triangleright (\mathsf{peer}\langle c'\rangle \sqcup \mathsf{own}^+ \setminus \mathsf{peer}\langle c'\rangle) & \sqsubseteq \mathsf{emp} \\
\mathsf{emp} \sqsubseteq \mathsf{emp} &
\end{array}$$

- When both m and m' are non-pure, then since we can directly show

$$\mathsf{own}^+ \cap \mathsf{peer} = \qquad \emptyset\triangleright \qquad (106)$$

we have

$$\begin{array}{ll}
\mathsf{rep}\langle c\rangle \sqcup \mathsf{peer} \triangleright \begin{pmatrix} \mathsf{peer}\langle c'\rangle \sqcup \mathsf{own}^+ \\ \setminus \mathsf{peer}\langle c'\rangle \end{pmatrix} & \sqsubseteq \mathsf{peer}\langle c\rangle \sqcup \mathsf{own}^+ \\
\mathsf{rep}\langle c\rangle \sqcup \mathsf{peer} \triangleright \mathsf{own}^+ & \sqsubseteq \mathsf{peer}\langle c\rangle \sqcup \mathsf{own}^+
\end{array}$$

At this point we use the identities (103), (104) derived earlier to obtain

$$\mathsf{self}\langle c\rangle \sqcup \mathsf{own}^+ \sqsubseteq \mathsf{peer}\langle c\rangle \sqcup \mathsf{own}^+$$

which is true because we can show $\mathsf{self}\langle c\rangle \sqsubseteq \mathsf{peer}\langle c\rangle$.

(S4): We have two cases.

- If $\mathbb{U}_{c,m,c'} = \mathsf{emp}$ we have two of cases and here we consider the case where m is not pure (the other case is similar).

$$\begin{array}{rl}
\mathsf{emp} \triangleright \mathsf{peer}\langle c'\rangle \sqcup \mathsf{own}^+ & \sqsubseteq \mathsf{peer}\langle c\rangle \sqcup \mathsf{own}^+ \\
\mathsf{emp} & \sqsubseteq \mathsf{peer}\langle c\rangle \sqcup \mathsf{own}^+
\end{array}$$

- If $\mathbb{U}_{c,m,c'} = \mathsf{peer}$ then we know that m is not pure and that c is visible from c', i.e., $\mathsf{vis}(c', c)$. We therefore obtain the proof

$$\begin{array}{rl}
\mathsf{peer} \triangleright \mathsf{peer}\langle c'\rangle \sqcup \mathsf{own}^+ & \sqsubseteq \mathsf{peer}\langle c\rangle \sqcup \mathsf{own}^+ \\
\mathsf{peer}\langle c'\rangle \sqcup \mathsf{own}^+ & \sqsubseteq \mathsf{peer}\langle c\rangle \sqcup \mathsf{own}^+
\end{array}$$

and by the symmetric property of $\mathsf{vis}(c', c)$ and the interpretation of $\mathsf{peer}\langle c\rangle$ we derive the identity $\mathsf{peer}\langle c\rangle = \mathsf{peer}\langle c'\rangle$ which make the above true.

(S5): This is similar to (S5) for OT.

□

6.2.2 Lu et al.

Lu, Potter, and Xue [23] define Oval, a verification technique based on ownership types, which support owner parameters for classes [5], thus permitting a more precise description of the heap topology. The distinctive features of Oval are: (1) Expected and vulnerable invariants are specific to every method in every class through the notion of *contracts*. (2) Invariant restrictions do not take subclassing into account. (3) Proof obligations are only imposed at the end of calls. (4) To address the call-back challenge, calls are subject to "subcontracting", a requirement that guarantees that the expected and vulnerable invariants of the callee are within those of the caller.

Here we describe Oval', an adaptation of Oval, where i) we omit non-rep fields, a refinement whereby the invariant of the current object cannot depend on such fields (but its owners can), and ii) we drop the existential class parameter "*" annotation - both features enhance programming expressivity of Oval, but are deemed as non-central to our analysis. Oval' also used different restrictions for method overriding, because the original restrictions defined in Oval lead to unsoundness [22], as we discuss later on. In [23] description of the Oval verification technique is intertwined with that of the ownership type system it is based on. However, for the presentation of Oval', we strive to disentangle the two.

Oval' classes have owner parameters, indicated by X, Y, and the subclass relationship is described through a judgment $c\langle\overline{X}\rangle \lhd c'\langle\overline{X'}\rangle$ defined as:

$$\frac{\texttt{class}\ c\langle\overline{X}\rangle\ \texttt{extends}\ c\langle\overline{X}\rangle \ldots}{\begin{array}{c} c\langle\overline{X}\rangle \lhd c\langle\overline{X}\rangle \\ c\langle\overline{X}\rangle \lhd c'\langle\overline{X'}\rangle \end{array}} \qquad \frac{\begin{array}{c} c\langle\overline{X}\rangle \lhd c'\langle\overline{X'}\rangle \\ c'\langle\overline{Y}\rangle \lhd c''\langle\overline{Y'}\rangle \end{array}}{c\langle\overline{X}\rangle \lhd c''\langle\{^{\overline{X'}}/_{\overline{Y}}\}\overline{Y'}\rangle}$$

where $\overline{X}$ are the disjoint formal class parameters of c in the program.

For simplicity we require that the formal class parameters are disjoint for every class. This assumption is very powerful, as, in contrast to usual systems, it allows the $\lhd$ relationship to be context independent. An Oval program also defines an "inside" partial order relation, $\preceq$ for parameters of the same class.

Fig. 12 shows our example in Oval using ownership parameters [5] to describe heap topologies. The ownership parameter o denotes the owner of the current object; p denotes the owner of o and specifies the position of holder in the hierarchy, more precisely than the **any** modifier in Universe types.

Method Contracts. Ownership parameters are also used to describe expected and vulnerable invariants, which are specific to each method. Every Oval program extends method signatures with a contract $\langle \mathsf{I}, \mathsf{E}\rangle$: the expected invariants at visible states ($\mathbb{X}$) are the invariants of the object characterised by I and all objects transitively owned by this object; the vulnerable invariants ($\mathbb{V}$) are the object at

```
class Account[o,p] {
  DebitCard⟨o⟩ card;
  Person⟨p⟩ holder;
   ...
  void withdraw(int amount)⟨this,this⟩
    { ... }
  void sendReport()⟨bot,p⟩
    { ... }
}

class Person[o] {
  Account⟨this⟩ account;
   ...
  void spend(int amount)⟨this,this⟩
    { account.withdraw(amount); }
  void notify ()⟨bot,top⟩
    { ... }
}
```
Figure 12. Ownership parameters and method contracts in Oval.

E and its transitive owners. These properties are syntactically characterised by Ls in the code (and Ks in typing rules), where:

$$\mathrm{L} ::= \mathsf{top} \mid \mathsf{bot} \mid \mathsf{this} \mid X \qquad\qquad \mathrm{K} ::= \mathrm{L} \mid \mathrm{K}\,;\mathsf{rep}$$

and where X stands for the class' owner parameters. "Contexts" L, obtained from [23], are syntactic descriptions of the standard and vulnerable properties. As in [23], the type system extends L to K to described context abstraction[23], i.e., objects owned by class parameters, and generalises the partial ordering $\preceq$ to K as a lattice bounded by top and bot, using rules from [23]. As in [23], the type system defines the judgement

$$c\langle\overline{\mathrm{K}}\rangle <: c\langle\overline{\mathrm{K}}\rangle \;\Leftrightarrow\; c\langle\overline{X}\rangle \lhd c'\langle\overline{X'}\rangle,\ \forall i.\mathrm{K}'_i = \{\overline{\mathrm{K}}/\bar{X}\}\bar{X}'$$

As in [23], the type system also requires all classes c and methods m to satisfy

$$\mathsf{I}(c,m) \preceq \mathsf{E}(c,m) \qquad \mathsf{I}(c,m) = \mathsf{E}(c,m) \Rightarrow \mathsf{I}(c,m) = \mathsf{this} \qquad (107)$$

which guarantees that the expected and the vulnerable invariants of every method can intersect at most at the current object. Central to Oval is *subcontracting*, which we adopt for Oval'(modulo renaming).

In class Account (Fig. 12), withdraw() expects the current object and the objects it transitively owns to be valid (I=**this**) and, during execution, this method may invalidate the current object and its transitive owners (E=**this**). The contract of sendReport() does not expect any objects to be valid at visible states (I=**bot**) but may violate object p and its transitive owners (E=p).

Subcontracting. Call-backs are handled via *subcontracting*, which is defined using the order $\mathrm{L} \preceq \mathrm{L}'$.To interpret Oval's subcontracting in our framework, we use $\mathsf{SC}(\mathsf{I},\mathsf{E},\mathsf{I}',\mathsf{E}',\mathrm{K})$, which holds iff:

$$\mathsf{I} \prec \mathsf{E} \Rightarrow \mathsf{I}' \preceq \mathsf{I} \qquad \mathsf{I} = \mathsf{E} \Rightarrow \mathsf{I}' \prec \mathsf{I} \qquad \mathsf{I}' \prec \mathsf{E}' \Rightarrow \mathsf{E} \preceq \mathsf{E}' \qquad \mathsf{I}' = \mathsf{E}' \Rightarrow \mathsf{E} \preceq \mathrm{K},$$

where I, E characterise the caller, I′, E′ characterise the callee, and K stands for the callee's owner. The first two requirements ensure that the caller guarantees the invariant expected by the callee. The other two conditions ensure that the invariants vulnerable to the callee are also vulnerable to the caller. For instance, the call holder . notify () in method sendReport satisfies subcontracting because caller and callee do not expect any invariants, and the callee has no vulnerable invariants. In particular, the receiver of a call may be owned by any of the owners of the current receiver, provided that subcontracting is respected ($\mathbb{C}$).

Given that $\mathsf{I} \preceq \mathsf{E}$ for all well-formed methods, and that $\mathbb{B}_{c,m,\mathbb{r}}$ =emp, the first two requirements of subcontracting exactly give (S1), while the latter two exactly give (S3) from Def. 5.

$$\frac{}{\mathsf{emp} \sqsubseteq \mathbb{r}} \qquad \frac{c\langle\overline{X}\rangle \lhd c\langle\overline{X'}\rangle}{c\langle\overline{\mathrm{K}}\rangle \sqsubseteq c\langle\{\overline{\mathrm{K}}/\overline{X}\}\overline{X'}\rangle} \qquad \frac{}{\mathbb{r} \sqsubseteq \mathbb{r} \sqcup \mathbb{r}'}$$

$$\frac{}{\mathrm{K} \sqsubseteq \mathrm{K}\,;\mathsf{rep}^*} \qquad \frac{}{\mathrm{K} \sqsubseteq \mathrm{K}\,;\mathsf{own}^*}$$

$$\frac{\mathrm{K} \preceq \mathrm{K}'}{\mathrm{K}\,;\mathsf{rep}^* \sqsubseteq \mathrm{K}'\,;\mathsf{rep}^*} \qquad \frac{\mathrm{K} \preceq \mathrm{K}'}{\mathrm{K}'\,;\mathsf{own}^* \sqsubseteq \mathrm{K}\,;\mathsf{own}^*}$$

Figure 13. The $\sqsubseteq$ relation for Oval'

The heap model defines an additional operation typ which gives the runtime type of each object, $c\langle\bar{\iota}\rangle$ where:

$$\mathrm{typ}(h,\iota) = c\langle\bar{\iota}\rangle \;\Rightarrow\; \mathrm{cls}(h,\iota) = c,\ c\langle\overline{X}\rangle \lhd c'\langle\overline{X'}\rangle,\ |\bar{\iota}| = |\overline{X}|$$

The owner of ι above is ι_1. We define address runtime typing and address ownership as:

$$h \vdash \iota : c\langle\bar{\iota}\rangle \;\Leftrightarrow\; \begin{cases} \mathrm{typ}(h,\iota) = c'\langle\overline{\iota'}\rangle,\ c'\langle\overline{X'}\rangle \lhd c\langle\overline{X}\rangle, \\ \forall i.\iota_i = \{\overline{\iota'}/\overline{X'}\}X_i \end{cases}$$

$$h \vdash \iota' \preceq \iota \;\Leftrightarrow\; \mathrm{typ}(h,\iota') = c\langle\iota,\bar{\iota}\rangle$$

$$h \vdash \iota' \preceq^* \iota \;\Leftrightarrow\; \iota' = \iota \vee \exists\iota''.h \vdash \iota' \preceq \iota'',\ h \vdash \iota'' \preceq^* \iota$$

Regions and Properties. To express Oval in our framework, we define regions and properties as follows (see App. **??** for their interpretations):

$$\mathbb{r} \in \mathbf{R} \quad ::= \mathsf{emp} \mid \mathsf{self} \mid c\langle\overline{\mathrm{K}}\rangle \mid \mathbb{r} \sqcup \mathbb{r} \qquad\quad \mathbb{p} \in \mathbf{P} \quad ::= \mathsf{emp} \mid \mathsf{self} \mid \mathrm{K} \mid \mathrm{K}\,;\mathsf{rep}^* \mid \mathrm{K}\,;$$

Remark 16. *Note, that our definition of regions introduces some redundancy, because a type $t = \mathbb{r}\ c$ would have the shape,* e.g.,*`C<rep,o2>` C. This redundancy is harmless.*

The interpretation for regions and properties is based on the interpretation of extended contexts:

$$\{\!\![\mathsf{top}]\!\!\}_{h,\iota} = \{\!\![\mathsf{bot}]\!\!\}_{h,\iota} = \emptyset \qquad\qquad \{\!\![\mathsf{this}]\!\!\}_{h,\iota} = \{\iota\}$$

$$\{\!\![X]\!\!\}_{h,\iota} = \{\iota'_i \mid h \vdash \iota : c\langle\bar{\iota}\rangle,\ c\langle\overline{X}\rangle \lhd \_,\ X = X_i\}$$

$$\{\!\![\mathrm{K}\,;\mathsf{rep}]\!\!\}_{h,\iota} = \{\iota' \mid \iota'' \in \{\!\![oEffK]\!\!\}_{h,\iota},\ h \vdash \iota' \preceq \iota''\}$$

The interpretation of regions is:

$$[\![\mathsf{emp}]\!]_{h,\iota} = \emptyset \qquad [\![\mathsf{this}]\!]_{h,\iota} = \{\iota\} \qquad [\![\mathbb{r} \sqcup \mathbb{r}']\!]_{h,\iota} = [\![\mathbb{r}]\!]_{h,\iota} \cup [\![\mathbb{r}']\!]_{h,\iota}$$

$$[\![c\langle\overline{\mathrm{K}}\rangle]\!]_{h,\iota} = \{\iota' \mid h \vdash \iota' : c\langle\bar{\iota}\rangle, \forall i.\, \iota_i \in \{\!\![\mathrm{K}_i]\!\!\}_{h,\iota}\}$$

The interpretation for properties is as follows:

$$[\![\mathsf{emp}]\!]_{h,\iota} = [\![\mathsf{top}]\!]_{h,\iota} = [\![\mathsf{bot}]\!]_{h,\iota} = \emptyset$$

$$[\![\mathrm{K}]\!]_{h,\iota} = \{(\iota',c) \mid \iota' \in \{\!\![\mathrm{K}]\!\!\}_{h.\iota},\ \mathrm{cls}(h,\iota') <: c\}$$

$$[\![\mathrm{K};\mathbb{p}]\!]_{h,\iota} = \begin{cases} \text{all} & \mathrm{K}=\mathsf{top}, \mathbb{p}=\mathsf{rep}^* \ \vee \mathrm{K}=\mathsf{bot}, \mathbb{p}=\mathsf{own}^* \\ \bigcup_{(\iota',c)\in[\![\mathrm{K}]\!]_{h,\iota}} [\![\mathbb{p}]\!]_{h,\iota'} & \mathbb{p} \in \{\mathsf{rep}^*, \mathsf{own}^*\} \end{cases}$$

$$[\![\mathsf{rep}^*]\!]_{h,\iota} = \{\iota' \mid h \vdash \iota' \preceq^* \iota\} \qquad [\![\mathsf{own}^*]\!]_{h,\iota} = \{\iota' \mid h \vdash \iota \preceq^* \iota'\}$$

Based on the ordering $\preceq$, we define the reflexive and transitive judgment $\sqsubseteq$ for regions and properties in Fig. 13. Based on the viewpoint type adaptation of the Oval type system[23] we define the "adaptation" operation **;** between regions and contexts L, returning extended contexts K:

$$\mathbb{r}\,;\mathrm{L} = \begin{cases} \mathrm{L} & \text{if } \mathbb{r} = \mathsf{this}; \\ \mathrm{K}_i & \text{if } \mathbb{r} = c\langle\overline{\mathrm{K}}\rangle, \mathrm{L} = X_i; \\ \mathrm{K}_1\,;\mathsf{rep} & \text{if } \mathbb{r} = c\langle\overline{\mathrm{K}}\rangle, \mathrm{L} = \mathsf{this} \\ \perp & \text{otherwise.} \end{cases}$$

from which we define the viewpoint adaptation operation

$$\mathbb{r} \triangleright \mathbb{p} = \begin{cases} \mathsf{emp} & \mathbb{r} = \mathsf{emp} \vee \mathbb{p} = \mathsf{emp} \\ \mathbb{p} & \mathbb{r} = \mathsf{this} \\ \mathbb{p}_1 \sqcup \mathbb{p}_2 & \mathbb{r} = \mathbb{r}_1 \sqcup \mathbb{r}_2,\ \mathbb{p}_i = \mathbb{r}_i \triangleright \mathbb{p} \\ \mathrm{K}_1; \mathsf{rep} & \mathbb{r} = c\langle\overline{\mathrm{K}}\rangle,\ \mathbb{p} = \mathsf{this} \\ \mathrm{K}_i & \mathbb{r} = c\langle\overline{\mathrm{K}}\rangle,\ \mathbb{p} = X_i \\ (\mathbb{r}; \mathrm{K}); \mathbb{p}' & \mathbb{r} = c\langle\overline{\mathrm{K}}\rangle,\ \mathbb{p} = \mathrm{K}; \mathbb{p}',\ \mathbb{p}' \in \{\mathsf{rep}, \mathsf{rep}^*, \mathsf{own}^*\} \end{cases}$$

As already stated, expected and vulnerable properties depend on the contract of the method and express $\mathbb{X}$ as $\mathsf{I}\,;\mathsf{rep}^*$ and $\mathbb{V}$ as $\mathsf{E}\,;\mathsf{own}^*$ (see Fig. 10). Similarly to OT, invariant dependencies are restricted to an object and the objects it transitively owns ($\mathbb{D}$). Therefore, I1 and I4 are legal, as well as I3, which depends on an inherited field. Oval imposes a restriction on contracts that the expected and vulnerable invariants of every method intersect at most at this. Consequently, at the end of a method, one has to prove the invariant of the current receiver, if $\mathsf{I} = \mathsf{E} = \mathsf{this}$, and nothing otherwise ($\mathbb{E}$). In the former case, the method is allowed to update fields of its receiver; no updates are allowed otherwise ($\mathbb{U}$). Therefore, spend and withdraw are the only methods in our example that are allowed to make field updates. Oval does not impose proof obligations on method calls ($\mathbb{B}$ is empty), but addresses the call-back challenge through subcontracting. Therefore, call-backs are safe because the callee cannot expect invariants that are temporarily broken. With the existing contracts in Fig. 12, subcontracting permits spend to call account.withdraw(), and withdraw to call **this** .sendReport(), and also sendReport to call holder . notify () . The last two subcalls may potentially lead to callbacks, but are safe because the contracts of sendReport and notify do not expect the receiver to be in a valid state ($\mathsf{I} = \mathbf{bot}$).

Subclassing and Subcontracting. Oval also requires subcontracting between a superclass method and an overriding subclass method. As we discuss later, this does not guarantee soundness [22], and we found a counterexample (cf. Sec. 5). Therefore, we require that a subclass expects no more than the superclass, and vice versa for vulnerable invariants, and that if an expected invariant in the superclass is vulnerable in the subclass, then it must also be expected in the subclass:[5]

$$\mathsf{I}' \preceq \mathsf{I} \preceq \mathsf{E} \preceq \mathsf{E}' \qquad \mathsf{I} = \mathsf{E}' \Rightarrow \mathsf{I}' = \mathsf{E}'$$

where $\mathsf{I}, \mathsf{E}, \mathsf{I}', \mathsf{E}'$ characterise the superclass, resp. subclass, method. This requirement gives exactly (S5) from Def. 5. It allows $\mathsf{I}' = \mathsf{I} = \mathsf{E} = \mathsf{E}'$ which is forbidden in Oval. We use the above requirement for Oval$'$.

Results.

Lemma 17. *Oval' is a programming language in the sense of definition 30. Also, Oval' has a sound type system in the sense of definition 32.*

Remark 18. *Note also, that usually in ownership type systems, and indeed in most systems with parameterized classes, the field and method lookup functions, $\mathcal{F}$, $\mathcal{M}$ and $\mathcal{B}$ are defined on types, rather than classes. For instance, one would expect to have $\mathcal{F}(c\langle o1, o2\rangle, f)$ rather than $\mathcal{F}(c, f)$ as in our framework. In contrast, in our framework, these functions are defined on classes. Namely, as we have requested the owner parameters to be disjoint across different classes, the meaning of.* e.g.,, *$\mathcal{F}(c, f)$ is, implicitly that of $\mathcal{F}(c\langle c1, c2\rangle, f)$ where $c1, c2$ are the formal ownership parameters of class c.*

Furthermore, in contrast to usual practice in ownership types, and parameterized classes, the type of an inherited field (or method) remains the same (as required in Def. 30, part F2 *and* F3 *of Def. 31. Again, because the owner parameters are disjoint across classes, we can make this simplification. For example, for*

```
class C<c1>{  A<c1> f; }
class D<d1> extends C<c1> { }
```

we would have that $\mathcal{F}(C, f)$=$\mathcal{F}(D, f)$=`A<c1>` *A.*

Our framework does not require the underlying type system of the programming language to be expressed in terms of the functions $\mathcal{F}$ and $\mathcal{M}$. Nevertheless, the underlying type system could *be expressed in terms of these functions. For example, for field access, we would have the underlying type system rule:*

$$\frac{\Gamma \vdash e : \mathbb{r}\ c \qquad \mathcal{F}(c, f) = \mathbb{r}'\ c'}{\Gamma \vdash e.f : (\mathbb{r} \triangleright \mathbb{r}')\ c'}$$

where we define $\mathcal{R}$, the owner parameter extraction function so that it extracts all owner parameters out of a context sequence, i.e., *$\mathcal{R}(top) = \mathcal{R}(bot) = \mathcal{R}(this) = \epsilon$, $\mathcal{R}(X) = X$, $\mathcal{R}(\mathrm{K}\,; rep) = \mathcal{R}(\mathrm{K})$, and where $\mathcal{R}(\mathrm{K}, \overline{\mathrm{K}}) = \mathcal{R}(\mathrm{K}), \mathcal{R}(\overline{\mathrm{K}})$, and where the formal parameters of a class are defined through $OP(c) = \overline{X}$ iff class c has formal owner parameters $\overline{X}$, and where we define the region adaptation operator $\triangleright$ as follows:*

$$c\langle\overline{\mathrm{K}}\rangle \triangleright c'\langle\overline{\mathrm{K}'}\rangle = \begin{cases} c'\langle\overline{\mathrm{K}'}\rangle & \text{if } \mathcal{R}(\overline{\mathrm{K}'}) = \epsilon \\ c'\langle\overline{[\overline{\mathrm{K}}/\overline{X''}]\mathrm{K}'}\rangle & \text{if } c\langle\overline{X}\rangle \lhd c''\langle\overline{X''}\rangle \\ & \text{ and } \mathcal{R}(\overline{\mathrm{K}'}) \subseteq OP(c'') \\ \bot & \text{otherwise.} \end{cases}$$

For example `D<o3>` $\triangleright$ `A<c1>` = `A<o3>`.

We define owner extraction function $\mathcal{O}$ as follows

$$\mathcal{O}_{\mathbb{r},c} = \begin{cases} \mathrm{K}_1, & \text{if } \mathbb{r} = c\langle\overline{\mathrm{K}}\rangle \\ X_1, & \text{if } \mathbb{r} = \mathsf{this},\ c\langle\overline{X}\rangle \lhd \_ \\ \bot & \text{otherwise} \end{cases}$$

These functions are used to describe the Oval' verification technique, as shown in Fig. 10.

Lemma 19. *Oval' is well-structured.*

Proof. We use the shorthand $\mathsf{I} = \mathsf{I}(c, m)$, $\mathsf{E} = \mathsf{E}(c, m)$, $\mathsf{I}' = \mathsf{I}(c', m')$ and $\mathsf{E}' = \mathsf{E}(c', m')$ where we recall that they all come from the domain of L. We also use the following Lemmas:

Lemma 20. $\mathrm{K} \prec \mathrm{K}' \Rightarrow \begin{cases} \mathrm{K}\,; rep^* \subseteq \mathrm{K}'\,; rep^* \\ \mathrm{K}'\,; own^* \subseteq \mathrm{K}\,; own^* \\ \mathrm{K}\,; rep^* \cap \mathrm{K}'\,; own^* = \emptyset \end{cases}$

Lemma 21. *If* $\mathbb{r}; \mathrm{L} \neq \bot$ *then* $\mathbb{r}; \mathrm{L} = \mathbb{r} \triangleright \mathrm{L}$

Lemma 22. $this\,; rep^* \cap this\,; own^* = this$

Lemma 23. $\mathrm{K} \prec this \Rightarrow \mathrm{K}\,; rep^* \subseteq (this\,; rep^* \setminus this)$

(S1): We need to show

$$\mathbb{r} \triangleright \mathsf{I}'\,; \mathsf{rep}^* \setminus (\mathsf{I}\,; \mathsf{rep}^* \setminus \mathsf{E}\,; \mathsf{own}^*) \subseteq \mathsf{emp} \tag{108}$$

If $\mathbb{r} \sqsubseteq \mathbb{C}_{c,m,c',m'}$ then by Fig. 10 we know

$$\mathsf{SC}(\mathsf{I}, \mathsf{E}, \mathbb{r}\,; \mathsf{I}', \mathbb{r}\,; \mathsf{E}', \mathcal{O}_{\mathbb{r},c}) \tag{109}$$

and from (109) and Def. **??** we obtain two subcases

$\mathsf{I} \prec \mathsf{E}$: From this subcase's clause, i.e., $\mathsf{I} \prec \mathsf{E}$, and Def. **??** we also know

$$\mathbb{r}; \mathsf{I}' \preceq \mathsf{I} \tag{110}$$

and thus, since the ordering $\preceq$ is not defined for $\bot$ values, we conclude

$$\mathbb{r}; \mathsf{I}' \neq \bot \tag{111}$$

From the subcase clause, $\mathsf{I} \prec \mathsf{E}$, and Lemma 20 we obtain

$$\mathsf{I}\,; \mathsf{rep}^* \setminus \mathsf{E}\,; \mathsf{own}^* = \mathsf{I}\,; \mathsf{rep}^*$$

[5] Note, that we had erroneously omitted the latter requirement in [7].

and thus from (108) we get

$$\mathbb{r} \rhd \mathsf{I}' \,;\mathsf{rep}^* \setminus \mathsf{I}\,;\mathsf{rep}^* \subseteq \mathsf{emp} \tag{112}$$

From (111) and Lemma 21 we can rewrite (110) as $\mathbb{r} \rhd \mathsf{I}' \preceq \mathsf{I}$ and by Lemma 20 we obtain

$$\mathbb{r} \rhd \mathsf{I}'\,;\mathsf{rep}^* \subseteq \mathsf{I}\,;\mathsf{rep}^*$$

and thus $\mathbb{r} \rhd \mathsf{I}'\,;\mathsf{rep}^* \setminus \mathsf{I}\,;\mathsf{rep}^* = \mathsf{emp}$ satisfying (112).

$\mathsf{I} = \mathsf{E} = \mathsf{this}$: Similar to the case before, from $\mathsf{I} = \mathsf{E}$, Def. **??** and Lemma 21 we get

$$\mathbb{r} \rhd \mathsf{I}' \prec \mathsf{this} \tag{113}$$

From the subcase clause, $\mathsf{I} = \mathsf{E} = \mathsf{this}$, and Lemma 22 we can derive

$$\mathsf{this}\,;\mathsf{rep}^* \setminus \mathsf{this}\,;\mathsf{own}^* = \mathsf{this}\,;\mathsf{rep}^* \setminus \mathsf{this}$$

and thus by (108) we obtain

$$\mathbb{r} \rhd \mathsf{I}'\,;\mathsf{rep}^* \setminus (\mathsf{this}\,;\mathsf{rep}^* \setminus \mathsf{this}) \subseteq \mathsf{emp}$$

Finally, from (113) and Lemma 23 we derive that

$$\mathbb{r} \rhd \mathsf{I}'\,;\mathsf{rep}^* \setminus (\mathsf{this}\,;\mathsf{rep}^* \setminus \mathsf{this}) = \mathsf{emp}$$

which satisfies the above.

(S2): Immediate from (107), Lemma 20 and Lemma 22.

(S3): We recall that

$$\mathbb{C}_{c,m,c',m'} = \sqcup \mathbb{r}_i \text{ such that } \mathsf{SC}(\mathsf{I}, \mathsf{E}, \mathbb{r}_i\,;\mathsf{I}', \mathbb{r}_i\,;\mathsf{E}', \mathcal{O}_{\mathbb{r}_i,c})$$

We here prove that for every such $\mathbb{r}_i$

$$\mathbb{r}_i \rhd (\mathbb{V}_{c',m'} \setminus \mathbb{E}_{c',m'}) \subseteq \mathbb{V}_{c,m}$$

From which (S3) follows from the monotonicity of $\rhd$. For this proof we find it convenient to distribute the adaptation in (S3) and show

$$\mathbb{r}_i \rhd \mathbb{V}_{c',m'} \setminus \mathbb{r}_i \rhd \mathbb{E}_{c',m'} \subseteq \mathbb{V}_{c,m} \tag{114}$$

From the subcontract definition, we have two subcases:

$\mathbb{r}_i\,;\mathsf{I}' \prec \mathbb{r}_i\,;\mathsf{E}'$: From the subcase clause $\mathbb{r}_i\,;\mathsf{I}' \prec \mathbb{r}_i\,;\mathsf{E}'$, Lemma 21 and Lemma 20 we deduce

$$\begin{aligned}\mathbb{r}_i \rhd \mathbb{V}_{c',m'} &\setminus \mathbb{r}_i \rhd \mathbb{E}_{c',m'} \\ &= \mathbb{r}_i \rhd \mathbb{V}_{c',m'} \setminus \quad \mathbb{r}_i \rhd \mathsf{emp} \\ &= \mathbb{r}_i \rhd \mathsf{E}'\,;\mathsf{own}^* \setminus \mathsf{emp} \\ &= \mathbb{r}_i \rhd \mathsf{E}'\,;\mathsf{own}^*\end{aligned}$$

and thus by (114) it suffices to prove

$$\mathbb{r}_i \rhd \mathsf{E}'\,;\mathsf{own}^* \subset \mathsf{E}\,;\mathsf{own}^* \tag{115}$$

From the subcase and Def. **??** we also know $\mathsf{E} \preceq \mathbb{r}_i\,;\mathsf{E}'$, thus by Lemma 21 we have $\mathsf{E} \preceq \mathbb{r}_i \rhd \mathsf{E}'$ and hence by Lemma 20 we obtain (115) as required.

$\mathbb{r}_i\,;\mathsf{I}' = \mathbb{r}_i\,;\mathsf{E}' = \mathsf{this}$: From Lemma 22, Lemma 21 and (114) we obtain

$$\mathbb{r}_i \rhd \mathsf{E}'\,;\mathsf{own}^* \setminus \mathsf{this} \subseteq \mathsf{E}\,;\mathsf{own}^* \tag{116}$$

From the subcase and Def. **??** we also know $\mathsf{E} \preceq \mathcal{O}_{\mathbb{r}_i,c}$ which proves (116) as required.

(S4): By (107) we have two subcases to consider:

$\mathsf{I} \prec \mathsf{E}$: From 10 we know $\mathbb{U}_{c,m,c'} = \mathsf{emp}$ thus we have the proof

$$\begin{aligned}\mathsf{emp} \rhd (\mathsf{this}\,;\mathsf{own}^*) &\subseteq \mathsf{E}\,;\mathsf{own}^* \\ \mathsf{emp} &\subseteq \mathsf{E}\,;\mathsf{own}^*\end{aligned}$$

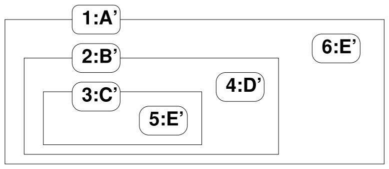


Figure 14. Heap h_0, with objects at addresses 1–6 belonging to indicated classes. Objects atop a box own those inside it. Assume that A' is a subclass of A and analogously for the other classes.

$\mathsf{I} = \mathsf{E} = \mathsf{this}$: From 10 we know $\mathbb{U}_{c,m,c'} = \mathsf{this}$ thus we have the proof

$$\begin{aligned}\mathsf{this} \rhd (\mathsf{this}\,;\mathsf{own}^*) &\subseteq \mathsf{this}\,;\mathsf{own}^* \\ \mathsf{this}\,;\mathsf{own}^* &\subseteq \mathsf{this}\,;\mathsf{own}^*\end{aligned}$$

(S5): Suppose $c' \leq c$. Recall that our amended requirements for method overriding are as follows:

$$\mathsf{I}' \preceq \mathsf{I} \preceq \mathsf{E} \preceq \mathsf{E}' \qquad \mathsf{I} = \mathsf{E}' \Rightarrow \mathsf{I}' = \mathsf{E}'$$

Therefore, it is immediate from Lemma 20 that:

$$\mathbb{X}_{c',m} \subseteq \mathbb{X}_{c,m} \tag{117}$$

holds (i.e., the first part of (S5)), and also that

$$\mathbb{V}_{c',m} \subseteq \mathbb{V}_{c,m} \tag{118}$$

It remains to show that

$$\mathbb{V}_{c',m} \setminus \mathbb{E}_{c',m} \subseteq \mathbb{V}_{c,m} \setminus \mathbb{E}_{c,m} \tag{119}$$

We first eliminate various cases. Firstly, if $\mathsf{I} \neq \mathsf{E}$, then $\mathbb{E}_{c,m} = \mathsf{emp}$, and so (119) follows immediately from (118). Therefore, we consider the remaining case $\mathsf{I} = \mathsf{E} = \mathsf{this}$, for which we know $\mathbb{E}_{c,m} = \mathsf{this}$, and so

$$\mathbb{V}_{c,m} \setminus \mathbb{E}_{c,m} = \mathbb{V}_{c,m} \setminus \mathsf{this} \tag{120}$$

Next, if $\mathsf{E} \prec \mathsf{E}'$, then (119) follows easily from (120). Therefore we consider the remaining case $\mathsf{E} = \mathsf{E}'$. Then, by our second requirement on overriding, we conclude $\mathsf{I}' = \mathsf{I} = \mathsf{E} = \mathsf{E}' = \mathsf{this}$. Therefore, in this case it follows that $\mathbb{V}_{c,m} = \mathbb{V}_{c',m}$ and $\mathbb{E}_{c,m} = \mathbb{E}_{c',m}$, and (119) is trivially satisfied.

□

6.2.3 Comparisons

We first illustrate differences between the techniques for structured heaps using the heap h_0 in Fig. 14. Fig. 15 shows the values of the components of the three techniques for class C and object 3.

OT and VT require knowledge of the class at which on object is owned; this information is shown in the last row of Fig. 15. For Oval, the methods have I and E as given in the last row.

Invariant Restrictions ($\mathbb{D}$). Both OT and Oval support multi-object invariants by permitting the invariant of an object o to depend on fields of o and of objects (transitively) owned by o. However, OT requires that fields of o are declared in the same class as the invariant to address the subclass challenge. For instance, $\mathbb{D}$ for OT does not include (3,C'), whereas $\mathbb{D}$ for Oval does.

In addition, VT allows dependencies on peers (therefore, $\mathbb{D}$ includes (4,D)) and thus can handle multi-object structures that are not organised hierarchically.

Program Restrictions ($\mathbb{U}$ and $\mathbb{C}$). In OT and Oval, an object may only modify its own fields, whereas VT also allows modifications of peers; thus, object 4 is part of $\mathbb{U}$ for VT. In Oval, an object may

	Müller et al. (OT)	Müller et al. (VT)	Lu et al.
1. $[\![\mathbb{X}_{\mathsf{C,m}}]\!]_{h_0,3}$	$\{$ (4, D), (4, D'), (3, C), (3, C'), (5, E), (5, E') $\}$	$\{$ (4, D), (4, D'), (3, C), (3, C'), (5, E), (5, E') $\}$	$\{$ (3, C), (3, C'), (5, E), (5, E') $\}$
2. $[\![\mathbb{V}_{\mathsf{C,m}}]\!]_{h_0,3}$	$\{$ (3, C), (2, B), (1, A') $\}$	$\{$ (3, C), (2, B), (1, A'), (4, D) $\}$	$\{$ (2, B), (2, B'), (1, A), (1, A') $\}$
3. $[\![\mathbb{D}_{\mathsf{C}}]\!]_{h_0,3}$	$\{$ (3, C), (2, B), (1, A') $\}$	$\{$ (3, C), (2, B), (1, A'), (4, D) $\}$	$\{$ (3, C), (3, C'), (2, B), (2, B'), (1, A), (1, A') $\}$
4. $[\![\mathbb{B}_{\mathsf{C,m},\mathbb{r}}]\!]_{h_0,3}$	$\emptyset$ if $\mathbb{r} = \mathsf{rep}\langle\mathsf{C}\rangle$ $\{$ (3, C) $\}$ if $\mathbb{r} = \mathsf{peer}$	$\emptyset$ if $\mathbb{r} = \mathsf{rep}\langle\mathsf{C}\rangle$ $\{$ (3, C), (4, D) $\}$ if $\mathbb{r} = \mathsf{peer}$	$\emptyset$
5a. $[\![\mathbb{E}_{\mathsf{C,m}}]\!]_{h_0,3}$	$\{$ (3, C) $\}$	$\{$ (3, C), (4, D) $\}$	$\emptyset$
5b. $[\![\mathbb{E}_{\mathsf{C,m1}}]\!]_{h_0,3}$	$\{$ (3, C) $\}$	$\{$ (3, C), (4, D) $\}$	$\{$ (3, C), (3,C') $\}$
6a. $[\![\mathbb{U}_{\mathsf{C,m,Objct}}]\!]_{h_0,3}$	$\{$ 3 $\}$	$\{$ 3, 4 $\}$	$\emptyset$
6b. $[\![\mathbb{U}_{\mathsf{C,m1,Objct}}]\!]_{h_0,3}$	$\{$ 3 $\}$	$\{$ 3, 4 $\}$	$\{$ 3 $\}$
7. $[\![\mathbb{C}_{\mathsf{C,m,Objct,m2}}]\!]_{h_0,3}$	$\{$ 3, 4, 5 $\}$	$\{$ 3, 4, 5 $\}$	$\{$ 1, 2, 3, 4, 5, 6 $\}$
assuming that	C::m not pure $ownr(h_0, 5) = 3, \mathsf{C'}$, $ownr(h_0, 3) = 2, \mathsf{B}$, $ownr(h_0, 4) = 2, \mathsf{B'}$, $ownr(h_0, 2) = 1, \mathsf{A}$	C::m not pure $ownr(h_0, 5) = 3, \mathsf{C'}$, $ownr(h_0, 3) = 2, \mathsf{B}$, $ownr(h_0, 4) = 2, \mathsf{B'}$, $ownr(h_0, 2) = 1, \mathsf{A}$ vis(C, D), $\neg$vis(C, D')	I(C, m) = this E(C, m) = X, and X maps to 2 I(C, m1) = E(C, m1) = this I(Obj, m2) = bot E(Obj, m2) = top

Figure 15. Comparison of techniques for structured heaps; differences are highlighted in grey.

only modify its own fields if the I, E annotations are this; this is why $\mathbb{U}$ is empty for m but contains 3 for m1.

Method calls in OT and VT are restricted to the peers and reps of an object; thus, a call on a rep object o cannot call back into one of o's (transitive) owners, whose invariants might not hold.

In Oval, the receiver of a method call may be *anywhere* within the owners of the current receiver, provided that the I and E annotations of the called method satisfy the subcontract requirement. Therefore, $\mathbb{C}$ for Oval includes for instance object 2, which is not permitted in OT and VT.

Proof Obligations ($\mathbb{B}$ and $\mathbb{E}$). Since OT uses rather restricted invariants, it has a small vulnerable set $\mathbb{V}$ and, thus, few proof obligations. The dependencies on peers permitted by VT lead to a larger vulnerable set and more proof obligations. For instance, (4,D) is part of the vulnerable set $\mathbb{V}$ (because executions on 3 might break 4's D-invariant). Hence, of the proof obligations $\mathbb{B}$ and $\mathbb{E}$.

Oval imposes end-of-body proof obligation only when I and E are the same (i.e. m1). Since Oval permits invariants to depend on inherited fields, it requires proof obligations for subclass invariants. For instance, (3,C') is part of $\mathbb{E}$ for m1. OT and VT disallow such dependencies and their proof obligations do not include (3,C'). This restriction is important for modularity.[6] Oval never impose proof obligations before method calls ($\mathbb{B}$ is empty), and prevents potentially dangerous call-backs through the subcontract requirement.

Implications for our example As an alternative comparison, we consider the application of the three techniques to our running example (Fig. 2).

1. *Invariant semantics:* In OT and VT, the invariants expected at the beginning of withdraw are I1, I2, and I3 for the receiver, as well as I5 for the associated DebitCard (which is a **peer**). For withdraw in Oval, I=this, therefore the expected invariants are I1, I2, and I3 for the receiver.
2. *Invariant restrictions:* Invariants I2 and I5 are illegal in OT and Oval, while they are legal in VT (which allows invariants to depend on the fields of **peer**s). Conversely, I3 is illegal in OT and VT (it mentions a field from a superclass), while it is legal in Oval.
3. *Proof obligations:* In OT, before the call to **this**.sendReport() and at the end of the body of withdraw, we have to establish I1 and I2 for the receiver. In addition to these, in VT we have to establish I5 for the debit card. In Oval, the same invariants as for OT have to be proven, but only at the end of the method because call-backs are handled through subcontracting. In addition, I3 is required.[7] In all three techniques, withdraw is permitted to leave the invariant I4 of the owning Person object broken. It has to be re-established by the calling Person method.
4. *Program restrictions:* OT and VT forbid the call holder. notify () (**rep**s cannot call their owners), while Oval allows it. On the other hand, if method sendReport required an invariant of its receiver (for instance, to ensure that holder is non-null), then Oval would prevent method withdraw from calling it, even though the invariants of the receiver might hold at the time of the call. The proof obligations before calls in OT and VT would make such a call legal.

6.3 Soundness of Verification Techniques

Instead of proving soundness for every single verification technique discussed in this section, Theorem 6 reduces this complex task to merely checking that the seven components of every instantiations satisfy the five (fairly simple) well-structured conditions of Def. 5. Assuming that the underlying type system is sound, once we show well-structuredness for a technique, verification technique soundness (Def. **??**) follows.

Lemma 24 (Type System Soundness for Universes)**.** *The Universes Type System satisfies Def. 32.*

Proof. The typing rules together with the soundness proof for the Universes type system has already been given in [**?**] bar the rules for the (novel) construct e prv $\mathbb{p}$ and the exceptions verfExc and fatalExc. The typing of the proof annotation construct however

[6] The Oval developers plan to solve this modularity problem by requiring that any inherited method has to be re-verified in the subclass [22].

[7] This means that verification of a class requires knowledge of a subclass. The Oval developers plan to solve this modularity problem by requiring that any inherited method has to be re-verified in the subclass [22].

depends exclusively on the typing of the subexpression e; typically this construct would be typechecked using a rule such as the one shown below. Also, the type system should typecheck exceptions related to the verification technique as shown below.

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash e \;\mathsf{prv}\; \mathbb{p} : t} \qquad \frac{}{\begin{array}{c}\Gamma \vdash \mathsf{verfExc} : t \\ \Gamma \vdash \mathsf{fatalExc} : t\end{array}}$$

With these additions, it is not hard to check that the type system satisfies the requirements set out by Def. 32. □

Lemma 25 (Type System Soundness for Oval'). *The Oval' Type System satisfies Def. 32.*

Proof. From [23]. □

Corollary 26. *The verification techniques by Poetzsch-Heffter, by Huizing & Kuiper, by Leavens & Müller, by Müller* et al. *(OT), by Müller* et al. *(VT), and Oval' are sound.*

Proof. Immediate from Theorem 6, Lemmas 24 and 25, and Lemmas 14, 15, 19. □

These proofs confirm soundness claims from the literature. We found that the semi-formal arguments supporting the original soundness claims at times missed crucial steps. For instance, the soundness proofs for OT and VT [28] do not mention any condition relating to (S3) of Def. 5; in our formal proof, (S3) was vital to determine what invariants still hold after a method returns.

***Unsoundness of* Oval.** The original Oval proposal [23] is unsound because it requires subcontracting for method overriding. As we said in the previous section, subcontracting corresponds to our (S1) and (S3). This gives, for $c' <: c$, the requirements that $\mathbb{X}_{c',m'} \subseteq \mathbb{X}_{c,m} \backslash \mathbb{V}_{c,m}$, and $\mathbb{V}_{c',m'} \backslash \mathbb{E}_{c',m'} \subseteq \mathbb{V}_{c,m}$, which do not imply (S5). We were alerted by this discrepancy, and using only the $\mathbb{X}$, $\mathbb{E}$ and $\mathbb{V}$ components (no type system properties, nor any other component), we constructed the following counterexample.

```
class D[o] {
   C1<this> c = new C2<this>();
   void m() <this,o> { c.mm() }
}

class C1[o]{
   void mm() <this,this> {...}
}

class C2[o] extends C1<o> {
   void mm() <bot,this> {...}
}
```

The call c.mm() is checked using the contract of C1::mm; it expects the callee to re-establish the invariant of the receiver (c), and is type correct. However, the body of C2::mm may break the receiver's invariants, but has no proof obligations ($\mathbb{E}_{\mathsf{C2},\mathsf{mm}} = \mathsf{emp}$). Thus, the call c.mm() might break the invariants of c, thus breaking the contract of m. The reason for this problem is, that the—initially appealing—parallel between subcontracting and method overriding does not hold. The authors confirmed our findings [22].

7. Related Work

Object invariants trace back to Hoare's implementation invariants [12] and monitor invariants [13]. They were popularised in object-oriented programming by Meyer [24]. Their work, as well as other early work on object invariants [20, 21] did not address the three challenges described in the introduction. Since they were not formalised, it is difficult to understand the exact requirements and soundness arguments (see [28] for a discussion). However, once the requirements are clear, a formalisation within our framework seems straightforward.

The idea of regions and properties is inspired from type and effects systems [34], which have been extremely widely applied, e.g., to support race-free programs and atomicity [10].

The verification techniques based on the Boogie methodology [1, 3, 18, 19] do not use a visible state semantics. Instead, each method specifies in its precondition which invariants it requires. Extending our framework to Spec# requires two changes. First, even though Spec# permits methods to specify explicitly which invariants they require, the default is to require the invariants of its arguments and all their peer objects. These defaults can be modelled in our framework by allowing method-specific properties $\mathbb{X}$. Second, Spec# checks invariants at the end of expose blocks instead of the end of method bodies. Expose blocks can easily be added to our formalism.

In separation logic [15, 33], object invariants are generally not as important as in other verification techniques. Instead, predicates specifying consistency criteria can be assumed/proven at *any* point in a program [29]. Abstract predicate families [30] allow one to do so without violating abstraction and information hiding. Parkinson and Bierman [31] show how to address the subclass challenge with abstract predicates. Their work as well as Chin et al.'s [4] allow programmers to specify which invariants a method expects and preserves, and do not require subclasses to maintain inherited invariants. The general predicates of separation logic provide more flexibility than can be expressed by our framework.

We know of only one technique based on visible states that cannot be directly expressed in our framework: Middelkoop et al. [26] use proof obligations that refer to the heap of the pre-state of a method execution. To formalise this technique, we have to generalise our proof obligations to take two properties; one for the pre-state heap and one for the post-state heap. Since this generality is not needed for any of the other techniques, we omitted a formal treatment in this paper.

Some verification techniques exclude the pre- and post-states of so-called helper methods from the visible states [16, 17]. Helper methods can easily be expressed in our framework by choosing different parameters for helper and non-helper methods. For instance in JML, $\mathbb{X}$, $\mathbb{B}$, and $\mathbb{E}$ are empty for helper methods, because they neither assume nor have to preserve any invariants.

Once established, strong invariants [11] hold throughout program execution. They are especially useful to reason about concurrency and security properties. Our framework can model strong invariants, essentially by preventing them from occurring in $\mathbb{V}$.

Existing techniques for visible state invariants have only limited support for object initialisation. Constructors are prevented from calling methods because the callee method in general requires all invariants to hold, but the invariant of the new object is not yet established. Fähndrich and Xia developed delayed types [9] to control call-backs into objects that are being initialised. Delayed types support strong invariants. Modelling these in our framework is future work.

8. Conclusions

We presented a framework that describes verification techniques for object invariants in terms of seven parameters and separates verification concerns from those of the underlying type system. Our formalism is parametric wrt. the type system of the programming language and the language used to describe and to prove assumptions. We illustrated the generality of our framework by instantiating it to describe three existing verification techniques. We identified sufficient conditions on the framework parameters that guarantee soundness, and we proved a universal soundness theorem. Our unified framework offers the following important advantages:

1. It allows a simpler understanding and separation of verification concerns. In particular, most of the aspects in which verification techniques differ are distilled in terms of subsets of the parameters of our framework.
2. It facilitates comparisons since relationships between parameters can be expressed at an abstract level (e.g., criteria for well-structuredness in Def. 5), and the interpretations of regions and properties as sets allow formal comparisons of techniques in terms of set operations.
3. It expedites the soundness analysis of verification techniques, since checking the soundness conditions of Def. 5 is significantly simpler than developing soundness proofs from scratch.
4. It captures the design space of *sound* visible states based verification techniques.

We are currently using our framework in developing verification techniques for static methods, and plan to use it to develop further, more flexible, techniques.

Acknowledgments

This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project. Müller's work was partly carried out at ETH Zurich.

We want to thank Rustan Leino for suggestions on notation, Ronald Middlekoop for discussion of related work, the Imperial College as well as the Cambridge University programming languages group for suggestions on presentation, and Yi Lu and John Potter for many discussions about Oval.

References

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

[2] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS*, LNCS, pages 49–69. Springer-Verlag, 2005.

[3] M. Barnett and D. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *MPC*, volume 3125 of *LNCS*, pages 54–84. Springer, 2004.

[4] W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Enhancing modular OO verification with separation logic. In *POPL*, pages 87–99. ACM Press, 2008.

[5] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, volume 33(10), pages 48–64. ACM Press, 1998.

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

[7] S. Drossopoulou, A. Francalanza, and P. Müller. A unified framework for verification techniques for object invariants. In *FOOL*, 2008.

[8] S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summmers. A unified framework for verification techniques for object invariants — ecoop'08 full paper. Available from `http://www.doc.ic.ac.uk/~ajs300m/papers/frameworkFull.pdf`, 2008.

[9] M. Fähndrich and S. Xia. Establishing object invariants with delayed types. In *OOPSLA*, pages 337–350. ACM Press, 2007.

[10] C. Flanagan and S. Qadeer. A Type and Effect System for Atomicity. In *PLDI*, pages 338–349. ACM Press, 2003.

[11] R. Hähnle and W. Mostowski. Verification of safety properties in the presence of transactions. In *CASSIS*, volume 3362 of *LNCS*, pages 151–171, 2005.

[12] C. A. R. Hoare. Proofs of correctness of data representation. *Acta Informatica*, 1:271–281, 1972.

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

[14] K. Huizing and R. Kuiper. Verification of object-oriented programs using class invariants. In *FASE*, volume 1783 of *LNCS*, pages 208–221. Springer-Verlag, 2000.

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

[16] G. T. Leavens and P. Müller. Information hiding and visibility in interface specifications. In *ICSE*, pages 385–395. IEEE, 2007.

[17] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. JML Reference Manual. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`, February 2007.

[18] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *ECOOP*, volume 3086 of *LNCS*, pages 491–516. Springer-Verlag, 2004.

[19] K. R. M. Leino and W. Schulte. Using history invariants to verify observers. In *ESOP*, volume 4421 of *LNCS*, pages 316–330. Springer-Verlag, 2007.

[20] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.

[21] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM ToPLAS*, 16(6):1811–1841, 1994.

[22] Y. Lu and J. Potter. Soundness of Oval. Priv. Commun., June 2007.

[23] Y. Lu, J. Potter, and J. Xue. Object Invariants and Effects. In *ECOOP*, volume 4609 of *LNCS*, pages 202–226. Springer-Verlag, 2007.

[24] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.

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

[26] R. Middelkoop, C. Huizing, R. Kuiper, and E. J. Luit. Invariants for non-hierarchical object structures. *Electr. Notes Theor. Comput. Sci.*, 195:211–229, 2008.

[27] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer, 2002.

[28] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, 2006.

[29] M. Parkinson. Class invariants: the end of the road? In *International Workshop on Aliasing, Confinement and Ownership*, 2007.

[30] M. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL*, pages 247–258. ACM Press, 2005.

[31] M. Parkinson and G. Bierman. Separation logic, abstraction and inheritance. In *POPL*, pages 75–86. ACM Press, 2008.

[32] A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, 1997.

[33] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

[34] J. P. Talpin and P. Jouvelot. The Type and Effect Discipline. In *LICS*, pages 162–173. IEEE Computer Society, 1992.

A. Appendix

A.1 Language Definitions

Definition 27. *A runtime structure is a tuple*

$$\text{RSTRUCT} = (\text{HP}, \text{ADR}, \simeq, \preceq, \text{dom}, \text{cls}, \text{fld}, \text{upd}, \text{new})$$

where HP, *and* ADR *are sets, and where*

$$\begin{array}{lll} \simeq & \subseteq \textsc{Hp}\times\textsc{Hp} & \preceq\ \subseteq\ \textsc{Hp}\times\textsc{Hp} \\ \text{dom} & :\ \textsc{Hp}\to\mathcal{P}(\textsc{Adr}) & \\ \text{cls} & :\ \textsc{Hp}\times\textsc{Adr}\rightharpoonup\textsc{Cls} & \\ \text{fld} & :\ \textsc{Hp}\times\textsc{Adr}\times\textsc{Fld}\rightharpoonup\textsc{Val} & \\ \text{upd} & :\ \textsc{Hp}\times\textsc{Adr}\times\textsc{Fld}\times\textsc{Val}\to\textsc{Hp} & \\ \text{new} & :\ \textsc{Hp}\times\textsc{Adr}\times\textsc{Typ}\to\textsc{Hp}\times\textsc{Adr} & \end{array}$$

where $\textsc{Val} = \textsc{Adr}\cup\{\textit{null}\}$ *for some element* $\textit{null}\notin\textsc{Adr}$. *For all* $h\in\textsc{Hp},\ \iota,\iota'\in\textsc{Adr},\ v\in\textsc{Val}$, *we require:*

(H1) $\iota\in\text{dom}(h)\ \Rightarrow\exists c.\text{cls}(h,\iota)=c$

(H2) $h\simeq h'\ \Rightarrow\begin{cases}\text{dom}(h)=\text{dom}(h'),\\ \text{cls}(h,\iota)=\text{cls}(h',\iota)\end{cases}$

(H3) $h\preceq h'\ \Rightarrow\begin{cases}\text{dom}(h)\subseteq\text{dom}(h'),\\ \forall\iota\in\text{dom}(h).\\ \quad\text{cls}(h,\iota)=\text{cls}(h',\iota)\end{cases}$

(H4) $\text{upd}(h,\iota,f,v)=h'\ \Rightarrow\begin{cases}h\simeq h'\\ \text{fld}(h',\iota,f)=v\\ \iota\neq\iota'\ \textit{or}\ f\neq f'\Rightarrow\\ \quad\text{fld}(h',\iota',f')=\text{fld}(h,\iota',f')\end{cases}$

(H5) $\text{new}(h,\iota,t)=h',\iota'\ \Rightarrow\begin{cases}h\preceq h'\\ \iota'\in\text{dom}(h')\backslash\text{dom}(h)\end{cases}$

Definition 28. *An* region/region structure *is a tuple*
$\textsc{AStruct}\ =\ (\mathbf{R},\mathbf{P},\triangleright)$
where $\mathbf{R}$ *and* $\mathbf{P}$ *are sets, and* $\triangleright$ *is an operation with signature:*
$\triangleright\ :\ \mathbf{R}\times\mathbf{P}\to\mathbf{P}$

Definition 29. $E[\cdot]$ *and* $F[\cdot]$ *are defined as follows:*

$$\begin{array}{rl} E[\cdot] & ::= [\cdot]\ \mid\ E[\cdot].f\ \mid\ E[\cdot].f := e\ \mid\ \iota.f := E[\cdot]\ \mid\ E[\cdot].m(e) \\ & \mid\ \iota.m(E[\cdot])\ \mid\ E[\cdot]\ \textit{prv}\,\mathbb{p}\ \mid\ \textit{ret}\,E[\cdot] \\ F[\cdot] & ::= [\cdot]\ \mid\ F[\cdot].f\ \mid\ F[\cdot].f := e\ \mid\ \iota.f := F[\cdot]\ \mid\ F[\cdot].m(e) \\ & \mid\ \iota.m(F[\cdot])\ \mid\ F[\cdot]\ \textit{prv}\,\mathbb{p}\ \mid\ \sigma\cdot F[\cdot]\ \mid\ \textit{call}\,F[\cdot]\ \mid\ \textit{ret}\,F[\cdot] \end{array}$$

Definition 30. *A programming language is a tuple*
$\textsc{PL}\ =\ (\textsc{Prg},\textsc{RStruct},\textsc{AStruct})$
where $\textsc{Prg}$ *is a set where every* $\mathrm{P}\in\textsc{Prg}$ *is a tuple*

$$\mathrm{P}\ =\ \left(\begin{array}{ll}\mathcal{F},\mathcal{M},\mathcal{B},<: & \text{(class definitions)}\\ \sqsubseteq,[\![\cdot]\!] & \text{(inclusion and projections)}\\ \models,\vdash & \text{(invariant and type satisfaction)}\end{array}\right)$$

with signatures:

$$\begin{array}{ll} \mathcal{F} & :\ \textsc{Cls}\times\textsc{Fld}\rightharpoonup\textsc{Typ}\times\textsc{Cls} \\ \mathcal{M} & :\ \textsc{Cls}\times\textsc{Mthd}\rightharpoonup\textsc{Typ}\times\textsc{Typ} \\ \mathcal{B} & :\ \textsc{Cls}\times\textsc{Mthd}\rightharpoonup\textsc{Expr}\times\textsc{Cls} \\ <: & \subseteq\ \textsc{Cls}\times\textsc{Cls}\ \cup\ \textsc{Typ}\times\textsc{Typ} \\ \sqsubseteq & \subseteq\ \mathbf{R}\times\mathbf{R} \\ [\![\cdot]\!] & :\ \mathbf{R}\times\textsc{Hp}\times\textsc{Adr}\to\mathcal{P}(\textsc{Adr}) \\ [\![\cdot]\!] & :\ \mathbf{P}\times\textsc{Hp}\times\textsc{Adr}\to\mathcal{P}(\textsc{Adr}\times\textsc{Cls}) \\ \models & \subseteq\ \textsc{Hp}\times\textsc{Adr}\times\textsc{Cls} \\ \vdash & \subseteq\ (\textsc{Env}\times\textsc{Expr}\ \cup\ \textsc{Hp}\times\textsc{Stk}\times\textsc{RExpr})\times\textsc{Typ} \end{array}$$

where every $\mathrm{P}\in\textsc{Prg}$ *must satisfy the constraints:*

(P1) $\mathcal{F}(c,f)=t,c'\ \Rightarrow c<:c'$
(P2) $\mathcal{B}(c,m)=e,c'\ \Rightarrow c<:c'$
(P3) $\mathcal{F}(\text{cls}(h,\iota),f)=t,\_\ \Rightarrow\exists v.\text{fld}(h,\iota,f)=v$
(P4) $\mathbb{r}_1\sqsubseteq\mathbb{r}_2\ \Rightarrow[\![\mathbb{r}_1]\!]_{h,\iota}\subseteq[\![\mathbb{r}_2]\!]_{h,\iota}$
(P5) $[\![\mathbb{r}\triangleright\mathbb{p}]\!]_{h,\iota}\ =\bigcup_{\iota'\in[\![\mathbb{r}]\!]_{h,\iota}}[\![\mathbb{p}]\!]_{h,\iota'}$
(P6) $[\![\mathbb{r}]\!]_{h,\iota}\ \subseteq\ \text{dom}(h)$
(P7) $h\preceq h'\ \Rightarrow[\![\mathbb{p}]\!]_{h,\iota}\subseteq[\![\mathbb{p}]\!]_{h',\iota}$
(P8) $\mathbb{r}\,c<:\mathbb{r}'\,c'\ \Rightarrow\ \mathbb{r}\sqsubseteq\mathbb{r}',\ c<:c'$

Definition 31. *For every program, the judgement:*
$\vdash_{\textit{wf}}\ :\ \ (\textsc{Hp}\times\textsc{Stk}\times\textsc{Stk}\times\mathbf{R})\ \cup\ (\textsc{Env}\times\textsc{Hp}\times\textsc{Stk})\ \cup\ \textsc{Prg}$
is defined as:

- $h,\sigma\vdash_{\textit{wf}}\sigma':\mathbb{r}\quad\Leftrightarrow\quad\sigma'=(\iota,\_,\_,\_),\quad h,\sigma\vdash\iota:\mathbb{r}\,\_$
- $\Gamma\vdash_{\textit{wf}}h,\sigma\quad\Leftrightarrow\quad\begin{cases}\exists c,m,t,\iota,v.\\ \quad\Gamma=c,m,t,\ \ \sigma=(\iota,v,c,m),\\ \quad\text{cls}(h,\iota)<:c,\ \ h,\sigma\vdash_{\boldsymbol{r}}v:t\end{cases}$

(vd-null) $$\frac{}{h\vdash_{\mathcal{V}}\sigma\cdot\textsf{null}}$$ (vd-addr) $$\frac{\iota\in\text{dom}(h)}{h\vdash_{\mathcal{V}}\sigma\cdot\iota}$$ (vd-new) $$\frac{}{h\vdash_{\mathcal{V}}\sigma\cdot\textsf{new}\,t}$$

(vd-Var) $$\frac{}{h\vdash_{\mathcal{V}}\sigma\cdot x}$$ (vd-this) $$\frac{}{h\vdash_{\mathcal{V}}\sigma\cdot\textsf{this}}$$ (vd-verEx) $$\frac{}{h\vdash_{\mathcal{V}}F[\textsf{verfExc}]}$$

(vd-ass) $$\frac{\begin{array}{l}h\vdash\sigma\cdot\mathrm{e}_r:\mathbb{r}\,c'\\ \mathcal{F}(c',f)=\_,c\\ \mathbb{r}\sqsubseteq\mathbb{U}_{\sigma,c}\\ h\vdash_{\mathcal{V}}\sigma\cdot\mathrm{e}_r\\ h\vdash_{\mathcal{V}}\sigma\cdot\mathrm{e}'_r\end{array}}{h\vdash_{\mathcal{V}}\sigma\cdot\mathrm{e}_r.f:=\mathrm{e}'_r}$$

(vd-fld) $$\frac{h\vdash_{\mathcal{V}}\sigma\cdot\mathrm{e}_r}{h\vdash_{\mathcal{V}}\sigma\cdot\mathrm{e}_r.f}$$

(vd-end) $$\frac{h\vdash_{\mathcal{V}}\sigma'\cdot v\quad h\models\mathbb{E}_{\sigma'},\ \sigma'}{h\vdash_{\mathcal{V}}\sigma\cdot\sigma'\cdot\textsf{ret}\,v}$$

(vd-call) $$\frac{\begin{array}{l}h\vdash\sigma\cdot\mathrm{e}_r:\mathbb{r}\,c'\\ \mathcal{B}(c',m)=\_,\ c\\ \mathbb{r}\sqsubseteq\mathbb{C}_{\sigma,c,m}\\ h\vdash_{\mathcal{V}}\sigma\cdot\mathrm{e}_r\\ h\vdash_{\mathcal{V}}\sigma\cdot\mathrm{e}'_r\end{array}}{h\vdash_{\mathcal{V}}\sigma\cdot\mathrm{e}_r.m(\mathrm{e}'_r\ \textsf{prv}\,\mathbb{B}_{\sigma,\mathbb{r}})}$$

(vd-call-2) $$\frac{\begin{array}{l}h\vdash\sigma\cdot v:\mathbb{r}\,c'\\ \mathcal{B}(c',m)=\_,\ c\\ h\models\mathbb{B}_{\sigma,\mathbb{r}},\ \sigma\\ \mathbb{r}\sqsubseteq\mathbb{C}_{\sigma,c,m}\\ h\vdash_{\mathcal{V}}\sigma\cdot v\\ h\vdash_{\mathcal{V}}\sigma\cdot v'\end{array}}{h\vdash_{\mathcal{V}}\sigma\cdot v.m(v')}$$

(vd-start) $$\frac{h\vdash_{\mathcal{V}}\sigma'\cdot e}{h\vdash_{\mathcal{V}}\sigma\cdot\sigma'\cdot\textsf{call}\,e\ \textsf{prv}\,\mathbb{E}_{\sigma'}}$$ (vd-frame) $$\frac{h\vdash_{\mathcal{V}}\sigma'\cdot\mathrm{e}_r}{h\vdash_{\mathcal{V}}\sigma\cdot\sigma'\cdot\textsf{ret}\,\mathrm{e}_r\ \textsf{prv}\,\mathbb{E}_{\sigma'}}$$

Figure 16. Well-annotated runtime expressions.

- $\vdash_{\textit{wf}}\mathrm{P}\quad\Leftrightarrow\quad\begin{cases}\text{(F1)}\quad\mathcal{M}(c,m)=t,\ t'\ \Rightarrow\\ \qquad\qquad\exists e.\ \mathcal{B}(c,m)=e,\_,\quad c,m,t\vdash e:t'\\ \text{(F2)}\quad c<:c',\ \mathcal{F}(c',f)=t,c''\ \Rightarrow\\ \qquad\qquad\mathcal{F}(c,f)=t',c'',\ t'=t\\ \text{(F3)}\quad c<:c',\ \mathcal{M}(c,m)=t,t',\\ \qquad\qquad\mathcal{M}(c',m)=t'',t'''\ \Rightarrow\\ \qquad\qquad t=t'',\ t'=t''''\\ \text{(F4)}\quad c<:c',\ \mathcal{B}(c',m)=e',c''\ \Rightarrow\\ \qquad\qquad\exists c'''.\ \mathcal{B}(c,m)=e,c''',\ c'''<:c''\end{cases}$

The judgement $h,\sigma\ \vdash_{\mathbf{wf}}\ \sigma'\ :\ \mathbb{r}$ expresses that the receiver of σ' is within $\mathbb{r}$ as seen from the point of view of σ. $\Gamma\ \vdash_{\mathbf{wf}}\ h,\sigma$ expresses that h,σ respect the typing environment Γ. $\vdash_{\mathbf{wf}}\mathrm{P}$ defines well-formed programs as those where method bodies respect their signatures (F1), fields are not overridden (F2), overridden methods preserve typing (F3), and do not "skip superclasses" (F4).

Definition 32. *A programming language* $\textsc{PL}$ *has a* sound type system *if all programs* $\mathrm{P}\in\textsc{PL}$ *satisfy the constraints:*

(T1) $\Gamma\vdash e:t,\ \ t<:t'\ \Rightarrow\Gamma\vdash e:t'$

(T2) $h\vdash\mathrm{e}_r:t,\ \ t<:t'\ \Rightarrow h\vdash\mathrm{e}_r:t'$

(T3) $h\vdash\mathrm{e}_r:t,\ \ h\preceq h'\ \Rightarrow h'\vdash\mathrm{e}_r:t$

(T4) $h\vdash\sigma\cdot\iota:\_\,c\ \Rightarrow\text{cls}(h,\iota)<:c$

(T5) $h\vdash\sigma\cdot\iota.m(v):t\ \Rightarrow\begin{cases}h\vdash\sigma\cdot\iota:\mathbb{r}\,c\\ \mathcal{M}(c,m)=t',t\\ h\vdash\sigma\cdot v:t'\end{cases}$

(T6) $h\vdash\sigma\cdot\sigma'\cdot\textit{ret}\,\mathrm{e}_r\ \textit{prv}\,\mathbb{p}:t\ \ \Rightarrow\ \ h\vdash\sigma'\cdot\mathrm{e}_r:t$

(T7) $\sigma=(\iota,\_,\_,\_),\ h\vdash\sigma\cdot\iota':\mathbb{r}\,\_\ \Rightarrow\iota'\in[\![\mathbb{r}]\!]_{h,\iota}$

(T8) $\Gamma\vdash e:\mathbb{r}\,c,\ \ \Gamma\vdash h,\sigma\ \Rightarrow h,\sigma\vdash e:\mathbb{r}\,c$

(T9) $\forall\mathbb{X}.\ \left.\begin{array}{c}\vdash\mathrm{P},\ \ h\vdash\mathrm{e}_r:t\\ \mathrm{e}_r,\ h\longrightarrow\mathrm{e}'_r,\ h'\end{array}\right\}\Rightarrow h'\vdash\mathrm{e}'_r:t$

(T1) and (T2) express subsumption. (T3) states that runtime expression typing does not depend on the field values assigned in the heap. (T4) states that addresses are typed according to their class in the heap. (T5) and (T6) are a technical constraint stating that method call typing implies that the parameter type and return type set by $\mathcal{M}$ for that method are respected and that proof obligations do not interfere with typing. (T7) states that the region component of a type assigned to an address respects the projection given for that region with respect to the same viewpoint of the typing. The most important constraints are (T8) and (T9): (T8) states the correspondence between typing source expressions and runtime expressions for heaps and stack frames that respect the typing environment; (T9) states that for all well-formed programs, reduction preserves typing.