Doctoral Thesis ETH No. 18522

Universe Types
Topology, Encapsulation, Genericity, and Tools

A dissertation submitted to the

Swiss Federal Institute of Technology Zurich
(ETH Zurich, Switzerland)

for the degree of

Doctor of Sciences

presented by

Werner Michael Dietl

Diplom-Ingenieur, Universität Salzburg
born December 7^{th}, 1976
citizen of Austria

accepted on the recommendation of

Prof. Dr. Peter Müller, examiner
Prof. Dr. Michael D. Ernst, co-examiner
Prof. Dr. Martin Odersky, co-examiner

2009

Acknowledgments

Absolutely Fabulous!

(TV Show & Pet Shop Boys song)

While writing these acknowledgments I realize, again, how many people contributed to making this thesis possible and the experience absolutely fabulous.

Without a doubt, the most influential person over the past years has been Peter Müller. He not only supervised this thesis, but also acted as my academic role model. His way of interacting with colleagues, students, administrators, and his family will always remain something to aspire to.

All the members of the Chair of Programming Methodology made sure that we had an inspiring and friendly work environment and also social activities. I thank Ádám D., Joseph R., Arsenii R., Hermann L., and Martin N. for the many years we spent together and am glad that I met Cédric F., Pietro F., Ioannis K., Laura K., and Alex S. before leaving the group.

I thank Bertrand Meyer for welcoming us into his larger group and the interesting discussions I had with him and members of his group.

Probably my biggest influence outside of ETH comes from Sophia Drossopoulou. She always forced me to try harder, simplify the formalizations, and make better examples. I am also grateful for the visits to Imperial College that she enabled and for her enormous hospitality every time.

I thank my co-examiners Mike Ernst and Martin Odersky for their time, feedback, and questions. Mike, also, for providing me with a new academic home.

My interest and involvement with JML is in large parts due to the help and numerous discussions with Gary Leavens. His continued investment into JML and building the JML community is very inspiring.

I am thankful for the many discussions and learning experiences with all my co-authors. During my thesis work, besides Sophia and Peter, I was inspired by Arnd Poetzsch-Heffter, Dave Cunningham, Adrian Francalanza, Alex Summers, and Nick Cameron. Thanks also to Ganesh group in Salzburg that started my academic endeavors and continued to provide an alternative view during the PhD: my supervisor Andreas Uhl, my co-authors Michael Brachtl and Peter Meerwald, and Dominik Engel and Rade Kutil for the camaraderie.

I am deeply indebted to all students I supervised in semester and master projects (listed in reverse chronological order): Phokham N., Manfred S., Timur E., Mathias O., Robin Z., Andreas F., Martin K., Dominique S., Annetta S., Ovidio M., Paolo B., Matthias N., Marco B., Stefan N., David G., Nathalie K., Marco M., Dirk W., Frank L., Alex S., Thomas H., Daniel S., and Yann M. Each one of you provided your own insights and taught me something new. I also wish to thank all students from lectures and exercise sessions that made teaching interesting and fun.

Particular thanks are due to the newlywed Katja Abrahams and Hermann Lehner, who

listened to many rants of mine and always were there to cheer me up. All the best with the baby!

After more than a month without it, I start to miss my regular coffee group, consisting of Ruth B., Franziska H., Sandra H., Peter K., Denise S., Marlies W., and Jutta Z. Its members provided the perfect mix of information and entertainment.

I would like to thank Claudia and Andrew P. for making every meeting special, Gabi and Stefan H. for teaching me British humor, Susanne and Luca P. for the good times, Stephanie B. for interesting lunches, Christoph W. for funny dinners, Ruth B. for making me do some exercises, Wolfgang K. for listening, Tobias B., Thomas J., Stefan A. for Sylt and the time since then, Peter M. for the dinners on short notice in Salzburg, Mark M. and Dave C. for the Californian spirit, and Claudio S. for the dinners and vacations spent together.

Thanks also go to my family for letting me roam the world: my mother Maria, brother Stephan, and sister Inge. My aunt Lucia and uncle Werner for showing me Vienna and helping in so many ways. Also, thanks to Hermine, Hedi, and Toni. In memories are my father Odo and my grandmother Grete.

I am deeply thankful to J. P. for all the help, the good food, the explorations of Switzerland, and for always having a smile for me.

Last, but not least, thanks to all the artists, musicians, free software developers, cooks, and everybody else that makes my work and personal life easier and more enjoyable.

Contents

Abstract

We present *Generic Universe Types*, a sound, lightweight ownership type system for type-generic object-oriented programming languages, which cleanly separates the ownership topology from the owner-as-modifier encapsulation discipline and is supported by a comprehensive set of tools.

Mutable references give object-oriented programming languages the power to build complex object structures and to efficiently modify them. However, this power comes at the cost of *aliasing*: two or more references to the same object can exist, and the modifications performed through one reference are visible through all other references. In main-stream object-oriented languages, like Java and C♯, objects and references build a complicated mesh and there is no support to structure the heap. Visibility modifiers (such as `private` and `protected` in Java) only deal with information hiding, for example, ensuring that a field is only accessible from within its declaring class. However, the object referenced by that field might still be aliased and modified by other objects at runtime. There is no support to encapsulate object structures and to ensure that objects at runtime are only accessed in a controlled fashion.

Aliasing and the unstructured nature of the heap lead to problems with, for example, understanding the behavior of a program, adapting a consistent locking discipline to ensure correct concurrent behavior, exchanging the implementation of an interface, and the formal verification of properties.

The concept of *object ownership* has been proposed as a mechanism to structure the heap hierarchically and to provide encapsulation of object structures. Each object is assigned at most one other object as its owning object, and restrictions are enforced on the references. Ownership type systems provide static type annotations to enforce an ownership topology and encapsulation.

For maintaining invariants, the existence of aliases is no problem, as long as these aliases are not used to modify the internal representation of a different object. We call this encapsulation system the owner-as-modifier discipline, because it guarantees that the owner object is always in control of modifications. The Universe type system is a lightweight ownership type system that enforces the owner-as-modifier discipline and supports the modular verification of object-oriented programs.

We define Generic Universe Types (GUT), a combination of type genericity with Universe types. GUT subsumes the previous non-generic Universe type system and cleanly separates the topology from the encapsulation system. We give a complete formalization of the GUT system and prove it sound.

Usually, ownership type systems entangle the enforcement of the ownership topology with the enforcement of an encapsulation system; that is, the structuring of the heap and the restrictions on the use of references are enforced together. In this thesis, we cleanly separate the ownership topology from the encapsulation system, giving a cleaner formalization and allowing the separate reuse.

Finally, we discuss the integration of a Generic Universe Types checker into the Java Modeling Language (JML) compiler suite, the support for Java 7 (JSR 308) annotations, and Scala compiler plug-ins. We also illustrate the automatic inference of ownership modifiers using a static and a runtime approach.

Zusammenfassung

Wir präsentieren *Generic Universe Types*, ein typsicheres, leichtgewichtiges *Ownership*-Typsystem für typ-generische objektorientierte Programmiersprachen, welches die Ownership-Topologie klar von der *Owner-as-Modifier* Kapselungsdisziplin trennt und von einer umfassenden Menge an Werkzeugen unterstützt wird.

Veränderbare Referenzen geben objektorientierten Programmiersprachen die Ausdrucksstärke, komplexe Objektstrukturen aufzubauen und sie effizient zu modifizieren. Für diese Ausdrucksstärke nimmt man jedoch *Aliasing* in kauf: zwei oder mehr Referenzen die das selbe Objekt referenzieren. Modifikationen die durch eine Referenz ausgeführt werden sind auch durch alle anderen Referenzen sichtbar. In geläufigen objektorientierten Programmiersprachen, wie Java und C♯, bilden Objekte und Referenzen ein komplexes Netzwerk und es gibt keine Unterstützung, um den Speicher zu strukturieren. Die Zugriffsmodifikatoren (wie zum Beispiel `private` und `protected` in Java) stellen nur das Geheimnisprinzip sicher, zum Beispiel, dass auf ein Feld nur innerhalb seiner deklarierenden Klasse zugegriffen werden kann. Jedoch kann das vom Feld referenzierte Objekt zur Laufzeit trotzdem mehrfach referenziert werden und über einen Alias modifiziert werden. Es gibt keine Unterstützung der Datenkapselung und keinen Mechanismus um sicherzustellen, dass Objekte zur Laufzeit nur in einer kontrollierten Art benutzt werden.

Aliasing und der unstrukturierte Speicher führen zu einigen Problemen, zum Beispiel beim Programmverständnis, beim Verwenden einer konsistenten Sperrdisziplin für Monitore um korrektes paralleles Verhalten sicherzustellen, beim Austauschen der Implementierung einer Schnittstelle und bei der formalen Verifikation von Programmeigenschaften.

Das Konzept der *Object Ownership* wurde als Mechanismus vorgeschlagen, um den Speicher hierarchisch zu strukturieren und um Datenkapselung für Objektstrukturen sicherzustellen. Jedes Objekt gehört zu maximal einem anderen Objekt und Beschränkungen auf die möglichen Referenzen werden eingehalten. Ownership-Typsysteme verwenden statische Typannotationen um eine Ownership-Topologie und Datenkapselung sicherzustellen.

Für den Erhalt von Invarianten sind verschiedene Referenzen auf das selbe Objekt kein Problem, solange ein Alias nicht zum Verändern der internen Repräsentation eines anderen Objekts verwendet wird. Wir bezeichnen diese Datenkapselungseigenschaft als die Owner-as-Modifier-Disziplin, weil sie sicherstellt, dass der Besitzer eines Objekts Veränderungen am Objekt kontrollieren kann. Das Universe Typsystem ist ein leichtgewichtiges Owernship-Typsystem das die Owner-as-Modifier-Disziplin erzwingt und dadurch die modulare Verifikation von objektorientierten Programmen ermöglicht.

Wir definieren Generic Universe Types (GUT), eine Kombination von Typgenerizität mit dem Universe Typsystem. GUT subsumiert das nicht-generische Universe Typsystem und trennt die Ownership-Topologie klar von der Datenkapselung. Wir geben eine komplette Formalisierung von GUT und zeigen die Fehlerfreiheit.

Normalerweise vermischen Ownership-Typsysteme das Sicherstellen einer Ownership-Topologie und der Datenkapselung, das heisst, die Strukturierung des Speichers und die Verwendung

von Referenzen werden gemeinsam beschränkt. In dieser Doktorarbeit trennen wir diese beiden Konzepte konsequent und bekommen dadurch eine klarere Formalisierung und können die Komponenten getrennt wiederverwenden.

Schlussendlich diskutieren wir die Integration von Generic Universe Types in den Kompiler der Java Modellierungssprache JML, die Unterstützung von Java 7 (JSR 308) Annotationen und die Verwendung von Erweiterungen für den Scala Übersetzer. Wir illustrieren auch wie Ownership Annotationen automatisch inferiert werden können und präsentieren dafür einen statischen und einen dynamischen Ansatz.

Chapter 1

Introduction

1.1 Aliasing in Object-Oriented Languages

This thesis is set in the context of imperative object-oriented languages in the style of Java [86], C♯ [101], and Eiffel [129]. Object-oriented programming languages use mutable objects and references to efficiently model the state of a program. A reference to an object can be used to change the state of the object, either directly by field updates or through method calls.

Individual objects model only a small part of a larger system. The interaction of multiple objects is used to model abstractions. For example, a list can be written as an object that manages a linked sequence of node objects. To add an element to the list, one can directly modify the list header and prepend an additional node. Clients of the list should not know or care about its internal representation.

The power and ease of references comes at a severe cost: *aliasing*, the possibility that the same object is referenced multiple times [129, 5]. Modifications performed through one reference are visible to the holders of aliases, but such changes may be unexpected.

Visibility modifiers (like `private` in Java) deal only with information hiding, for example, ensuring that a field is accessible only from within its declaring class. However, the object referenced by that field might still be aliased and modified by other objects at runtime.

It is easy to accidentally create aliases to supposedly encapsulated objects, by either *leaking* a reference to the internal representation or *capturing* an external object and using it as if it were a representation object.

The programmer has to take great care to avoid accidental leaking or capturing of references. There is no support to encapsulate object structures and ensure that objects at runtime are accessed only in a controlled fashion. The "Secure Coding Guide" from Sun [183], for example, suggests in guideline 2-1 to "Create a copy of mutable inputs and outputs". Creating copies solves the problem of aliasing, but is an easy-to-forget operation, causes performance overhead, and results in the loss of object identities.

The unstructured nature of the heap leads to many problems, ranging from difficulty with program understanding for programmers; to concurrency errors, like deadlocks from locking the same object twice and race conditions by modifying one object through different aliases; to difficulties for the modular, formal verification of programs. The problems with aliasing in imperative and object-oriented languages have long been recognized [98].

In this thesis, we are particularly interested in the modular verification of program properties. For example, assume that we want to keep track of how many elements are in a list and add an integer field `length` to store the current length. We want to maintain the invariant that the number of nodes in the list is equal to the value of the `length` field. The programmer correctly adapts the implementation of the list to keep the length current. However, this is not enough. If a reference to a node can escape from the list abstraction, then modifications of the node can break the invariant of the list, without the list having the possibility to establish the invariant.

For example, a subtype of the list could leak a reference to the representation. Again, the information hiding features of programming languages are not enough to allow the (re-)use of classes within an abstraction and at the same time forbid the misuse from outside.

1.2 Object Ownership

The concept of object ownership allows programmers to structure the object store hierarchically and to control aliasing and access between objects. Ownership has been applied successfully to various problems, for instance, program verification [69, 116, 135, 142], thread synchronization [24, 103], memory management [12, 27], and representation independence [13].

Existing ownership models share a fundamental topology: Each object has at most one owner object. The set of all objects with the same owner is called a *context*. The *root context* is the set of objects with no owner. The ownership relation is a tree order.

However, existing models differ in the encapsulation system they enforce. The original ownership types [48] and their descendants [23, 45, 46, 162] restrict aliasing and enforce the *owner-as-dominator* discipline: All reference chains from an object in the root context to an object o in a different context go through o's owner. This severe restriction of aliasing is necessary for some of the applications of ownership, for instance, memory management and representation independence.

For applications such as program verification, restricting aliasing is not necessary. Instead, it suffices to enforce the *owner-as-modifier* discipline: An object o may be referenced by any other object, but reference chains that do not pass through o's owner must not be used to modify o. This allows owner objects to control state changes of owned objects and thus maintain invariants. The owner-as-modifier discipline has been inspired by Flexible Alias Protection [150]. It is enforced by the Universe type system [63, 55], in Spec♯'s dynamic ownership model [116, 17], and Effective Ownership Types [122]. The owner-as-modifier discipline imposes weaker restrictions than the owner-as-dominator discipline, which allows it to handle common implementations where objects are shared between objects, such as collections with iterators, shared buffers, or the Flyweight pattern [63, 145]. Some implementations can be slightly adapted to satisfy the owner-as-modifier discipline, for example an iterator can delegate modifications to the corresponding collection which owns the internal representation.

The difference between the owner-as-dominator and the owner-as-modifier discipline is illustrated in Fig. 1.1. References that cross context boundaries without going through the owner are always forbidden in the owner-as-dominator system. They are allowed in the owner-as-modifier discipline, but may not be used to modify the referenced object.

1.3 Generic Universe Types

In this thesis, we present Generic Universe Types, an ownership type system for a programming language with generic types similar to Java 5 and C♯ 2.0. Our work builds on our previous combination of Universe Types with generic types [60], but distinguishes itself from our and other previous work by (1) cleanly separating the topological system from the encapsulation system, (2) enforcing the owner-as-modifier discipline, (3) formally integrating type parameters and ownership, and (4) a simple language design building on Universe Types.

The topological system ensures that the hierarchical structure of the object store is enforced and the separate encapsulation system enforces additional rules to ensure the owner-as-modifier discipline or some other set of encapsulation rules. Cleanly separating the ownership topology from the encapsulation system improves the formalization and presentation of ownership

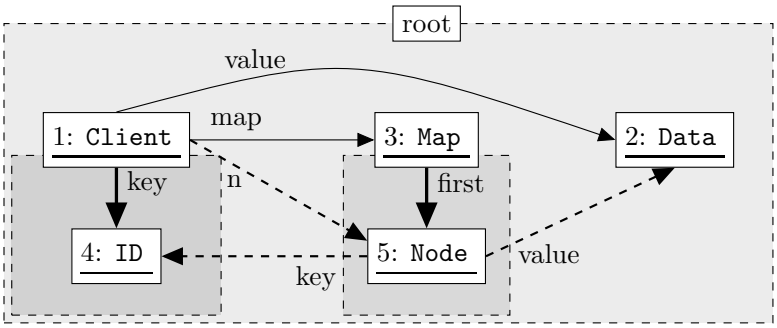

Figure 1.1: Object structure of a map from `ID` to `Data` objects. The map is represented by `Node` objects. The client has a direct reference to a node. Objects, references, and contexts are depicted by rectangles, arrows, and dashed rectangles, respectively. Owner objects sit atop the context of objects they own. Arrows are labeled with the name of the variable that stores the reference. Dashed arrows depict references that cross context boundaries without going through the owner. They are permitted by the owner-as-modifier discipline, but not by owner-as-dominator. If the owner-as-modifier discipline is enforced, such references must not be used to modify the state of the referenced objects. The source code for this example will be shown in Figs. 2.1, 2.2, and 2.3.

type systems. For some applications of ownership, e.g., concurrency [56], the topological structure is sufficient and no encapsulation system needs to be enforced. Also Spec♯'s dynamic ownership model [116] could use our topological system and use logic to enforce a more flexible encapsulation system.

Enforcing only the topological system already provides strong guarantees. Suppose there was a second map object in Fig. 1.1. The topological system guarantees that the `first` field of a map will only reference node objects that belong to its representation. Mixing nodes from different maps is prevented.

1.4 Summary and Contributions

The rest of this thesis is organized as follows.

Chapter 2 is the technical core of the thesis. We introduce Generic Universe Types (GUT) using an example and then develop a complete formalization of GUT by defining the programming language and operational semantics, the topological type system, and an encapsulation system. We state properties of the formalization and, in particular, give the soundness theorem. Finally, we discuss interesting examples and related work.

We previously published the material on non-generic Universe types [62, 67, 63, 66, 115, 55, 34] and a first version of Generic Universe Types [59, 61, 60]; an additional paper on the current version of GUT is under development.

Chapter 3 presents the tool support for GUT, in particular, Sec. 3.1 discusses the type checkers that support GUT and preliminary results about type inference for GUT are presented in Sec. 3.2. First results about the compiler implementation and inference tools were published [68, 65, 64].

Chapter 4 presents future work and Chapter 5 concludes.

Finally, App. A gives the detailed properties and proofs. We formalize GUT using Ott

[176], a system that provides support for defining languages by having a simple input language, sort-checking input, and producing LaTeX and theorem prover output. We include the complete Ott formalization of Generic Universe Types in App. B.

Chapter 2

Generic Universe Types

This chapter presents Generic Universe Types, our extension of the Universe type system to generic types.

Sec. 2.1 illustrates the main concepts of Generic Universe Types by examples. Sec. 2.2 defines the programming language, the runtime system, and the operational semantics. The topological system is given in Sec. 2.3, and Sec. 2.4 presents the encapsulation system that builds on top of the topological system. In Sec. 2.5, we discuss further examples, and, finally, Sec. 2.6 discusses related work.

Technical details are relegated to the appendix. App. A.1 presents additional properties of GUT that are needed in the proofs, and App. A.2 contains the proofs of the properties. App. B.1 gives the complete grammar with all functions that are used in the formalization. In the main part, we omit the definition of helper functions, e.g., definitions that lift operations from single elements to sequences of elements. See App. B.2 for the complete set of definitions.

2.1 Main Concepts

In this section, we explain the main concepts of Generic Universe Types (GUT) informally by two examples: a generic map and an implementation of the decorator pattern.

Class `Map` (Fig. 2.1) implements a generic map from keys to values. Key-value pairs are stored in singly-linked `Node` objects (Fig. 2.2). The `main` method of class `Client` (Fig. 2.3) builds up the map structure shown in Fig. 1.1. In the second example, class `Decorator` can be used to decorate arbitrary objects (Fig. 2.4) as shown in class `Demo` (Fig. 2.5). For simplicity, we omit access modifiers from all examples.

2.1.1 Ownership Modifiers

A type in GUT is either a type variable or consists of an ownership modifier, a class name, and possibly type arguments. The *ownership modifier* expresses object ownership relative to the current receiver object `this`[1]. Programs may contain the ownership modifiers `peer`, `rep`, and `any`. These have the following meanings:

- `peer` expresses that an object has the same owner as the `this` object, that is, that the current object and the referenced object share the same owner and are therefore in the same context.
- `rep` expresses that an object is owned by `this`, that is, the current object is the owner of the referenced object.

[1] We ignore static fields and methods here, but an extension is possible; see Secs. 2.5.5 and 2.5.6.

```
class Map<K, V> {
  rep Node<K, V> first;

  void put(K key, V value) {
    rep Node<K, V> newfirst = new rep Node<K, V>();
    newfirst.init(key, value, first);
    first = newfirst;
  }

  pure V get(any Object key) {
    rep Node<K, V> n = getNode(key);
    return n != null ? n.value : null;
  }

  pure rep Node<K, V> getNode(any Object key) {
    rep Node<K, V> n = first;
    while (n != null) {
      if (n.key.equals(key)) return n;
      n = n.next;
    }
    return null;
  }
}
```

Figure 2.1: An implementation of a generic map. `Map` objects own their `Node` objects, as indicated by the `rep` modifier in all occurrences of class `Node`.

- `any` expresses that an object may have an arbitrary owner. The `any` modifier is a "don't care" modifier and expresses that the ownership of the referenced object is deliberately unspecified for this reference; `any` types therefore are supertypes of the `rep` and `peer` types with the same class and type arguments, as `any` types convey less specific ownership information.

The use of ownership modifiers is illustrated by class `Map` (Fig. 2.1). A `Map` object owns its `Node` objects since they form the internal representation of the map. This ownership relation is expressed by the `rep` modifier of `Map`'s field `first`, which points to the first node of the map.

Internally, the type system uses two additional ownership modifiers, `self` and `lost`:

- `self` is only used as the modifier for the current object `this` and distinguishes the current object from other objects that have the same owner. Therefore, a type with the `self` modifier is a subtype of a type with the `peer` modifier with the same class and type arguments. The use of a separate `self` modifier highlights the special role that the current object plays in ownership systems and simplifies the overall system by removing special cases for accesses on `this`.

- `lost` signifies that the ownership information cannot be expressed statically with one of the other ownership modifiers. It is a "don't know" modifier that indicates that ownership information was "lost" in the type checking process; in contrast to the `any` modifier, concrete ownership information might be needed for the reference. `lost` types are subtypes of corresponding `any` types, because we want to be able to use `any` references to refer to arbitrary objects, including objects with lost ownership; on the other hand,

```
class Node<K, V> {
  K key; V value;
  peer Node<K, V> next;

  void init(K k, V v, peer Node<K, V> n) { key = k; value = v; next = n; }
}
```

Figure 2.2: Nodes form the internal representation of maps. Class `Node` implements nodes for singly-linked lists of keys and values.

> `lost` types are supertypes of the corresponding `peer` and `rep` types, because `lost` types provide less detailed information.

Our encapsulation system enforces the owner-as-modifier discipline by restricting modifications of objects to `self`, `peer`, and `rep` receivers. That is, an expression of a `lost` or an `any` type may be used as receiver of field reads and calls to side-effect free (*pure*) methods, but not of field updates or calls to non-pure methods. To check this property, the encapsulation system requires side-effect free methods to be annotated with the keyword `pure`. This distinction between pure and non-pure methods is not relevant for the topological system.

2.1.2 Viewpoint Adaptation

Since ownership modifiers express ownership relative to `this`, they have to be adapted when this "viewpoint" changes. Consider `Node`'s method `init` (Fig. 2.2). The third parameter has type `peer Node<K,V>` and is used to initialize the `next` field. The `peer` modifier expresses that the parameter object must have the same owner as the receiver of the method. On the other hand, `Map`'s method `put` calls `init` on a `rep Node` receiver, that is, an object that is owned by `this`. Therefore, the third parameter of the call to `init` also has to be owned by `this`. This means that from this particular call's viewpoint, the third parameter needs a `rep` modifier, although it is declared with a `peer` modifier. In the type system, this *viewpoint adaptation* is done by combining the type of the receiver of a call (here, `rep Node<K,V>`) with the type of the formal parameter (here, `peer Node<K,V>`). This combination yields the argument type from the caller's point of view (here, `rep Node<K,V>`).

Viewpoint adaptation results in lost ownership information if the ownership is not expressible from the new viewpoint. For instance imagine there was a field access `map.first` in Fig. 2.3; the viewpoint adaptation of the field type, `rep Node<K,V>`, yields a `lost` type because there is no ownership modifier to more precisely express a reference into the representation of object `map`. As a consequence, soundness of the topological system requires that methods cannot directly modify a `rep` field of an object other than `this`.

However, if only the topological system is enforced, a reference containing lost or arbitrary ownership information can still be used as receiver. Consider the main program in Fig. 2.3. Local variable `n` stores a reference into the representation of another object, because method `getNode` returns a reference to the internal nodes of the peer map. The update `n.key` is valid, as it preserves the topology of the heap. We have full knowledge of the type of the field after viewpoint adaptation, and no ownership information is lost. On the other hand, the update of field `next` has to be forbidden. After the viewpoint adaptation, the type of the left-hand side contains a `lost` ownership modifier and, therefore, the heap topology cannot be ensured statically.

Viewpoint adaptation and the owner-as-modifier discipline provide encapsulation of internal representation objects. Again, let us consider method `getNode` from class `Map`. By viewpoint

adaptation of the return type, `rep Node<K,V>`, clients of the map can only obtain a `lost` reference to the nodes. The owner-as-modifier discipline requires a `self`, `peer`, or `rep` receiver type for modifications and, thus, guarantees that clients cannot directly modify the node structure. This allows the map to maintain invariants over the nodes, for instance, that the node structure is acyclic.

2.1.3 Type Parameters

Ownership modifiers are also used in actual type arguments. For instance, `Client`'s method `main` instantiates `Map` with the type arguments `rep ID` and `any Data`. Thus, field `map` has type `peer Map<rep ID, any Data>`, which has three ownership modifiers. The *main modifier* `peer` expresses that the `Map` object has the same owner as `this`, whereas the *argument modifiers* `rep` and `any` express ownership of the keys and values relative to the `this` object, in this case that the keys are `ID` objects owned by `this` and that the values are `Data` objects in an arbitrary context. It is important to understand that the argument modifiers again expresses ownership relative to the current `this` object (here, the `Client` object), and not relative to the instance of the generic class that contains the argument modifier (here, the `Map` object `map`).

Type variables are not subject to the viewpoint adaptation that is performed for non-variable types. When type variables are used, for instance, in field declarations, the ownership information they carry stays implicit and does, therefore, not have to be adapted. The substitution of type variables by their actual type arguments happens in the scope in which the type variables were instantiated. Therefore, the viewpoint is the same as for the instantiation, and no viewpoint adaptation is required. For instance, imagine there was a field read `n.key` in method `main` (Fig. 2.3). The declared type of the field is the type variable `K`. Reading the field through the `n` reference substitutes the type variable by the actual type argument `rep ID`, and does not perform a viewpoint adaptation.

Thus, even though the `Map` class does not know the owner of the keys and values (due to the implicit `any` upper bound for `K` and `V`, see below), clients of the map can recover the exact ownership information from the type arguments. This illustrates that Generic Universe Types provide strong static guarantees similar to those of owner-parametric systems [48], even in the presence of `any` types. The corresponding implementation in non-generic Universe Types requires a downcast from the `any` type to a `rep` type and the corresponding runtime check [63].

Type variables have upper bounds, which default to `any Object`. In a class C, the ownership modifiers of an upper bound express ownership relative to the C instance `this`. However, when C's type variables are instantiated, the modifiers of the actual type arguments are relative to the receiver of the method that contains the instantiation. Therefore, checking the conformance of a type argument to its upper bound requires a viewpoint adaptation. Equally, method type variables have upper bounds that are relative to the current instance of the declaring class.

As an example, consider the implementation of the decorator pattern presented in Figs. 2.4 and 2.5. Class `Decorator` (Fig. 2.4) can be used to decorate arbitrary objects with a type provided as method type variable `O`. The upper bound of `O` has the type `peer Decoration<V>`. Method `decorateList` decorates a list of elements. Class `Demo` (Fig. 2.5) presents a use of the decorator: the call of method `decorate` is type correct, because the upper bound for type variable `O` after viewpoint adaptation is `rep Decoration<rep Data>`, which is a supertype of the actual type argument, `rep MyDecoration`. This subtype relation can be derived from the superclass declaration of class `MyDecoration` and adapting to a `rep` viewpoint.

```
class ID { /* ... */ }
class Data { /* ... */ }

class Client {
  peer Map<rep ID, any Data> map;

  void main() {
    map = new peer Map<rep ID, any Data>();
    peer Data value = new peer Data();
    rep ID key = new rep ID();
    map.put(key, value);

    any Node<rep ID, any Data> n = map.getNode(key);
    n.key = new rep ID(); // OK
    n.next = new rep Node<rep ID, any Data>(); // Error
  }
}
```

Figure 2.3: Main program for our map example. The execution of method `main` creates the object structure in Fig. 1.1.

```
class Decoration<V> {
  void set(V val) {}
}

class Decorator {
  <V, O extends peer Decoration<V>>
  O decorate(V in) {
    O res = new O();
    res.set(in);
    return res;
  }

  <V, O extends peer Decoration<V>>
  peer List<O> decorateList(any List<V> inlist) {
    peer List<O> res = new peer List<O>();
    for( V in : inlist ) {
      res.add( decorate<V, O>(in) );
    }
    return res;
  }
}
```

Figure 2.4: A decorator for arbitrary objects. As shown in method `decorate`, type variables with `peer` or `rep` upper bounds can be instantiated.

```
class MyDecoration extends Decoration<peer Data> {
  peer Data f;

  void set(peer Data d) { f = d; }
}

class Demo {
   void main() {
      rep Data d = new rep Data();
      rep MyDecoration dd =
         new rep Decorator().decorate<rep Data, rep MyDecoration>(d);
   }
}
```

Figure 2.5: Class `MyDecoration` decorates `peer Data` objects and is then used by class `Demo`.

```
class ClientUser {
  void useMap( peer Client client ) {
    client.map.put(new rep ID(), new peer MyData()); // Error
  }
}
```

Figure 2.6: Viewpoint adaptation of the map results in lost ownership.

2.1.4 The `lost` and `any` Modifiers and Limited Covariance

There is a fundamental difference between a generic type that uses an arbitrary owner as type argument and a generic type that uses an unknown owner as type argument.

For example, the map from Fig. 2.3 has type `peer Map<rep ID, any Data>` and specifies that the map object has the same owner as the current object, that the keys are `ID` objects owned by the current object, and that the values are Data objects that have arbitrary owners. The type specifies that an arbitrary owner is allowed for the values and that it is legal to use peers, reps, or objects with any other kind of ownership.

The adaptation of a type argument might yield a `lost` type, signifying that ownership information could not be expressed from the new viewpoint, as illustrated in Fig. 2.6. The type of the field access `client.map` is `peer Map<lost ID, any Data>`. We can statically still express that the map object itself is in the same context as the current object and we still know that the values are in an arbitrary context. But from this new viewpoint, we cannot express that the keys have to be owned by the client instance `client`; there is no specific ownership modifier for this relation and therefore the `lost` modifier is used. It would not be type safe to allow the call of method `put` on the receiver of this type. The signature for method `put` after viewpoint adaptation contains `lost` and the topological system cannot express the precise ownership required for the first argument. On the other hand, the signature of method `get` does not contain `lost` and can still be called. Note that methods `get` and `getNode` use `any Object` as parameter types and not the type variable `K`. Using the upper bound of a type variable instead of the type variable allows us to call a method even if the actual type argument loses ownership information. This is particularly useful for pure methods that do not modify the heap. Note that the same design is used in the Java 5 interface `java.util.List`, e.g., by methods `contains`, `indexOf`, and `remove`. These methods use `Object` as parameter type instead of the corresponding type variable, which allows them to be called on receivers that contain wildcards and, thus, increases the applicability of these methods.

```
class Cast {
  void m( any Object obj ) {
    peer Map<rep ID, any Data> map = (peer Map<rep ID, any Data>) obj;
    map.put(new rep ID(), new peer Data());
  }
}
```

Figure 2.7: Demonstration of a cast.

Subtyping with covariant type arguments is in general not statically type safe. For instance, if `List<String>` were a subtype of `List<Object>`, then clients that view a string list through type `List<Object>` could store `Object` instances in the string list, which breaks type safety. The same problem occurs for the ownership information encoded in types. If `peer Map<rep ID, any Data>` were a subtype of `peer Map<any ID, any Data>`, then clients that view the map through the latter type could use method `put` (Fig. 2.1) to add a new `Node` object where the key has an arbitrary owner, even though the map instance requires a specific owner. The covariance problem can be prevented by disallowing covariant type arguments (like in Java and C♯), using use-site or declaration-site variance annotations (i.e., wildcards as found in Java or variance annotations as found in Scala [153]), by runtime checks (as done for arrays in Java), or by elaborate syntactic support [70].

Our topological system supports a limited form of covariance without requiring additional checks. Covariance is permitted if the corresponding modifier of the supertype is `lost`. For example, `peer Map<rep ID, any Data>` is a subtype of `peer Map<lost ID, any Data>`. This is safe because the topological system already prevents updates of variables that contain `lost`. In particular, it is not possible to call method `put`, because the signature after substitution contains `lost`, which prevents the unsound addition of an arbitrary object illustrated above.

2.1.5 Runtime Representation

We store the ownership information and the runtime type arguments including their associated ownership information explicitly in the heap because this information is needed in the runtime checks for casts and for instantiating type variables. In this respect, our runtime model is similar to that of the .NET CLR [107], where runtime information about generics is present and "`new` constraints" can be used to allow the instantiation of type variables.

For example, method `m` in Fig. 2.7 takes an object with an arbitrary owner as argument and uses a cast to retrieve ownership information for the main modifier of the map reference and also for the type arguments. To check this cast at runtime, the object needs to store a reference to its owner and the ownership and type information for the type arguments.

Storing the ownership information at runtime also enables us to create instances of type variables if the main modifier of the corresponding upper bound is `peer` or `rep`. In our language, every class can be instantiated using a uniform `new` expression, which initializes all fields to `null`. Type variables with `any` as upper bound cannot be instantiated, as we could not ensure that the actual type argument provides concrete ownership information, which is necessary for the correct placement in the ownership topology. In the implementation of the decorator pattern, presented in Figs. 2.4 and 2.5, we want to instantiate the type variable `O`; its upper bound is `peer` (`rep` would also be possible, but would limit the possible callers of the method) and we therefore know that the new object will have the same owner as the current object.

There are alternatives to storing the genericity information at runtime: erasure of genericity as found in Java 5 and expansion of generic class declarations as found in C++ templates. It is

possible to erase a GUT program into a Universe Types program without generics [55], using casts. The interpretation of casts and type arguments is the same: both are from the viewpoint of the current object. Therefore, the casts inserted into the erased program use the same types that are used as type arguments. In contrast, expanding the type arguments into the declaring class does not work in general, as the viewpoint for the type argument and the expanded type differ and a viewpoint adaptation is not always possible. See Sec. 2.5.2 for an example.

This concludes our informal introduction to Generic Universe Types. In the next section we present the programming language and semantics on which we build.

2.2 Programming Language Syntax and Semantics

In this section, we define the syntax and operational semantics of the programming language. It presents a standard model for a class-based object-oriented language that is independent of the topological and the encapsulation system, which will be presented in Secs. 2.3 and 2.4.

2.2.1 Programming Language

We formalize Generic Universe Types for a sequential subset of Java 5 and C♯ 2.0 including classes and inheritance, instance fields, dynamically-bound methods, and the usual operations on objects (allocation, field read, field update, casts). For simplicity, we omit several features of Java and C♯ such as interfaces, enum types, exceptions, constructors, static fields and methods, inner classes, primitive types and the corresponding expressions, and all statements for control flow. We do not expect that any of these features is difficult to handle (see for instance [23, 62, 115, 135]). The language we use is similar to Featherweight Generic Java [99]. We added field updates because the treatment of side effects is essential for ownership type systems and especially the owner-as-modifier discipline.

Fig. 2.8 summarizes the syntax of our language and our naming conventions for variables. We assume that all identifiers of a program are globally unique except for `this` as well as method and parameter names of overridden methods. This can be achieved easily by preceding each identifier with the class or method name of its declaration (but we omit this prefix in our examples).

The superscript ^{s} distinguishes the sorts for static checking from the corresponding sorts used to describe the runtime behavior.

A sequence of A's is denoted as $\overline{A}$. In such a sequence, we denote the i-th element by A_i. We denote sequences of a certain length k by $\overline{A_k}$. A sequence $\overline{A}$ can be empty; the empty sequence is denoted by $\emptyset$. We use sequences of "maplets" $S = \overline{a \mapsto b}$ as maps and use a function-like notation to access an element $S(a_i) = b_i$. We use dom to denote the domain of a sequence of maplets, e.g., $\mathrm{dom}(S) = \overline{a}$. We present definitions that lift operations from single elements to sequences of elements and trivial helper functions in App. B.2.

A program P consists of a sequence of classes $\overline{Cls}$, the identifier of a main class C, and a main expression e. A program is executed by creating an instance o of C and then evaluating e with o as `this` object. We assume that we always have access to the current program P, and keep P implicit in the notations. Each class Cls has a class identifier, type variables with upper bounds, a superclass with type arguments, a sequence of field declarations, and a sequence of method declarations. f is used for field identifiers. Like in Java, each class directly or transitively extends the predefined class `Object`.

A method declaration md consists of the purity annotation, the method type variables with their upper bounds, the return type, the method identifier m, the formal method parameters

$$
\begin{array}{rcl}
P & ::= & \overline{Cls},\ C,\ e \\
Cls & ::= & \texttt{class}\ Cid\texttt{<}\overline{TP}\texttt{>}\ \texttt{extends}\ C\texttt{<}\overline{{}^{s}T}\texttt{>}\ \texttt{\{}\ \overline{fd}\ \overline{md}\ \texttt{\}} \\
C & ::= & Cid \mid \texttt{Object} \\
TP & ::= & X\ \texttt{extends}\ {}^{s}N \\
fd & ::= & {}^{s}T\ f\texttt{;} \\
md & ::= & p\ \texttt{<}\overline{TP}\texttt{>}\ {}^{s}T\ m\texttt{(}\overline{mpd}\texttt{)}\ \texttt{\{}\ e\ \texttt{\}} \\
p & ::= & \texttt{pure} \mid \texttt{impure} \\
mpd & ::= & {}^{s}T\ pid \\
e & ::= & \texttt{null} \mid x \mid \texttt{new}\ {}^{s}T\texttt{()} \mid e\texttt{.}f \mid e_0\texttt{.}f\ \texttt{=}\ e_1 \mid \\
 & & \quad e_0\ \texttt{.}\ m\texttt{<}\overline{{}^{s}T}\texttt{>(}\overline{e}\texttt{)} \mid \texttt{(}{}^{s}T\texttt{)}\ e \\
{}^{s}T & ::= & {}^{s}N \mid X \\
{}^{s}N & ::= & u\ C\texttt{<}\overline{{}^{s}T}\texttt{>} \\
u & ::= & \texttt{self} \mid \texttt{peer} \mid \texttt{rep} \mid \texttt{lost} \mid \texttt{any} \\
x & ::= & pid \mid \texttt{this}
\end{array}
$$

pid	parameter identifier
f	field identifier
m	method identifier
X	type variable identifier
Cid	class identifier

Figure 2.8: Syntax of our programming language.

pid with their types, and an expression as body. The result of evaluating the expression is returned by the method. Method parameters x include the explicit method parameters *pid* and the implicit method parameter `this`.

To be able to enforce the owner-as-modifier discipline, we have to distinguish statically between side-effect free (*pure*) methods and methods that potentially have side effects. Pure methods are marked by the keyword `pure`. In our syntax, we mark all other methods by `impure`, although we omit this keyword in our examples. Method purity is not relevant for the discussions in the current section and for the topological system presented in Sec. 2.3; it will be used by the encapsulation system in Sec. 2.4.

An expression e can be the `null` literal, a method parameter access, object creation, field read, field update, method call, or cast.

A type ${}^{s}T$ is either a non-variable type or a type variable identifier X. A non-variable type ${}^{s}N$ consists of an ownership modifier, a class identifier, and a sequence of type arguments.

An ownership modifier u can be `self`, `peer`, `rep`, `lost`, or `any`. Note that we restrict the use of `self` and `lost` in the formalization only as much as is needed for the soundness of the system. For example, we allow the use of `lost` in the declared field type, even though such a field can never be assigned a value. In Sec. 2.5.1 we discuss rules for programmers that prevent useless programs without removing significant expressiveness.

The following subsections define subclassing and look-up functions that make accessing different parts of the program simpler.

2.2.1.1 Subclassing

We use the term *subclassing* (symbol $\sqsubseteq$) to refer to the reflexive and transitive relation on classes declared in a program by the `extends` keyword, irrespective of main modifiers. It is defined on instantiated classes $C\texttt{<}\overline{{}^{s}T}\texttt{>}$, which are denoted by ${}^{s}CT$. The subclass relation is the smallest relation satisfying the rules in Def. 2.2.1. Each class that is instantiated with its type variables

is a subclass of the class and type arguments it is declared to extend (SC1). Subclassing is reflexive (SC2) and transitive (SC3). In all three rules, the subclass is the class instantiated with its declared type variables whereas the superclass is instantiated with type arguments that depend on the relationship between sub- and superclass; this makes substitutions in later rules simpler. The substitution of the type arguments $\overline{{}^{s}T}$ for the type variables $\overline{X}$ in ${}^{s}T$ is denoted by ${}^{s}T\left[\overline{{}^{s}T}/\overline{X}\right]$. For the substitution to be defined, the two sequences have to have the same length.

Definition 2.2.1 (Subclassing)

$\boxed{{}^{s}CT \sqsubseteq {}^{s}CT'}$ subclassing

$$\frac{\texttt{class } Cid\texttt{<}\overline{X_k \texttt{ extends } \_}\texttt{> extends } C'\texttt{<}\overline{{}^{s}T}\texttt{> \{ \_ \_ \}} \in P}{Cid\texttt{<}\overline{X_k}\texttt{>} \sqsubseteq C'\texttt{<}\overline{{}^{s}T}\texttt{>}}\ \text{SC1}$$

$$\frac{\texttt{class } C\texttt{<}\overline{X_k \texttt{ extends } \_}\texttt{>} \ldots \in P}{C\texttt{<}\overline{X_k}\texttt{>} \sqsubseteq C\texttt{<}\overline{X_k}\texttt{>}}\ \text{SC2} \qquad \frac{C\texttt{<}\overline{X}\texttt{>} \sqsubseteq C_1\texttt{<}\overline{{}^{s}T_1}\texttt{>} \quad C_1\texttt{<}\overline{X}_1\texttt{>} \sqsubseteq C'\texttt{<}\overline{{}^{s}T}'\texttt{>}}{C\texttt{<}\overline{X}\texttt{>} \sqsubseteq C'\texttt{<}\overline{{}^{s}T}'\left[\overline{{}^{s}T}_1/\overline{X}_1\right]\texttt{>}}\ \text{SC3}$$

Consider the declaration of class `MyDecoration` in Fig. 2.5. Using rule SC1 we can derive `MyDecoration` $\sqsubseteq$ `Decoration<peer Data>`.

2.2.1.2 Field Type Look-up

Function FType is used to look up the declared type of a field in a class. Note that the function is defined only if the field is declared in the given class; superclasses are not considered. Class `Object` has no fields and therefore the function is undefined in this case.

Definition 2.2.2 (Field Type Look-up)

$\boxed{\text{FType}(C, f) = {}^{s}T}$ look up field f in class C

$$\frac{\texttt{class } Cid\texttt{<\_> extends \_<\_> \{ \_ }{}^{s}T\ f\texttt{; \_ \_ \}} \in P}{\text{FType}(Cid, f) = {}^{s}T}\ \text{SFTC\_DEF}$$

In the above definition, the part `_` ${}^{s}T\ f$`; _ _` is read as "some sequence of field declarations, then a field declaration with static type ${}^{s}T$ and identifier f, followed by another sequence of field declarations, and finally an arbitrary sequence of method declarations."

2.2.1.3 Method Signature Look-up

The look-up of a method signature in a class works like field look-up and yields the method signature of a method with the given name in class C:

Definition 2.2.3 (Method Signature Look-up)

$\boxed{\text{MSig}(C, m) = ms_o}$ look up signature of method m in class C

$$\frac{\begin{array}{l}\texttt{class } Cid\texttt{<\_> extends \_<\_> \{ \_ \_ } ms\ \texttt{\{ } e\ \texttt{\} \_ \}} \in P \\ \text{MName}(ms) = m\end{array}}{\text{MSig}(Cid, m) = ms}\ \text{SMSC\_DEF}$$

MName yields the method name of a method signature. Note that we do not support method overloading, so the method name is sufficient to uniquely identify a method. In the definition of MSig we use ms_o as result to signify an *optional* method signature. In the definition of the

$$
\begin{array}{rcl}
h & ::= & \overline{(\iota \mapsto o)} \\
\iota & & \text{Address} \\
{}^{o}\iota & ::= & \iota \mid \texttt{any}_a \mid \texttt{root}_a \\
v & ::= & \iota \mid \texttt{null}_a \\
o & ::= & \left({}^{r}T, \overline{fv}\right) \\
{}^{r}T & ::= & {}^{o}\iota\ C\texttt{<}\overline{{}^{r}T}\texttt{>} \\
\overline{fv} & ::= & \overline{f \mapsto v} \\
{}^{r}\Gamma & ::= & \left\{\overline{X \mapsto {}^{r}T};\ \overline{x \mapsto v}\right\}
\end{array}
$$

Figure 2.9: Definitions for the runtime model.

method overriding rules (presented in Def. 2.3.17), we need to explicitly distinguish between an undefined method signature (using the notation *None*) and a defined method signature *ms*.

Like in FGJ [99], in a method signature $\texttt{\_ <}\overline{X_l\ \texttt{extends}\ {}^{s}N_l}\texttt{>}\ {}^{s}T\ m\texttt{(}\overline{{}^{s}T_q\ pid}\texttt{)}$ the method type variables $\overline{X_l}$ are bound in the types $\overline{{}^{s}N_l}$, ${}^{s}T$, and $\overline{{}^{s}T_q}$ and α-convertible signatures are equivalent.

2.2.1.4 Class Domain Look-up

The domain of a class is the sequence of type variables that it declares. The predefined class `Object` does not declare any type variables.

Definition 2.2.4 (Class Domain Look-up)

$\boxed{\text{ClassDom}(C) = \overline{X}}$ look up type variables of class C

$$\frac{\texttt{class}\ \mathit{Cid}\texttt{<}\overline{X_k\ \texttt{extends}\ \_}\texttt{>}\ \texttt{extends}\ \_\texttt{<}\_\texttt{>}\ \texttt{\{ \_ \_ \}} \in P}{\text{ClassDom}(\mathit{Cid}) = \overline{X_k}}\ \text{SCD\_\_NVAR}$$

$$\frac{}{\text{ClassDom}(\texttt{Object}) = \emptyset}\ \text{SCD\_\_OBJECT}$$

2.2.1.5 Upper Bounds Look-up

The bounds of a class is the sequence of upper bounds of the type variables that the class declares.

Definition 2.2.5 (Upper Bounds Look-up)

$\boxed{\text{ClassBnds}(C) = \overline{{}^{s}N}}$ look up bounds of class C

$$\frac{\texttt{class}\ \mathit{Cid}\texttt{<}\overline{X_k\ \texttt{extends}\ {}^{s}N_k}\texttt{>}\ \texttt{extends}\ \_\texttt{<}\_\texttt{>}\ \texttt{\{ \_ \_ \}} \in P}{\text{ClassBnds}(\mathit{Cid}) = \overline{{}^{s}N_k}}\ \text{SCBC\_\_NVAR}$$

$$\frac{}{\text{ClassBnds}(\texttt{Object}) = \emptyset}\ \text{SCBC\_\_OBJECT}$$

2.2.2 Runtime Model

Fig. 2.9 defines our model of the runtime system. The prefix ^{r} distinguishes sorts of the runtime model from their static counterparts.

A heap h maps addresses to objects. An address ι is an element of a countable, infinite set of addresses. The domain of a heap h, written $\text{dom}(h)$, is the set of all addresses that are mapped to an object in the heap h. A value v can be an address ι or the special null-address $\texttt{null}_a$. An object o consists of its runtime type and a mapping from field identifiers to the values stored in

the fields. The notations $h(\iota)\downarrow_1$ and $h(\iota)\downarrow_2$ are used to access the first and second component of the object at address ι in heap h.

The runtime type ${}^rT$ of an object o consists of o's owner address, of o's class, and of the runtime types for the type arguments of this class. An owner address ${}^o\iota$ can be the address ι of the owning object, the root owner $\texttt{root}_a$, or the $\texttt{any}_a$ owner. The owner address of objects in the root context is $\texttt{root}_a$. The special owner address $\texttt{any}_a$ is used when the corresponding static type has the `any` modifier. Consider for instance an execution of method `main` (Fig. 2.3), where the address of `this` is 1 and the owner of 1 is $\texttt{root}_a$. The runtime type of the object stored in `map` is $\texttt{root}_a$ `Map`$<$1 `ID`, $\texttt{any}_a$ `Data`$>$.

The first component of a runtime environment ${}^rI$ maps method type variables to their runtime types. The second component is the current stack frame, which maps method parameters to the values they store. Since the domains of these mappings are disjoint, we overload the notation and use ${}^rI(X)$ to access the runtime type for type variable X and ${}^rI(x)$ to access the value for method parameter x.

The following subsections again define various functions to simplify the notation.

2.2.2.1 Heap Operations

Our heap model is very simple: we can create an empty heap $\emptyset$ and can add to or update in an existing heap h a mapping from address ι to an object o, written as $h + (\iota \mapsto o)$. If address ι is already mapped to an object, this mapping is overwritten. We use the shorthand notation $h(\iota.f)$ for reading field f of the object at address ι, i.e., $h(\iota)\downarrow_2(f)$.

For convenience we provide two additional operations, that can be modeled on top of the basic operations: (1) creation of a new object and (2) updating a field value in a heap.

(1) For the addition of an object o as a new object to heap h, resulting in a new heap h' and address ι, we use the notation $h + o = (h', \iota)$. We ensure that ι is a fresh address and that the only modification to the heap is the addition of the new object.

Definition 2.2.6 (Object Addition)

$\boxed{h + o = (h', \iota)}$ add object o to heap h resulting in heap h' and fresh address ι

$$\frac{\iota \notin \mathrm{dom}(h) \qquad h' = h + (\iota \mapsto o)}{h + o = (h', \iota)}\ \textsc{hnew\_\_def}$$

(2) We write $h[\iota.f = v] = h'$ for the update of field f of the object at address ι in heap h to the new value v, resulting in new heap h'. We ensure that the new field value is valid, i.e., v is either $\texttt{null}_a$ or the address of an object in the heap. We also ensure that there already is an object at address ι and that the field identifier f is already in the set of fields of the object, because we do not want to add fields that would not be defined in the corresponding class. In the set of field values $\overline{fv}$, we overwrite the existing mapping for f to arrive at $\overline{fv}'$ and update the heap with the object that consists of the old runtime type and the new field values $\overline{fv}'$. Note that we only change the field value of the single object at address ι and in particular that the runtime types in the heap remain unchanged.

Definition 2.2.7 (Field Update)

$\boxed{h[\iota.f = v] = h'}$ field update in heap

$$\frac{\begin{array}{l} v = \texttt{null}_a \ \vee \ (v = \iota' \ \wedge \ \iota' \in \mathrm{dom}(h)) \\ h(\iota) = \left({}^rT, \overline{fv}\right) \qquad f \in \mathrm{dom}\left(\overline{fv}\right) \qquad \overline{fv}' = \overline{fv}[f \mapsto v] \\ h' = h + \left(\iota \mapsto \left({}^rT, \overline{fv}'\right)\right) \end{array}}{h[\iota.f = v] = h'}\ \textsc{hup\_\_def}$$

2.2.2.2 Runtime Method Signature and Body Look-up

The following function is used to look up the method signature for an object at a particular address. The type-to-value assignment judgment (Def. 2.2.13) determines a class C, a superclass of the runtime class of ι, for which the static method signature function MSig yields a method signature. All overriding methods have the same method name and the overriding rule (Def. 2.3.17) ensures that the different signatures are consistent. $\text{MSig}(P, h, \iota, m)$ yields an arbitrary possible signature, not necessarily the one declared in the smallest supertype of the runtime class of ι; as the different signatures are consistent, this suffices. Note that we do not support method overloading, so the method name is sufficient to uniquely identify a method.

Definition 2.2.8 (Runtime Method Signature Look-up)

$\boxed{\text{MSig}(h, \iota, m) = ms_o}$ look up method signature of method m at ι

$$\frac{h \vdash \iota : \_\ C\texttt{<\_>} \qquad \text{MSig}(C, m) = ms}{\text{MSig}(h, \iota, m) = ms}\ \text{RMS\_DEF}$$

At runtime, we need the ability to look up the implementation of a method that is declared in the smallest supertype of the runtime type of the receiver. We first define a look-up function that determines the method body e from the smallest superclass of class C that implements method m.

Definition 2.2.9 (Static Method Body Look-up)

$\boxed{\text{MBody}(C, m) = e}$ look up most-concrete body of m in class C or a superclass

$$\frac{\texttt{class } \mathit{Cid}\texttt{<\_> extends \_<\_> \{ \_ \_ } ms\ \texttt{\{ } e\ \texttt{\} \_ \}} \in P \quad \text{MName}(ms) = m}{\text{MBody}(\mathit{Cid}, m) = e}\ \text{SMBC\_FOUND}$$

$$\frac{\texttt{class } \mathit{Cid}\texttt{<\_> extends } C_1\texttt{<\_> \{ \_ } \overline{ms_n\ \texttt{\{ } e_n\ \texttt{\}}}\ \texttt{\}} \in P \quad \overline{\text{MName}(ms_n) \neq m} \quad \text{MBody}(C_1, m) = e}{\text{MBody}(\mathit{Cid}, m) = e}\ \text{SMBC\_INH}$$

The following function uses the most concrete runtime type of the object at address ι to determine the corresponding method body.

Definition 2.2.10 (Runtime Method Body Look-up)

$\boxed{\text{MBody}(h, \iota, m) = e}$ look up most-concrete body of method m at ι

$$\frac{h(\iota)\downarrow_1 = \_\ C\texttt{<\_>} \qquad \text{MBody}(C, m) = e}{\text{MBody}(h, \iota, m) = e}\ \text{RMB\_DEF}$$

2.2.3 Static Types, Runtime Types, and Values

In this subsection, we discuss two functions that convert static types to corresponding runtime types, the subtyping of runtime types, and what runtime and static types can be assigned to a value. The discussion gives precise semantics to the static types, in particular the meaning of the ownership modifiers.

2.2.3.1 Simple Dynamization of Static Types

We need to put static and runtime types into a relation, e.g., when we evaluate an object creation, we need to convert the static type from the expression into the corresponding runtime type that is stored with the newly created object.

We define two dynamization functions: first, a simple dynamization function sdyn that puts strong requirements on the static types that it can convert to runtime types; second, a more general dynamization function dyn that is less restrictive. Defining two distinct dynamization functions allows us to avoid a cyclic dependency between runtime subtyping and the dynamization of static types. To determine runtime subtypes we need to dynamize the instantiation of a superclass into a runtime type and, in general, to determine the dynamization of a static type we need to find a runtime supertype of the type of the current object. We use sdyn only to dynamize static types where we do not need runtime subtyping to determine the runtime type and use dyn in the general case where we can use runtime subtyping.

The simple dynamization function sdyn (Def. 2.2.11) relates a sequence of static types to a corresponding sequence of runtime types. The dynamization is relative to a heap, a viewpoint object ι, a runtime type ${}^{r}T$, and substitutions for `lost` ownership modifiers. This simple version of the dynamization uses only the type information available from its arguments and does not determine additional runtime types from the heap. It is used only to convert static types that appear in the upper bounds declaration and the superclass instantiation of a class.

To be defined, the sdyn function requires consistency between its arguments. A `peer` modifier in a static type expresses that the referenced object has the same owner as the current object, which is the owner ${}^{o}\iota$ of the runtime type ${}^{r}T$. If the owner ${}^{o}\iota$ of the runtime type is not the $\texttt{any}_a$ address, then we use that owner for the substitution of `peer`. However, if ${}^{o}\iota$ is the $\texttt{any}_a$ address, we cannot simply substitute `peer` by $\texttt{any}_a$. In this case we use an address ${}^{o}\iota_1$ that is either an address in the heap or the root owner to substitute `peer` modifiers. If a `rep` modifier appears in the static type, we need to ensure that the substitution for `rep` and for `peer` are consistent, i.e., that the owner of the object that is used to substitute `rep` is the owner address that is used to substitute `peer`. $\text{owner}(h, \iota)$ yields the owner address of the object at address ι in heap h. Note that the viewpoint address ι is used only if the static types contain a `rep` modifier. If there is no `rep` modifier, the viewpoint address is ignored, and ι can be an arbitrary address that is not necessarily in the domain of the heap h.

Definition 2.2.11 (Simple Dynamization of Static Types)

$\boxed{\text{sdyn}\big(\overline{{}^{s}T}, h, \iota, {}^{r}T, \overline{{}^{o}\iota}\big) = \overline{{}^{r}T}}$ simple dynamization of types $\overline{{}^{s}T}$

$$\frac{\begin{array}{cc} {}^{o}\iota' \in \text{dom}(h) \cup \{\texttt{root}_a\} & {}^{o}\iota \neq \texttt{any}_a \implies {}^{o}\iota' = {}^{o}\iota \\ \text{ClassDom}(C) = \overline{X} & \texttt{rep} \in \overline{{}^{s}T} \implies \text{owner}(h, \iota) = {}^{o}\iota' \\ \multicolumn{2}{c}{\overline{{}^{s}T}\left[{}^{o}\iota' / \texttt{peer}, \iota / \texttt{rep}, \texttt{any}_a / \texttt{any}, \overline{{}^{r}T} / \overline{X}, \overline{{}^{o}\iota_i / \texttt{lost}}\right] = \overline{{}^{r}T}'} \end{array}}{\text{sdyn}\big(\overline{{}^{s}T}, h, \iota, {}^{o}\iota\ C\texttt{<}\overline{{}^{r}T}\texttt{>}, \overline{{}^{o}\iota_i}\big) = \overline{{}^{r}T}'}\ \text{SDYN}$$

We use the notation ${}^{s}T\left[{}^{o}\iota / u, \overline{{}^{r}T} / \overline{X}\right]$ for the substitution of owner address ${}^{o}\iota$ for occurrences of ownership modifier u and of runtime types $\overline{{}^{r}T}$ for type variables $\overline{X}$ in the static type ${}^{s}T$. The substitution yields a runtime type if all ownership modifiers and type variables in ${}^{s}T$ are replaced by their runtime counterparts, i.e., owner addresses and runtime types. In the above definition, this might not be the case if the `self` modifier or type variables that are not declared in class C appear in $\overline{{}^{s}T}$. Also, the `lost` modifiers are substituted by the corresponding owners from the sequence $\overline{{}^{o}\iota}$. The number of `lost` modifiers in $\overline{{}^{s}T}$ has to correspond to the length of $\overline{{}^{o}\iota}$

```
class C<X extends rep Data, Y extends any Object> {}
class D<Z extends peer Object> extends C<rep Data, Z> {}
```

$$h = (1 \mapsto (\texttt{root}_a\ \texttt{Client}, \_), 2 \mapsto (1\ \texttt{D<1 Object>}, \_))$$

Figure 2.10: Example program and heap.

and the i-th occurrence of `lost` is substituted by the i-th owner in $\overline{o_l}$. If the lengths do not match, the substitution is undefined. Our type system applies sdyn only within its domain.

As a simple example, using the decorator from Fig. 2.5, let us consider a heap
$h = (1 \mapsto (\texttt{root}_a\ \texttt{Client}, \_), 2 \mapsto (\texttt{root}_a\ \texttt{MyDecoration}, \_))$.
Then
$\text{sdyn}(\texttt{peer Data}, h, 2, \texttt{root}_a\ \texttt{MyDecoration}, \emptyset)$
results in $\texttt{root}_a\ \texttt{Data}$.

A more complicated example is given in Fig. 2.10. We can deduce that
$\text{sdyn}(\texttt{rep Data}, h, 2, 1\ \texttt{D<1 Object>}, \emptyset) = 2\ \texttt{Data}$
and
$\text{sdyn}(Z, h, 2, 1\ \texttt{D<1 Object>}, \emptyset) = 1\ \texttt{Object}$.
However,
$\text{sdyn}(X, h, 2, 1\ \texttt{D<1 Object>}, \emptyset)$
is not defined. Type variable `X` is defined in a supertype of the given runtime type. sdyn does not apply subtyping to find a correct substitution for type variables that are defined in supertypes. The more complicated dyn is used for such purposes.

2.2.3.2 Subtyping of Runtime Types

Subtyping of runtime types follows subclassing (see Def. 2.2.1). The owner of the supertype ${}^{o}\iota'$ either is the same as the owner of the subtype or $\texttt{any}_a$. The static superclass instantiation $\overline{{}^{s}T}$ is converted into the runtime types $\overline{{}^{r}T}'$ using the runtime subtype ${}^{o}\iota\ C\texttt{<}\overline{{}^{r}T}\texttt{>}$ as argument to sdyn. This ensures that type variables in $\overline{{}^{s}T}$ are substituted by the runtime type arguments $\overline{{}^{r}T}$ that are used in the subtype. There is no variance in the type arguments of a runtime type.

Definition 2.2.12 (Runtime Subtyping)

$\boxed{h \vdash {}^{r}T <:\ {}^{r}T'}$ type ${}^{r}T$ is a subtype of ${}^{r}T'$

$$\frac{C\texttt{<}\overline{X}\texttt{>} \sqsubseteq C'\texttt{<}\overline{{}^{s}T}\texttt{>} \qquad {}^{o}\iota' \in \{{}^{o}\iota, \texttt{any}_a\} \qquad \text{sdyn}\big(\overline{{}^{s}T}, h, \_, {}^{o}\iota\ C\texttt{<}\overline{{}^{r}T}\texttt{>}, \overline{o_l}\big) = \overline{{}^{r}T}'}{h \vdash {}^{o}\iota\ C\texttt{<}\overline{{}^{r}T}\texttt{>} <:\ {}^{o}\iota'\ C'\texttt{<}\overline{{}^{r}T}'\texttt{>}}\ \text{RT\_DEF}$$

Note how an arbitrary address is used as viewpoint object for sdyn. A `rep` modifier in $\overline{{}^{s}T}$ can be substituted with an arbitrary address that creates a consistent result (note that sdyn checks for consistency between the viewpoint address and the owner of the runtime type ${}^{o}\iota$). Also note that there is no equivalent to the `lost` modifier at runtime and that the substitution for `lost` modifiers $\overline{o_l}$ can be chosen arbitrarily. This reflects the interpretation of `lost` as an existential modifier that is substituted by whichever owner address fits the current context. The definition of a well-formed class in the topological system (see Def. 2.3.14) enforces that subclassing never introduces `lost` in the instantiation of a superclass.

For the decorator example (Fig. 2.5) we deduced in Sec. 2.2.1.1 that `MyDecoration` $\sqsubseteq$ `Decoration<peer Data>`. In the previous subsection we deduced the result of applying sdyn to `peer Data`. Combining these results we can deduce

$h \vdash$ `root`$_a$ `MyDecoration` $<:$ `root`$_a$ `Decoration<root`$_a$ `Data>`

and

$h \vdash$ `root`$_a$ `MyDecoration` $<:$ `any`$_a$ `Decoration<root`$_a$ `Data>`.

For the example from Fig. 2.10, we can use subclassing rule SC1 to deduce `D<Z>` $\sqsubseteq$ `C<rep Data, Z>`. Using the results of the application of sdyn from the previous subsection, we can deduce

$h \vdash 1$ `D<1 Object>` $<: 1$ `C<2 Data`$, 1$ `Object>`. Instead of address 2, any other address in the heap whose owner is address 1 could be chosen by the subtyping judgment.

We also have

$h \vdash 1$ `D<1 Object>` $<:$ `any`$_a$ `D<1 Object>`.

What supertypes of `any`$_a$ `D<1 Object>` exist? We can deduce

$h \vdash$ `any`$_a$ `D<1 Object>` $<:$ `any`$_a$ `C<2 Data`$, 1$ `Object>`.

But note that instead of address 2 any other address in the heap h could be chosen. sdyn can choose an arbitrary viewpoint address, and because there are no `peer` modifiers in the superclass instantiation, the only restriction is that the owner look-up in the heap is defined.

2.2.3.3 Assigning a Runtime Type to a Value

To assign a runtime type ${}^rT$ to an address ι, we determine the most concrete runtime type ${}^rT_1$ from the heap and check whether ${}^rT_1$ is a runtime subtype of ${}^rT$. An arbitrary runtime type can be assigned to the `null`$_a$ value.

Definition 2.2.13 (Assigning a Runtime Type to a Value)

$\boxed{h \vdash v : {}^rT}$ runtime type ${}^rT$ assignable to value v

$$\frac{h(\iota)\downarrow_1 = {}^rT_1 \qquad h \vdash {}^rT_1 <: {}^rT}{h \vdash \iota : {}^rT} \text{ RTT\_ADDR}$$

$$\frac{}{h \vdash \texttt{null}_a : {}^rT} \text{ RTT\_NULL}$$

In the decorator example, we can deduce

$h \vdash 2 :$ `root`$_a$ `MyDecoration`

and

$h \vdash 2 :$ `root`$_a$ `Decoration<root`$_a$ `Data>`.

For the example from Fig. 2.10, we can deduce

$h \vdash 2 : 1$ `D<1 Object>`

and

$h \vdash 2 : 1$ `C<2 Data`$, 1$ `Object>`.

2.2.3.4 Dynamization of a Static Type

This version of the dynamization function is applicable in a more general setting than the simple dynamization function sdyn introduced before. A static type is converted into a runtime type, using a heap, a runtime environment, and substitutions for `lost` modifiers. dyn builds on runtime subtyping, and therefore on sdyn, to determine all necessary type information. This allows dyn to find the runtime equivalents to types that use type variables that are declared in a superclass of the type of the current object—types which cannot be dynamized using sdyn.

Function dyn is given in Def. 2.2.14. We determine the runtime supertype with class C that can be assigned to the current object ι, for which the domain of class C together with the method type variables from the runtime environment define all type variables that appear in the static type. If such a type does not exist, then the dynamization is not defined. The topological type rules (which will be presented in Sec. 2.3) ensure that dyn is never used on such a static type.

The owner ${}^{o}\iota$ of the current object has to be another object in the heap or the root owner $\texttt{root}_a$. We need an owner address other than $\texttt{any}_a$ for the substitution of `peer` and `self` modifiers and prevent that the runtime type-to-value judgment uses $\texttt{any}_a$ in the type by the constraint on the value of ${}^{o}\iota$. For a well-formed heap (which will be defined in Def. 2.3.24), we know that the owner of each object in the heap is not $\texttt{any}_a$ and we can determine such an ${}^{o}\iota$. Note that the runtime type arguments are invariant and that we do not need an additional constraint to ensure that the $\overline{{}^{r}T}$ are the most precise types possible.

We use a substitution that is very similar to the one from the simple dynamization to convert the static type ${}^{s}T$ into the corresponding runtime type ${}^{r}T'$. The owner ${}^{o}\iota$ is used to substitute `self` and `peer` modifiers, the current object ι is used to substitute `rep` modifiers, the class type variables are substituted by the runtime types $\overline{{}^{r}T}$ we determined from the heap for the current object, the method type variables are substituted by the runtime types $\overline{{}^{r}T_l}$ from the runtime environment, and the `lost` modifiers are substituted by the corresponding owners from the sequence $\overline{{}^{o}\iota}$. At runtime we do not distinguish between `self` and `peer` modifiers and substitute both with ${}^{o}\iota$. In Def. 2.2.15 we separately check for uses of `self` as main modifier.

Definition 2.2.14 (Dynamization of a Static Type)

$\boxed{\mathrm{dyn}({}^{s}T, h, {}^{r}\Gamma, \overline{{}^{o}\iota}) = {}^{r}T}$ dynamization of static type (relative to ${}^{r}\Gamma$)

$$\frac{\begin{array}{cc} {}^{r}\Gamma = \left\{\overline{X_l \mapsto {}^{r}T_l}\ ;\ \texttt{this} \mapsto \iota, \_\right\} & h \vdash \iota :\ {}^{o}\iota\ C\texttt{<}\overline{{}^{r}T}\texttt{>} \\ {}^{o}\iota \in \mathrm{dom}(h) \cup \{\texttt{root}_a\} & \mathrm{ClassDom}(C) = \overline{X} \\ {}^{s}T\left[{}^{o}\iota\,/\,\texttt{self}\,,\ {}^{o}\iota\,/\,\texttt{peer}\,,\ \iota\,/\,\texttt{rep}\,,\ \texttt{any}_a\,/\,\texttt{any}\,,\ \overline{{}^{r}T}\,/\,\overline{X}\,,\ \overline{{}^{r}T_l\,/\,X_l}\,,\ \overline{{}^{o}\iota_i\,/\,\texttt{lost}}\right] = {}^{r}T' & \end{array}}{\mathrm{dyn}({}^{s}T, h, {}^{r}\Gamma, \overline{{}^{o}\iota_i}) = {}^{r}T'}\ \textsc{dyne}$$

Note that the outcome of *dyn* depends on finding ${}^{o}\iota\ C\texttt{<}\overline{{}^{r}T}\texttt{>}$, an appropriate supertype of the runtime type of the current object ι, which contains substitutions for all type variables not mapped by the environment. Thus, one may wonder whether there is more than one such appropriate superclass. However, because type variables are globally unique, if the free variables of ${}^{s}T$ are in the domain of a class then they are not in the domain of any other class.

In the Decorator example, using an ${}^{r}\Gamma$ where ${}^{r}\Gamma(\texttt{this}) = 1$, we can deduce

$\mathrm{dyn}(\texttt{peer MyDecoration}, h, {}^{r}\Gamma, \emptyset) = \texttt{root}_a\ \texttt{MyDecoration}$

and

$\mathrm{dyn}(\texttt{peer Decoration<peer Data>}, h, {}^{r}\Gamma, \emptyset) =$
$\texttt{root}_a\ \texttt{Decoration<root}_a\ \texttt{Data>}$.

Using the example from Fig. 2.10, and again an ${}^{r}\Gamma$ where ${}^{r}\Gamma(\texttt{this}) = 1$, we can deduce

$\mathrm{dyn}(\texttt{rep D<rep Object>}, h, {}^{r}\Gamma, \emptyset) = 1\ \texttt{D<1 Object>}$

and

$\mathrm{dyn}(\texttt{rep C<lost Data, rep Object>}, h, {}^{r}\Gamma, 2) = 1\ \texttt{C<2 Data}, 1\ \texttt{Object>}$.

Earlier, we explained that

$\mathrm{sdyn}(X, h, 2, 1\ \texttt{D<1 Object>}, \emptyset)$

is not defined. Let us now consider ${}^{r}\Gamma'(\texttt{this}) = 2$ and $\mathrm{dyn}(\texttt{X}, h, {}^{r}\Gamma', \emptyset)$. Above, we deduced

$h \vdash 2 : 1\ \texttt{C<2 Data}, 1\ \texttt{Object>}$

and class `C` defines a type variable `X`. Therefore, we have that
$\mathrm{dyn}(\texttt{X}, h, {}^{r}\Gamma', \emptyset) = 2\ \texttt{Data}$.

2.2.3.5 Assigning a Static Type to a Value

To assign a static type to a value, we convert the static type into a runtime type, using the provided heap and runtime environment, and check whether this runtime type can be assigned to the value. If the main modifier of the static type is `self` we also have to ensure that the value corresponds to the current object in the runtime environment. Note that we use an arbitrary substitution $\overline{\alpha}$ for `lost` modifiers that might appear in the static type. This expresses the meaning of `lost` as existential quantifier that chooses a suitable owner to fulfill the runtime type-to-value judgment.

Definition 2.2.15 (Assigning a Static Type to a Value (relative to ${}^{r}\Gamma$))

$\boxed{h, {}^{r}\Gamma \vdash v :\ {}^{s}T}$ static type ${}^{s}T$ assignable to value v (relative to ${}^{r}\Gamma$)

$$\frac{\begin{array}{l}\mathrm{dyn}({}^{s}T, h, {}^{r}\Gamma, \overline{\alpha}) = {}^{r}T \qquad h \vdash v :\ {}^{r}T \\ {}^{s}T = \texttt{self\_<\_>} \implies v = {}^{r}\Gamma(\texttt{this})\end{array}}{h, {}^{r}\Gamma \vdash v :\ {}^{s}T}\ \text{RTSTE\_\_DEF}$$

For the well-formed heap judgment (which will be defined in Def. 2.3.24) it is convenient to define a second version of this judgment that only takes the address of the current object instead of a complete runtime environment.

Definition 2.2.16 (Assigning a Static Type to a Value (relative to ι))

$\boxed{h, \iota \vdash v :\ {}^{s}T}$ static type ${}^{s}T$ assignable to value v (relative to ι)

$$\frac{\begin{array}{l}{}^{r}\Gamma = \{\emptyset\ ;\ \texttt{this} \mapsto \iota\} \\ h, {}^{r}\Gamma \vdash v :\ {}^{s}T\end{array}}{h, \iota \vdash v :\ {}^{s}T}\ \text{RTSTA\_\_DEF}$$

For the Decorator example, using the previous results, we can now deduce
$h, 1 \vdash 2 :$ `peer MyDecoration`
and
$h, 1 \vdash 2 :$ `peer Decoration<peer Data>`.

Finally, combining all the results we deduced for the example from Fig. 2.10 we can deduce
$h, 1 \vdash 2 :$ `rep D<rep Object>`
and
$h, 1 \vdash 2 :$ `rep C<lost Data, rep Object>`.
Note that there is no ownership modifier that could be used instead of `lost` and that would fulfill the judgment, because there is no modifier to express that an object is the representation of an object other than `this`.

2.2.4 Operational Semantics

We describe program execution by a big-step operational semantics for expressions and programs.

2.2.4.1 Evaluation of an Expression

The transition ${}^{r}\Gamma \vdash h, e \rightsquigarrow h', v$ expresses that the evaluation of an expression e in heap h and runtime environment ${}^{r}\Gamma$ results in value v and successor heap h'. The rules for evaluating expressions are presented and explained in the following.

Definition 2.2.17 (Evaluation of an Expression)

$\boxed{{}^{r}\Gamma \vdash h, e \rightsquigarrow h', v}$ big-step operational semantics

$$\frac{}{{}^{r}\Gamma \vdash h, \texttt{null} \rightsquigarrow h, \texttt{null}_a}\ \textsc{os\_null} \qquad \frac{{}^{r}\Gamma(x) = v}{{}^{r}\Gamma \vdash h, x \rightsquigarrow h, v}\ \textsc{os\_var}$$

$$\frac{\begin{array}{c}\mathrm{dyn}({}^{s}T, h, {}^{r}\Gamma, \emptyset) = {}^{r}T \qquad \mathrm{ClassOf}({}^{r}T) = C \\ \left(\forall f \in \mathrm{fields}(C)\,.\, \overline{fv}(f) = \texttt{null}_a\right) \qquad h + \left({}^{r}T, \overline{fv}\right) = (h', \iota)\end{array}}{{}^{r}\Gamma \vdash h, \texttt{new}\ {}^{s}T\texttt{()} \rightsquigarrow h', \iota}\ \textsc{os\_new}$$

$$\frac{\begin{array}{c}{}^{r}\Gamma \vdash h, e \rightsquigarrow h', v \\ h', {}^{r}\Gamma \vdash v : {}^{s}T\end{array}}{{}^{r}\Gamma \vdash h, \texttt{(}{}^{s}T\texttt{)}\ e \rightsquigarrow h', v}\ \textsc{os\_cast}$$

$$\frac{\begin{array}{c}{}^{r}\Gamma \vdash h, e_0 \rightsquigarrow h', \iota_0 \\ h'(\iota_0\,.f) = v\end{array}}{{}^{r}\Gamma \vdash h, e_0\,.f \rightsquigarrow h', v}\ \textsc{os\_read} \qquad \frac{\begin{array}{c}{}^{r}\Gamma \vdash h, e_0 \rightsquigarrow h_0, \iota_0 \\ {}^{r}\Gamma \vdash h_0, e_1 \rightsquigarrow h_1, v \\ h_1[\iota_0\,.f\ \texttt{=}\ v] = h'\end{array}}{{}^{r}\Gamma \vdash h, e_0\,.f\ \texttt{=}\ e_1 \rightsquigarrow h', v}\ \textsc{os\_write}$$

$$\frac{\begin{array}{c}{}^{r}\Gamma \vdash h, e_0 \rightsquigarrow h_0, \iota_0 \qquad {}^{r}\Gamma \vdash h_0, \overline{e_q} \rightsquigarrow h_1, \overline{v_q} \\ \mathrm{MBody}(h_0, \iota_0, m) = e \qquad \mathrm{MSig}(h_0, \iota_0, m) = \texttt{\_ <}\overline{X_l}\ \texttt{extends \_>}\ \texttt{\_}\ m\texttt{(}\overline{\texttt{\_}\ pid}\texttt{)} \\ \mathrm{dyn}\left(\overline{{}^{s}T_l}, h, {}^{r}\Gamma, \emptyset\right) = \overline{{}^{r}T_l} \qquad {}^{r}\Gamma' = \left\{\overline{X_l \mapsto {}^{r}T_l}\ ;\ \texttt{this} \mapsto \iota_0\ ,\ \overline{pid \mapsto v_q}\right\} \\ {}^{r}\Gamma' \vdash h_1, e \rightsquigarrow h', v\end{array}}{{}^{r}\Gamma \vdash h, e_0\ .\ m\texttt{<}\overline{{}^{s}T_l}\texttt{>(}\overline{e_q}\texttt{)} \rightsquigarrow h', v}\ \textsc{os\_call}$$

The `null` expression always evaluates to the $\texttt{null}_a$ value (OS_NULL). Parameters, including `this`, are evaluated by looking up the stored value in the stack frame, which is part of the runtime environment ${}^{r}\Gamma$ (OS_VAR). Object creation determines the runtime type for the object from the static type using the heap h and the runtime environment ${}^{r}T$, and initializes all field values to $\texttt{null}_a$. ($\mathrm{ClassOf}({}^{r}T)$ yields the class of the runtime type ${}^{r}T$; note that ${}^{s}T$ could be a type variable and cannot be used to determine the class. $\mathrm{fields}(C)$ yields all fields declared in or inherited by C.) We construct the new object using this runtime type and the field values and add it to the original heap h, resulting in an updated heap h' and the address ι of the new object in the new heap (OS_NEW). For cast expressions, we evaluate the expression e and check that the resulting value is well-typed with the static type given in the cast expression w.r.t. the current environment (OS_CAST).

For field reads (OS_READ), we evaluate the receiver expression e_0 and then look up the field in the heap. We require that the receiver expression evaluates to an address ι_0 and not to the $\texttt{null}_a$ value. For the update of a field f, we evaluate the receiver expression e_0 to address ι_0 and the right-hand side expression to value v, and update the heap h_1 with the new field value (OS_WRITE).

For method calls (OS_CALL), we evaluate the receiver expression e_0 and actual method arguments $\overline{e_q}$ in the usual order. The receiver object is used to look up the most concrete method body and the method signature (from which we extract the names of the method type variables $\overline{X_l}$ and the parameter names $\overline{pid}$ used to construct the new runtime environment ${}^{r}\Gamma'$; the static types in the method signature are irrelevant here). The method body expression e is then evaluated in the runtime environment that maps m's type variables to actual type arguments as well as m's formal method parameters (including `this`) to the actual method arguments. (Note that the method type arguments are dynamized using an empty substitution for `lost` modifiers, which forbids occurrences of `lost` in the type arguments; our type rules enforce this constraint and, therefore, ensure that the dynamization is defined.) The resulting heap and address are the result of the call.

2.2.4.2 Evaluation of a Program

A program with main class C is executed by evaluating the main expression e in a heap h_0 that contains exactly one C instance in the root context where all fields of C are initialized to $\texttt{null}_a$ and a runtime environment ${}^r\Gamma_0$ that maps `this` to this C instance.

Definition 2.2.18 (Evaluation of a Program)

$\boxed{\vdash P \rightsquigarrow h, v}$ big-step operational semantics of a program

$$\frac{\begin{array}{l}\forall f \in \text{fields}(C)\,.\,\overline{fv}(f) = \texttt{null}_a \\ \emptyset + \left(\texttt{root}_a\ C\texttt{<>}, \overline{fv}\right) = (h_0, \iota_0) \\ {}^r\Gamma_0 = \{\emptyset\ ;\ \texttt{this} \mapsto \iota_0\} \qquad {}^r\Gamma_0 \vdash h_0, e \rightsquigarrow h, v\end{array}}{\vdash \overline{Cls},\ C,\ e \rightsquigarrow h, v}\ \text{OSP\_\_DEF}$$

In the example from Sec. 2.1, we would have the classes from Figs. 2.1, 2.2, and 2.3 as the sequence of classes, use class identifier `Client` as main class, and the expression `this.main()` as the main expression.

This concludes our discussion of the programming language syntax and semantics. The following section presents the topological system of GUT.

2.3 Topological System

In this section, we formalize the topological system of GUT. We first formalize viewpoint adaptation and define the ordering of ownership modifiers and subtyping. We then present the static well-formedness conditions, including the type rules, and the runtime well-formedness conditions. We conclude this section by discussing the properties of the topological system, most importantly type safety.

Note that the formalization presents the rules that are necessary for type safety; it allows programs which are not meaningful for programmers to write, e.g., it allows the declared field type to contain the `lost` ownership modifier, even though such a field can never be updated. This design choice keeps the formalization minimal and highlights what is necessary for type safety. In Sec. 2.5.1 we discuss additional constraints that are not needed for soundness, but restrict the programs to a reasonable subset.

Type checking is performed in a type environment ${}^s\Gamma$, which maps the type variables of the enclosing class and method to their upper bounds and method parameters to their types:

$${}^s\Gamma ::= \left\{\overline{X \mapsto {}^sN};\ \overline{x \mapsto {}^sT}\right\}$$

Again, we overload the notation, where ${}^s\Gamma(X)$ refers to the upper bound of type variable X, and ${}^s\Gamma(x)$ refers to the type of method parameter x.

Note that the static environment stores the upper bounds for class and method type variables in its first component; the runtime environment only needs to store the actual type arguments for the method type variables as the arguments for the class type variables are stored in the heap.

2.3.1 Viewpoint Adaptation

Since ownership modifiers express ownership relative to an object, they have to be adapted whenever the viewpoint changes. In the type rules, we need to *adapt a type ${}^sT$ from a viewpoint*

that is described by another type ${}^{s}T'$ *to the viewpoint* `this`. In the following, we omit the phrase "to the viewpoint `this`". To perform the viewpoint adaptation, we define an overloaded operator $\triangleright$ to: (1) Adapt an ownership modifier from a viewpoint that is described by another ownership modifier; (2) Adapt a type from a viewpoint that is described by an ownership modifier; (3) Adapt a type from a viewpoint that is described by another type.

2.3.1.1 Adapting an Ownership Modifier w.r.t. an Ownership Modifier

We explain viewpoint adaptation using a field access $e_1.f$. Analogous adaptations occur for method parameters and results as well as upper bounds of type parameters. Let u be the main modifier of e_1's type, which expresses ownership relative to `this`. Let u' be the main modifier of f's type, which expresses ownership relative to the object that contains f. Then relative to `this`, the type of the field access $e_1.f$ has main modifier $u \triangleright u'$.

Definition 2.3.1 (Adapting Ownership Modifiers)

$\boxed{u \triangleright u' = u''}$ combining two ownership modifiers

$$\frac{}{\texttt{self} \triangleright u = u}\ \text{UCU\_\_SELF} \qquad \frac{}{\texttt{peer} \triangleright \texttt{peer} = \texttt{peer}}\ \text{UCU\_\_PEER}$$

$$\frac{}{\texttt{rep} \triangleright \texttt{peer} = \texttt{rep}}\ \text{UCU\_\_REP} \qquad \frac{}{u \triangleright \texttt{any} = \texttt{any}}\ \text{UCU\_\_ANY}$$

$$\frac{\text{otherwise}}{u \triangleright u' = \texttt{lost}}\ \text{UCU\_\_LOST}$$

The field access $e_1.f$ illustrates the motivation for this definition:

1. Accesses through the current object `this` (that is, e_1 is the variable `this`) do not require a viewpoint adaptation since the ownership modifier of the field is already relative to `this`. The `self` modifier is used to distinguish accesses through `this` from other accesses.

2. If the main modifiers of both e_1 and f are `peer`, then the object referenced by e_1 has the same owner as `this` and the object referenced by $e_1.f$ has the same owner as e_1 and, thus, the same owner as `this`. Consequently, the main modifier of $e_1.f$ is also `peer`.

3. If the main modifier of e_1 is `rep` and the main modifier of f is `peer`, then the main modifier of $e_1.f$ is `rep`, because the object referenced by e_1 is owned by `this` and the object referenced by $e_1.f$ has the same owner as e_1, that is, `this`.

4. If the object referenced by f can have an arbitrary owner, then also the object referenced by $e_1.f$ can have an arbitrary owner, regardless of the owner of e_1. That is, if the main modifier of f is `any`, then also the main modifier of $e_1.f$ is `any`, regardless of the modifier of e_1.

5. In all other cases, we cannot determine statically that the object referenced by $e_1.f$ has the same owner as `this`, is owned by `this`, or that it can be an object with an arbitrary owner. Therefore, in these cases the main modifier of $e_1.f$ is `lost`.

2.3.1.2 Adapting a Type w.r.t. an Ownership Modifier

As explained in Sec. 2.1, type variables are not subject to viewpoint adaptation. For non-variable types, we determine the adapted main modifier and adapt the type arguments recursively:

Definition 2.3.2 (Adapting a Type w.r.t. an Ownership Modifier)

$\boxed{u \triangleright {}^{s}T \;=\; {}^{s}T'}$ ownership modifier - type adaptation

$$\frac{}{u \triangleright X \;=\; X}\ \text{UCT\_\_VAR} \qquad \frac{\begin{array}{c} u \triangleright u' \;=\; u'' \\ u \triangleright \overline{{}^{s}T} \;=\; \overline{{}^{s}T}' \end{array}}{u \triangleright u'\ C\texttt{<}\overline{{}^{s}T}\texttt{>} \;=\; u''\ C\texttt{<}\overline{{}^{s}T}'\texttt{>}}\ \text{UCT\_\_NVAR}$$

2.3.1.3 Adapting a Type w.r.t. a Type

We adapt a type ${}^{s}T$ from the viewpoint described by another type, $u\ C\texttt{<}\overline{{}^{s}T}\texttt{>}$:

Definition 2.3.3 (Adapting a Type w.r.t. a Type)

$\boxed{{}^{s}N \triangleright {}^{s}T \;=\; {}^{s}T'}$ type - type combination

$$\frac{\begin{array}{c} u \triangleright {}^{s}T \;=\; {}^{s}T_1 \\ {}^{s}T_1\left[\overline{{}^{s}T}/\overline{X}\right] = {}^{s}T' \qquad \text{ClassDom}(C) \;=\; \overline{X} \end{array}}{u\ C\texttt{<}\overline{{}^{s}T}\texttt{>} \triangleright {}^{s}T \;=\; {}^{s}T'}\ \text{TCT\_\_DEF}$$

The operator $\triangleright$ adapts the ownership modifiers of ${}^{s}T$ and substitutes the type arguments $\overline{{}^{s}T}$ for the type variables $\overline{X}$ of C. Since the type arguments already are relative to `this`, they are not subject to viewpoint adaptation. Therefore, the substitution of type variables happens after the viewpoint adaptation of ${}^{s}T$'s ownership modifiers.

Note that the first parameter is a non-variable type, because concrete ownership information u is needed to adapt the viewpoint and the actual type arguments $\overline{{}^{s}T}$ are needed to substitute the type variables $\overline{X}$. In the type rules, subsumption will be used to replace type variables by their upper bounds and thereby obtain a concrete type as first argument of $\triangleright$. The adaptation is undefined, if the look-up of the domain of class C is undefined, i.e., C is not a valid class name in the program, or if the number of type arguments does not correspond to the number of type variables.

As an example, consider the call `map.getNode(key)` in Fig. 2.3. The receiver `map` has type `peer Map<rep ID, any Data>`. The return type of the method is `rep Node<K, V>`. This type is first adapted from the viewpoint `peer`, resulting in the type `lost Node<K, V>`; then the substitution of the type arguments for the type variables results in the type `lost Node<rep ID, any Data>`.

If the order of the viewpoint adaptation and the substitution were the other way around, we would first substitute `rep Data` for `K` and `any Data` for `V`, resulting in `rep Node<rep ID, any Data>`. Then, adapting this type from the viewpoint `peer` would result in `lost Node<lost ID, any Data>`. This order of operations would not correctly represent the ownership information of the first type argument.

It is convenient to define look-up functions that determine the declared type(s) of a field, method signature, or upper bound, and adapt it from the viewpoint given by a type. These functions are defined in the following subsections.

2.3.1.4 Adapted Field Type Look-up

To look up the viewpoint-adapted type of a field, we look up the declared type of the field and apply viewpoint adaptation. ClassOf(${}^{s}N$) yields the class of the non-variable type ${}^{s}N$. The FType function is again only defined if the field is declared in the class of the given type.

Definition 2.3.4 (Adapted Field Type Look-up)

$\boxed{\text{FType}({}^{s}N, f) = {}^{s}T}$ look up field f in type ${}^{s}N$

$$\frac{\begin{array}{l}\text{FType}(\text{ClassOf}({}^{s}N), f) = {}^{s}T_1 \\ {}^{s}N \rhd {}^{s}T_1 = {}^{s}T\end{array}}{\text{FType}({}^{s}N, f) = {}^{s}T}\ \text{SFTN\_DEF}$$

2.3.1.5 Adapted Method Signature Look-up

To look up the viewpoint-adapted method signature, we look up the signature in the class of the type and then viewpoint adapt the upper bounds, the return type, and the parameter types. The method type arguments $\overline{{}^{s}T_l}$ are substituted for the method type variables $\overline{X_l}$ in all types.

Definition 2.3.5 (Adapted Method Signature Look-up)

$\boxed{\text{MSig}\left({}^{s}N, m, \overline{{}^{s}T}\right) = ms_o}$ m in ${}^{s}N$ with method type arguments $\overline{{}^{s}T}$ substituted

$$\frac{\begin{array}{l}\text{MSig}(\text{ClassOf}({}^{s}N), m) = p\ \texttt{<}\overline{X_l\ \texttt{extends}\ {}^{s}N_l}\texttt{>}\ {}^{s}T\ m\texttt{(}\overline{{}^{s}T'_q\ pid}\texttt{)} \\ \left({}^{s}N \rhd \overline{{}^{s}N_l}\right)\left[\overline{{}^{s}T_l}/\overline{X_l}\right] = \overline{{}^{s}N'_l} \qquad ({}^{s}N \rhd {}^{s}T)\left[\overline{{}^{s}T_l}/\overline{X_l}\right] = {}^{s}T' \\ \left({}^{s}N \rhd \overline{{}^{s}T'_q}\right)\left[\overline{{}^{s}T_l}/\overline{X_l}\right] = \overline{{}^{s}T''_q}\end{array}}{\text{MSig}\left({}^{s}N, m, \overline{{}^{s}T_l}\right) = p\ \texttt{<}\overline{X_l\ \texttt{extends}\ {}^{s}N'_l}\texttt{>}\ {}^{s}T'\ m\texttt{(}\overline{{}^{s}T''_q\ pid}\texttt{)}}\ \text{SMSN\_DEF}$$

Note that we have to perform capture-avoiding substitutions, that is, free type variables in ${}^{s}N$ must not be captured by the $\overline{X_l}$. If necessary, the $\overline{X_l}$ can be α-renamed in the declared method signature.

2.3.1.6 Adapted Upper Bounds Look-up

The bounds of a type are the upper bounds of the class of the type after viewpoint adaptation.

Definition 2.3.6 (Adapted Upper Bounds Look-up)

$\boxed{\text{ClassBnds}({}^{s}N) = \overline{{}^{s}N}}$ look up bounds of type ${}^{s}N$

$$\frac{\begin{array}{l}\text{ClassBnds}(\text{ClassOf}({}^{s}N)) = \overline{{}^{s}N}_1 \\ {}^{s}N \rhd \overline{{}^{s}N}_1 = \overline{{}^{s}N}\end{array}}{\text{ClassBnds}({}^{s}N) = \overline{{}^{s}N}}\ \text{SCBN\_DEF}$$

2.3.2 Static Ordering Relations

We first define an ordering relation $<:_u$ for ownership modifiers. Recall the definition of subclassing (symbol $\sqsubseteq$) from Def. 2.2.1, which is the reflexive and transitive relation on classes declared in a program. Building on the ordering of ownership modifiers and subclassing we define *subtyping* (symbol $<:$), which additionally takes main modifiers into account.

2.3.2.1 Ordering of Ownership Modifiers

The ordering of ownership modifiers $<:_u$ relates more concrete modifiers below less concrete modifiers. Both `self` and `peer` express that an object has the same owner as `this`, where `self` is only used for the object `this` and is therefore more specific than `peer` (OMO_TP). Both `peer` and `rep` are more specific than `lost` (OMO_PL and OMO_RL). All ownership modifiers are below `any` (OMO_UA) and the ordering of ownership modifiers is reflexive (OMO_REFL).

Definition 2.3.7 (Ordering of Ownership Modifiers)

$\boxed{u <:_u u'}$ ordering of ownership modifiers

$$\frac{}{\texttt{self} <:_u \texttt{peer}}\ \text{OMO\_\_TP} \qquad \frac{}{\texttt{peer} <:_u \texttt{lost}}\ \text{OMO\_\_PL}$$

$$\frac{}{\texttt{rep} <:_u \texttt{lost}}\ \text{OMO\_\_RL} \qquad \frac{}{u <:_u \texttt{any}}\ \text{OMO\_\_UA}$$

$$\frac{}{u <:_u u}\ \text{OMO\_\_REFL}$$

Note that the ordering of ownership modifiers is not transitive, as $\texttt{self} <:_u \texttt{lost}$ is not included; this could be added, but only transitivity of subtyping is needed.

2.3.2.2 Static Subtyping

The subtype relation <: is defined on static types. The judgment ${}^s\Gamma \vdash {}^sT <: {}^sT'$ expresses that type ${}^sT$ is a subtype of type ${}^sT'$ in type environment ${}^s\Gamma$. The environment is needed since static types may contain type variables. The rules for this subtyping judgment are presented in Def. 2.3.8 below.

Two types with the same main modifier are subtypes if the corresponding classes are subclasses. Ownership modifiers in the superclass instantiation ($\overline{{}^sT}_1$) are relative to the instance of class `C`, whereas the modifiers in a type are relative to `this`. In particular, from the subclass relation $C\texttt{<}\overline{X}\texttt{>} \sqsubseteq C'\texttt{<}\overline{{}^sT}_1\texttt{>}$ we cannot simply derive ${}^s\Gamma \vdash u\ C\texttt{<}\overline{X}\texttt{>} <:\ u\ C'\texttt{<}\overline{{}^sT}_1\texttt{>}$. Instead, $\overline{{}^sT}_1$ has to be adapted from the viewpoint of the `C` instance to `this` (ST1). For types with the same class, according to ST2, the main modifiers have to follow the ordering of ownership modifiers and the type arguments have to follow the type argument subtyping $<:_l$, explained below. A type variable is a subtype of itself and of its upper bound in the type environment (ST3). Subtyping is transitive (ST4).

Definition 2.3.8 (Static Subtyping)

$\boxed{{}^s\Gamma \vdash {}^sT <:\ {}^sT'}$ static subtyping

$$\frac{\begin{array}{c} C\texttt{<}\overline{X}\texttt{>} \sqsubseteq C'\texttt{<}\overline{{}^sT}_1\texttt{>} \\ u\ C\texttt{<}\overline{{}^sT}\texttt{>} \rhd \overline{{}^sT}_1\ =\ \overline{{}^sT}' \end{array}}{{}^s\Gamma \vdash u\ C\texttt{<}\overline{{}^sT}\texttt{>} <:\ u\ C'\texttt{<}\overline{{}^sT}'\texttt{>}}\ \text{ST1} \qquad \frac{u <:_u u' \qquad \vdash \overline{{}^sT} <:_l \overline{{}^sT}'}{{}^s\Gamma \vdash u\ C\texttt{<}\overline{{}^sT}\texttt{>} <:\ u'\ C\texttt{<}\overline{{}^sT}'\texttt{>}}\ \text{ST2}$$

$$\frac{{}^sT = X\ \vee\ {}^s\Gamma(X) = {}^sT}{{}^s\Gamma \vdash X <:\ {}^sT}\ \text{ST3} \qquad \frac{\begin{array}{c} {}^s\Gamma \vdash {}^sT <:\ {}^sT_1 \\ {}^s\Gamma \vdash {}^sT_1 <:\ {}^sT' \end{array}}{{}^s\Gamma \vdash {}^sT <:\ {}^sT'}\ \text{ST4}$$

Reflexivity of non-variable types can be deduced from the reflexivity of ownership modifier ordering, type argument subtyping, and rule ST2. For type variables, rule ST3 gives reflexivity.

The type `any Object` is at the root of the type hierarchy. Every other type is a subtype of it.

In Sec. 2.2.1.1 we derived

`MyDecoration` $\sqsubseteq$ `Decoration<peer Data>`.

Using rule ST1 we can derive the two subtype relationships

`rep MyDecoration` <: `rep Decoration<rep Data>`

and

`any MyDecoration` <: `any Decoration<lost Data>`.

Notice that in the second example, we cannot give concrete ownership information for the type argument in the supertype, because we do not know the location of the object and cannot express the relationship between the object and the type argument from an arbitrary viewpoint.

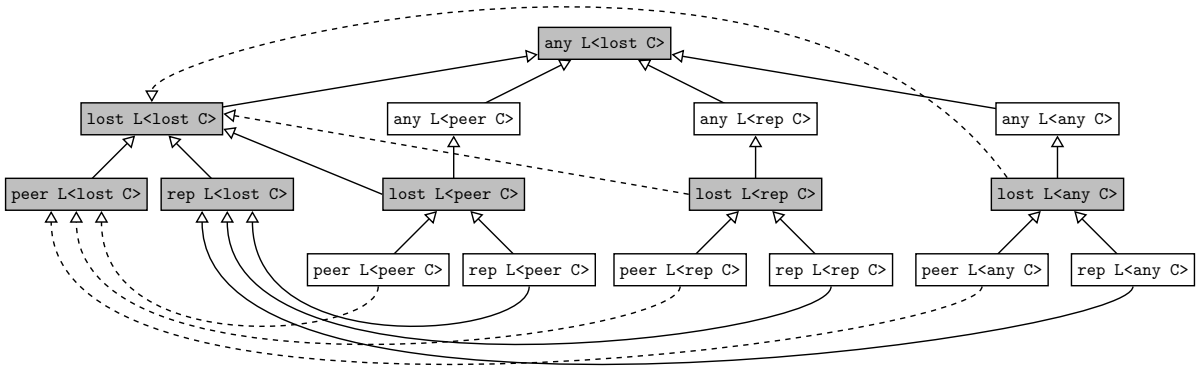

Figure 2.11: Static subtyping illustrated on an example. The dashed lines are only used to make the paths clearer, they have the same meaning as the solid lines. Gray types contain `lost` and cannot be supertypes for strict subtyping (Def. 2.3.10).

By rule ST2 we can derive `rep MyDecoration` <: `any MyDecoration`.

We illustrate rule ST3 using the method call `res.set(in)` from class `Decorator` (Fig. 2.4). The variable `res` has the type variable `O` as its type. To type check the method call, we need a concrete class to look up the method signature for method `set`. We use rule ST3 to go from the type variable `O` to its upper bound `peer Decoration<V>` and can then successfully type check the call.

Type Argument Subtyping. We use type argument subtyping (symbol $<:_l$) only for subtyping in type argument positions. Two non-variable type arguments either have the same main modifier or the supertype has the `lost` main modifier. The type arguments can recursively vary by the type argument relation. A type variable is only a type argument subtype of itself.

Definition 2.3.9 (Type Argument Subtyping)

$\boxed{\vdash {}^sT <:_l {}^sT'}$ type argument subtyping

$$\frac{\begin{array}{l} u' \in \{u\,,\, \texttt{lost}\} \\ \vdash \overline{{}^sT} <:_l \overline{{}^sT}' \end{array}}{\vdash u\ C\texttt{<}\overline{{}^sT}\texttt{>} <:_l u'\ C\texttt{<}\overline{{}^sT}'\texttt{>}}\ \text{AST1} \qquad \frac{}{\vdash X <:_l X}\ \text{AST2}$$

This relation allows us to abstract away ownership information in type arguments. Note that we do not use the same subtyping relation that is used on "top-level" types. It would not be type safe to allow `peer List<peer Data>` as a subtype of `peer List<any Object>`, as the latter type allows storing objects of an arbitrary ownership and class information whereas the former is restricted to `Data` objects that share the same owner as the current object. Also note that we impose restrictions on the uses of types that contain the `lost` modifier. Nevertheless, this abstraction is helpful, as it allows us to reference and modify objects with partially unknown ownership.

Consider the following class declaration:

```
class L<X extends any Object> { ...}
```

Fig. 2.11 shows the relation between instantiations of class `L`. Note that an arbitrary ${}^s\Gamma$ can be used, as we do not consider type variables here.

As another example, consider the code from Fig. 2.10. We can derive the subclass relation `D<Z>` $\sqsubseteq$ `C<rep Data, Z>` and the subtype relation ${}^s\Gamma \vdash$ `self D<Z>` <: `self C<rep Data, Z>`. However, deriving ${}^s\Gamma \vdash$ `peer D<Z>` <: `peer C<rep Data, Z>` is not possible, because it would interpret the instantiation of type variable `X` as representation of the current object, even though it is meant to be the representation of the `D` object. The correct subtype relation is ${}^s\Gamma \vdash$ `peer D<Z>` <: `peer C<lost Data, Z>`.

Strict Subtyping. In certain judgments it is convenient to express that a type ${}^sT$ is a subtype of type ${}^sT'$ and that ${}^sT'$ does not contain the `lost` ownership modifier. We define a strict subtyping judgment to express this.

Definition 2.3.10 (Strict Subtyping)

$\boxed{{}^s\Gamma \vdash {}^sT <:_s {}^sT'}$ strict static subtyping

$$\frac{{}^s\Gamma \vdash {}^sT <: {}^sT' \qquad \texttt{lost} \notin {}^sT'}{{}^s\Gamma \vdash {}^sT <:_s {}^sT'} \text{ SSTDEF}$$

In Fig. 2.11, types that contain `lost` are marked with a gray background. These types cannot appear as a strict supertype. For the example in Fig. 2.10, `peer D<Z>` is a subtype, but not a strict subtype of `peer C<lost Data, Z>`.

2.3.3 Static Well Formedness

This subsection defines well-formedness judgments for the static system, including the topological type rules.

2.3.3.1 Well-formed Static Type

The judgment ${}^s\Gamma \vdash {}^sT$ OK expresses that type ${}^sT$ is well formed in type environment ${}^s\Gamma$. Type variables are well formed, if they are contained in the type environment (WFT_VAR). Well-formedness of the upper bounds is checked by the well-formed static environment judgment (Def. 2.3.18 on page 36) that checks well-formedness of the types in the environment. A non-variable type $u\ C\texttt{<}\overline{{}^sT}\texttt{>}$ is well formed (WFT_NVAR) if its type arguments $\overline{{}^sT}$ are well formed, do not contain `self` (denoted by $\texttt{self} \notin \overline{{}^sT}$), and each actual type argument is a subtype of the upper bound adapted to the current viewpoint.

Definition 2.3.11 (Well-formed Static Type)

$\boxed{{}^s\Gamma \vdash {}^sT \ \ \mathsf{OK}}$ well-formed static type

$$\frac{X \in {}^s\Gamma}{{}^s\Gamma \vdash X \ \ \mathsf{OK}} \text{ WFT\_VAR}$$

$$\frac{{}^s\Gamma \vdash \overline{{}^sT} \ \ \mathsf{OK} \qquad \texttt{self} \notin \overline{{}^sT} \qquad \text{ClassBnds}\big(u\ C\texttt{<}\overline{{}^sT}\texttt{>}\big) = \overline{{}^sN} \qquad {}^s\Gamma \vdash \overline{{}^sT} <: \overline{{}^sN}}{{}^s\Gamma \vdash u\ C\texttt{<}\overline{{}^sT}\texttt{>} \ \ \mathsf{OK}} \text{ WFT\_NVAR}$$

We restrict the `self` modifier to the main modifier of a non-variable type. Allowing the use of `self` in a type argument position would complicate the runtime system, without adding expressiveness.

Note how the look-up of the upper bounds can result in lost ownership information. This well-formed type judgment is intentionally weak and forbids only what is needed for the

soundness of the system; this simplifies the formalization, but complicates the proofs. It allows static types that will never reference a valid object at runtime and could therefore be forbidden without limiting the expressiveness of the system and providing earlier detection of likely errors. See Sec. 2.5.1 for a more restrictive definition, further discussion, and examples.

As an example, consider class C from Fig. 2.10:

```
class C<X extends rep Data, Y extends any Object> {}
```

The type `peer C<peer Data, peer Object>` is well formed. The viewpoint-adapted upper bound of type variable `X` is `lost Data`, which is a supertype of `peer Data`. However, this type will never reference an object at runtime, because the ownership of the type argument is not consistent with the upper bound, and the type system forbids the creation of such objects as we explain next.

2.3.3.2 Strictly Well-formed Static Type

To guarantee well-formedness of the heap, we also define a stricter form of well-formed types that is used for types that can be used for object creations.

A type variable is strictly well formed if it is contained in the type environment (SWFT_VAR). We do not need to put additional constraints on the upper bound of the type variable. A non-variable type is strictly well formed if its type arguments are also strictly well formed, the modifiers `self` and `lost` do not appear in the type, and the type arguments are strict subtypes of the adapted upper bounds (SWFT_NVAR).

Definition 2.3.12 (Strictly Well-formed Static Type)

$\boxed{{}^s\Gamma \vdash {}^sT \ \ \mathsf{strictly\ OK}}$ strictly well-formed static type

$$\frac{X \in {}^s\Gamma}{{}^s\Gamma \vdash X \ \ \mathsf{strictly\ OK}}\ \text{SWFT\_VAR}$$

$$\frac{{}^s\Gamma \vdash \overline{{}^sT} \ \ \mathsf{strictly\ OK} \qquad \{\texttt{self}, \texttt{lost}\} \notin u\ C\texttt{<}\overline{T}\texttt{>} \\ \mathrm{ClassBnds}\big(u\ C\texttt{<}\overline{T}\texttt{>}\big) = \overline{N} \qquad {}^s\Gamma \vdash \overline{{}^sT} <:_s \overline{{}^sN}}{{}^s\Gamma \vdash u\ C\texttt{<}\overline{{}^sT}\texttt{>} \ \ \mathsf{strictly\ OK}}\ \text{SWFT\_NVAR}$$

The use of strict subtyping ensures that `lost` does not appear in the viewpoint-adapted upper bounds and, thus, that no ownership information was lost by the viewpoint adaptation.

Continuing the example from above, the type `peer C<peer Data, peer Object>` is not strictly well formed. The viewpoint-adapted upper bound of type variable `X` contains `lost` and therefore the type cannot be used in `new` expressions. This ensures that a well-formed heap will never contain such an ill-formed object.

We do need both non-strict and strict well-formed type judgments to allow flexible access to objects with lost ownership information. We will present an example after the topological type rules.

2.3.3.3 Topological Type Rules

We are now ready to present the topological type rules. The judgment ${}^s\Gamma \vdash e : {}^sT$ expresses that expression e is well typed with type ${}^sT$ in environment ${}^s\Gamma$. The definition also uses the strict typing judgment ${}^s\Gamma \vdash e :_s {}^sT$ to expresses that expression e is well typed with type ${}^sT$ in environment ${}^s\Gamma$ and that ${}^sT$ does not contain `lost` ownership modifiers; this definition simplifies the type rules TR_WRITE and TR_CALL.

Definition 2.3.13 (Topological Type Rules)

$\boxed{{}^s\Gamma \vdash e : {}^sT}$ expression typing

$$\frac{{}^s\Gamma \vdash e : {}^sT_1 \quad {}^s\Gamma \vdash {}^sT_1 <: {}^sT \quad {}^s\Gamma \vdash {}^sT \ \mathsf{OK}}{{}^s\Gamma \vdash e : {}^sT}\ \text{TR\_SUBSUM} \qquad \frac{\texttt{self} \notin {}^sT \quad {}^s\Gamma \vdash {}^sT \ \mathsf{OK}}{{}^s\Gamma \vdash \texttt{null} : {}^sT}\ \text{TR\_NULL} \qquad \frac{{}^s\Gamma(x) = {}^sT}{{}^s\Gamma \vdash x : {}^sT}\ \text{TR\_VAR}$$

$$\frac{{}^s\Gamma \vdash {}^sT \ \mathsf{strictly\ OK} \quad \mathrm{om}({}^sT, {}^s\Gamma) \in \{\texttt{peer}, \texttt{rep}\}}{{}^s\Gamma \vdash \texttt{new}\ {}^sT\texttt{()} : {}^sT}\ \text{TR\_NEW} \qquad \frac{{}^s\Gamma \vdash e : \_ \quad {}^s\Gamma \vdash {}^sT \ \mathsf{OK}}{{}^s\Gamma \vdash \texttt{(}{}^sT\texttt{)}\ e : {}^sT}\ \text{TR\_CAST}$$

$$\frac{{}^s\Gamma \vdash e_0 : {}^sN_0 \quad \mathrm{FType}({}^sN_0, f) = {}^sT}{{}^s\Gamma \vdash e_0.f : {}^sT}\ \text{TR\_READ} \qquad \frac{{}^s\Gamma \vdash e_0 : {}^sN_0 \quad \mathrm{FType}({}^sN_0, f) = {}^sT \quad {}^s\Gamma \vdash e_1 :_s {}^sT}{{}^s\Gamma \vdash e_0.f = e_1 : {}^sT}\ \text{TR\_WRITE}$$

$$\frac{\begin{array}{c}{}^s\Gamma \vdash e_0 : {}^sN_0 \qquad {}^s\Gamma \vdash \overline{{}^sT_l} \ \mathsf{strictly\ OK} \\ \mathrm{MSig}\big({}^sN_0, m, \overline{{}^sT_l}\big) = \_\ \texttt{<}\overline{X_l\ \texttt{extends}\ {}^sN_l}\texttt{>}\ {}^sT\ m\texttt{(}\overline{{}^sT'_q\ pid}\texttt{)} \\ {}^s\Gamma \vdash \overline{e_q} :_s \overline{{}^sT'_q} \qquad {}^s\Gamma \vdash \overline{{}^sT_l} <:_s \overline{{}^sN_l}\end{array}}{{}^s\Gamma \vdash e_0\ \texttt{.}\ m\texttt{<}\overline{{}^sT_l}\texttt{>(}\overline{e_q}\texttt{)} : {}^sT}\ \text{TR\_CALL}$$

$\boxed{{}^s\Gamma \vdash e :_s {}^sT}$ strict expression typing

$$\frac{{}^s\Gamma \vdash e : {}^sT \quad \texttt{lost} \notin {}^sT}{{}^s\Gamma \vdash e :_s {}^sT}\ \text{STR\_DEF}$$

An expression of type ${}^sT_1$ can also be typed with ${}^sT_1$'s well-formed supertypes (TR_SUBSUM). The `null`-reference can have any well-formed type that does not contain the `self` modifier (TR_NULL). The type of method parameters (including `this`) is determined by a look-up in the type environment (TR_VAR). Objects must be created in a specific context. Therefore, only types with main modifiers `peer` and `rep` are allowed for object creations. Also, the type must be a strictly well-formed static type (TR_NEW). Function om yields the main ownership modifier of a non-variable type and the main ownership modifier of the upper bound of a type variable.

The rule for casts (TR_CAST) is straightforward; it could be strengthened to prevent more cast errors statically, but we omit these checks since they are not strictly needed.

As explained in Sec. 2.3.1, the type of a field access is determined by adapting the declared type of the field from the viewpoint described by the type of the receiver (TR_READ). If the receiver type is a type variable, subsumption is used to determine its upper bound, because FType is defined on non-variable types only. Subsumption is also used for inherited fields to ensure that f is actually declared in ${}^sN_0$.

For a field update, the right-hand side expression must be typable as the viewpoint-adapted field type, which is also the type of the whole field update expression (TR_WRITE). The rule is analogous to field read, but has the additional requirement that the adapted field type does not contain `lost`, which is enforced by using strict expression typing. In this case, we cannot enforce statically that the right-hand side has the required owner, and therefore have to forbid the update.

The rule for method calls (TR_CALL) is in many ways similar to field reads (for result passing) and updates (for argument passing). The method signature is determined using the receiver type ${}^sN_0$ and the actual type arguments $\overline{{}^sT_l}$ substituted for the method's type variables $\overline{X_l}$. Again, subsumption is used to find a type for the receiver that declares the method. The type of the invocation expression is determined by the return type ${}^sT$. The method type arguments must be subtypes of the upper bounds and, modulo subsumption, the actual method argument

expressions $\overline{e_q}$ must have the formal parameter types. For these checks to be precise, we have to forbid `lost` in the upper bounds and the parameter types, which is achieved by using strict subtyping and strict expression typing, respectively. Note that the return type may contain `lost`.

The method type arguments must be strictly well-formed types. Like the static type that is used in an object creation, the static types that are supplied as method type arguments are dynamized to the corresponding runtime types. They are used in the operational semantics to construct the runtime environment. We need to show that the method type arguments are well formed from the viewpoint of the receiver and therefore need to enforce strict well-formed types as method type arguments.

Note that the topological type system treats pure and non-pure methods identically. For type soundness we always need to forbid method calls where the viewpoint-adapted upper bounds or parameter types contain `lost`. The purity information is only used for the encapsulation system, which is presented in Sec. 2.4.

Deterministic object creation. We forbid to create objects that contain the `lost` modifier, statically by enforcing that `lost` is not contained in the type and at runtime by using dyn with an empty substitution for `lost`. Also, the `any` modifier is forbidden as main modifier of the static type. A design alternative would be to allow the creation of types that contain `lost` anywhere in the type and `any` also as main modifier and then at runtime choose an arbitrary owner that fulfills the constraints imposed by the upper bounds. Even though this is not a soundness issue, we prefer the more stringent rules that ensure deterministic behavior of object creation.

We ensure that all subexpressions are well formed, not strictly well formed, to allow flexible access to objects with lost ownership information. Imagine that class `C` from Fig. 2.10 has a field `f` of type `Y`. The type `peer D<peer Object>` is both non-strictly and strictly well formed. Imagine a variable `x` of type `peer D<peer Object>` and a field read `x.f`. By rule TR_READ we need to use subsumption to find the supertype that declares field `f`. This supertype is `peer C<lost Data, peer Object>`, because from the current viewpoint we cannot express the ownership of type argument `X`. This type is not strictly well formed, because it contains `lost` in a type argument and also in an upper bound. However, we can still consider this type well formed and can determine `peer Object` as type of the field access. Similarly, an update of field `f` would be valid. If we required strict well-formedness for all static types, we would lose this significant expressiveness. However, see Sec. 2.5.1 for meaningful further restrictions.

2.3.3.4 Well-formed Class Declaration

The judgment *Cls* OK expresses that class declaration *Cls* is well formed. According to rule WFC_DEF, this is the case if: (1) the upper bounds of *Cls*'s type variables are well formed in the type environment that maps *Cls*'s type variables to their upper bounds; (2) the `self` modifier is not used in *Cls*'s upper bounds; (3) the type arguments to the superclass are strictly well formed and they are strict subtypes of the upper bounds of the superclass; (4) *Cls*'s fields are well formed; and (5) *Cls*'s methods are well formed.

Definition 2.3.14 (Well-formed Class Declaration)

$\boxed{Cls\ \ \mathsf{OK}}$ well-formed class declaration

$$\frac{{}^{s}\Gamma = \left\{\overline{X_k \mapsto {}^{s}N_k}\ ;\ \texttt{this} \mapsto \texttt{self}\ Cid\texttt{<}\overline{X_k}\texttt{>}, \_\right\} \quad {}^{s}\Gamma \vdash \overline{{}^{s}N_k}\ \ \mathsf{OK} \quad \texttt{self} \notin \overline{{}^{s}N_k}}{{}^{s}\Gamma \vdash \overline{{}^{s}T}\ \ \text{strictly OK} \quad \text{ClassBnds}\big(\texttt{self}\ C\texttt{<}\overline{{}^{s}T}\texttt{>}\big) = \overline{{}^{s}N}' \quad {}^{s}\Gamma \vdash \overline{{}^{s}T} <:_s \overline{{}^{s}N}' \quad {}^{s}\Gamma \vdash \overline{fd}\ \ \mathsf{OK} \quad {}^{s}\Gamma, Cid \vdash \overline{md}\ \ \mathsf{OK}}{\texttt{class}\ Cid\texttt{<}\overline{X_k\ \texttt{extends}\ {}^{s}N_k}\texttt{>}\ \texttt{extends}\ C\texttt{<}\overline{{}^{s}T}\texttt{>}\ \texttt{\{}\ \overline{fd}\ \overline{md}\ \texttt{\}}\ \ \mathsf{OK}}\ \text{WFC\_DEF}$$

$$\frac{}{\texttt{class Object \{\}}\ \ \mathsf{OK}}\ \text{WFC\_OBJECT}$$

This definition allows the use of `lost` modifiers in the declaration of the upper bounds of type variables. However, note that such a class can never be instantiated (type rule TR_NEW requires a strictly well-formed type) and also never subclassed (because the instantiation of a superclass does not allow `lost` in the upper bounds). Again, this is not a soundness issue; more stringent checks and examples are discussed in Sec. 2.5.1.

The `self` modifier has the special meaning of referring only to the current object. Using the `self` modifier in an upper bound type would result in the undesired situation that the supertype of the corresponding type variable contains the `self` modifier, even though the type variable obviously does not contain the `self` modifier. For the soundness of the static-type-to-value judgment (see Def. 2.2.15) this has to be forbidden.

In Fig. 2.10, we introduced class `C` with a type variable `X` that has `rep Data` as upper bound. Class C can never be instantiated, because the viewpoint-adapted upper bound always results in lost ownership information. However, the subclass `D` can be instantiated. We therefore consider class C well formed, even though the class can never be instantiated.

2.3.3.5 Well-formed Field Declaration

Field declarations are well formed if their corresponding types are well formed.

Definition 2.3.15 (Well-formed Field Declaration)

$\boxed{{}^{s}\Gamma \vdash {}^{s}T\ f\texttt{;}\ \ \mathsf{OK}}$ well-formed field declaration

$$\frac{{}^{s}\Gamma \vdash {}^{s}T\ \ \mathsf{OK}}{{}^{s}\Gamma \vdash {}^{s}T\ f\texttt{;}\ \ \mathsf{OK}}\ \text{WFFD\_DEF}$$

For soundness, the field types only need to be well formed; they do not need to be strictly well formed. However, we again refer to Sec. 2.5.1 for useful restrictions.

2.3.3.6 Well-formed Method Declaration

The judgment ${}^{s}\Gamma, C \vdash md\ \ \mathsf{OK}$ expresses that method md is well formed in type environment ${}^{s}\Gamma$ and class C. A method declaration md is well formed if: (1) the return type, the upper bounds of md's type variables, and md's parameter types are well formed in the type environment that maps md's and ${}^{s}\Gamma$'s type variables to their upper bounds as well as `this` and the explicit method parameters to their types. The type of `this` is the enclosing class instantiated with its type variables, $C\texttt{<}\overline{X'_k}\texttt{>}$, with main modifier `self`; (2) the upper bounds must not contain the `self` modifier; (3) the method body, expression e, is well typed with md's return type; and (4) md respects the rules for overriding, see below.

Definition 2.3.16 (Well-formed Method Declaration)

$\boxed{{}^{s}\Gamma, C \vdash md \ \ \mathsf{OK}}$ well-formed method declaration

$$\frac{\begin{array}{c}{}^{s}\Gamma = \left\{\overline{X'_k \mapsto {}^{s}N'_k}\ ;\ \texttt{this} \mapsto \texttt{self}\ C\texttt{<}\overline{X'_k}\texttt{>}, \_\right\} \\ {}^{s}\Gamma' = \left\{\overline{X'_k \mapsto {}^{s}N'_k}, \overline{X_l \mapsto {}^{s}N_l}\ ;\ \texttt{this} \mapsto \texttt{self}\ C\texttt{<}\overline{X'_k}\texttt{>}, \overline{pid \mapsto {}^{s}T_q}\right\} \\ {}^{s}\Gamma' \vdash \overline{{}^{s}N_l}, {}^{s}T, \overline{{}^{s}T_q} \ \ \mathsf{OK} \qquad \texttt{self} \notin \overline{{}^{s}N_l} \\ {}^{s}\Gamma' \vdash e : {}^{s}T \qquad C\texttt{<}\overline{X'_k}\texttt{>} \vdash m \ \ \mathsf{OK}\end{array}}{{}^{s}\Gamma, C \vdash \_\ \texttt{<}\overline{X_l\ \texttt{extends}\ {}^{s}N_l}\texttt{>}\ {}^{s}T\ m\texttt{(}\overline{{}^{s}T_q\ pid}\texttt{)}\ \texttt{\{}\ e\ \texttt{\}} \ \ \mathsf{OK}}\ \text{WFMD\_DEF}$$

We allow `lost` in the parameter types and the upper bounds of the method type variables; such a method is never callable, as the type rule for method calls TR_CALL forbids the occurrence of `lost` in these types. See Sec. 2.5.1 for a discussion. Also note that `self` is forbidden only in the upper bound types, but is allowed as main modifier of the return and parameter types. A method with `self` as main modifier of a parameter type is only callable on receiver `this`; `self` as main modifier of the return type will result in `lost`, if the method is not called on receiver `this`.

Method m respects the rules for overriding if it does not override a method or if all overridden methods have the identical signatures after substituting type variables of the superclasses by the instantiations given in the subclass (the notation $ms'[\overline{{}^{s}T}/\overline{X}']$ is used to apply the substitution to the upper bounds, the return type, and the parameter types in the method signature ms'). Consistent renaming of method type variable identifiers and parameter identifiers is allowed.

For simplicity, we require that overrides do not change the purity of a method, although overriding non-pure methods by pure methods would be safe for the encapsulation system in Sec. 2.4. Moreover, parameter and return types are invariant, although contravariant respectively covariant changes could be allowed [55].

Definition 2.3.17 (Method Overriding)

$\boxed{{}^{s}CT \vdash m \ \ \mathsf{OK}}$ method overriding OK

$$\frac{\forall C'\texttt{<}\overline{X}'\texttt{>}.\ \forall \overline{{}^{s}T}.\ \left(C\texttt{<}\overline{X}\texttt{>} \sqsubseteq C'\texttt{<}\overline{{}^{s}T}\texttt{>} \implies C\texttt{<}\overline{X}\texttt{>}, C'\texttt{<}\overline{{}^{s}T}, \ \overline{X}'\texttt{>} \vdash m \ \ \mathsf{OK}\right)}{C\texttt{<}\overline{X}\texttt{>} \vdash m \ \ \mathsf{OK}}\ \text{OVR\_DEF}$$

$\boxed{{}^{s}CT, C\texttt{<}\overline{{}^{s}T}, \ \overline{X}\texttt{>} \vdash m \ \ \mathsf{OK}}$ method overriding OK auxiliary

$$\frac{\begin{array}{c}\text{MSig}(C, m) = ms \qquad \text{MSig}(C', m) = ms'_o \\ ms'_o = None \ \vee \ \left(ms'_o = ms' \ \wedge \ ms'[\overline{{}^{s}T}/\overline{X}'] = ms\right)\end{array}}{C\texttt{<}\overline{X}\texttt{>}, C'\texttt{<}\overline{{}^{s}T}, \ \overline{X}'\texttt{>} \vdash m \ \ \mathsf{OK}}\ \text{OVRA\_DEF}$$

The requirement is expressed by two rules. Rule OVRA_DEF determines the method signatures in classes C and C'. The method signature in the superclass must either be undefined, signified by the value *None*, or must be identical after substitution of the type arguments and possibly renaming of method type variables and parameter identifiers. Rule OVR_DEF quantifies over all superclass instantiations and checks that the methods are consistent using rule OVRA_DEF.

2.3.3.7 Well-formed Type Environment

The judgment ${}^{s}\Gamma \ \ \mathsf{OK}$ expresses that type environment ${}^{s}\Gamma$ is well formed. This is the case if all upper bounds of type variables and the types of method parameters are well formed and `self` does not appear in the upper bounds. Moreover, `this` must be mapped to a non-variable type with main modifier `self` and using the declared type variables of the class as type arguments.

Definition 2.3.18 (Well-formed Type Environment)

$\boxed{{}^{s}\Gamma\ \ \mathsf{OK}}$ well-formed static environment

$$\frac{\begin{array}{c}{}^{s}\Gamma = \left\{\overline{X_k \mapsto {}^{s}N_k}\,,\ \overline{X'_l \mapsto {}^{s}N'_l}\ ;\ \texttt{this} \mapsto \texttt{self}\ C\texttt{<}\overline{X_k}\texttt{>},\ \overline{pid \mapsto {}^{s}T_q}\right\} \\ \mathrm{ClassDom}(C) = \overline{X_k} \qquad \mathrm{ClassBnds}(C) = \overline{{}^{s}N_k} \\ {}^{s}\Gamma \vdash \overline{{}^{s}T_q}\,,\ \overline{{}^{s}N_k}\,,\ \overline{{}^{s}N'_l}\ \ \mathsf{OK} \qquad \texttt{self} \notin \overline{{}^{s}N_k}\,,\ \overline{{}^{s}N'_l}\end{array}}{{}^{s}\Gamma\ \ \mathsf{OK}}\ \text{SWFE\_\_DEF}$$

Note that `self` C`<`$\overline{X_k}$`>` is well formed, because we check that class C is instantiated with its type variables, i.e., $\mathrm{ClassDom}(C) = \overline{X_k}$.

2.3.3.8 Well-formed Program Declaration

The judgment $\vdash P\ \ \mathsf{OK}$ expresses that program P is well formed. This holds if all classes in P are well formed, the main class C is a non-generic class in P, the main expression e is well typed in an environment with `this` as an instance of class C, and where subclassing does not contain cycles.

Definition 2.3.19 (Well-formed Program Declaration)

$\boxed{\vdash P\ \ \mathsf{OK}}$ well-formed program

$$\frac{\begin{array}{l}\overline{Cls_i\ \ \mathsf{OK}} \\ \{\emptyset\ ;\ \texttt{this} \mapsto \texttt{self}\ C\texttt{<>}\} \vdash \texttt{self}\ C\texttt{<>}\ \ \mathsf{OK} \\ \{\emptyset\ ;\ \texttt{this} \mapsto \texttt{self}\ C\texttt{<>}\} \vdash e : \_ \\ \forall C', C''.\ ((C'\texttt{<\_>} \sqsubseteq C''\texttt{<\_>} \ \wedge\ C''\texttt{<\_>} \sqsubseteq C'\texttt{<\_>}) \implies C' = C'')\end{array}}{\vdash \overline{Cls_i},\ C,\ e\ \ \mathsf{OK}}\ \text{WFP\_\_DEF}$$

2.3.4 Runtime Well Formedness

This subsection defines the well-formedness conditions of the runtime system.

2.3.4.1 Runtime Field Type Look-up

It is convenient to look up the declared type of a field for an object. We determine the supertype that can be assigned to the object at address ι, whose class C declares the field f. Note that there is at most one such class in which the field can be declared.

Definition 2.3.20 (Runtime Field Type Look-up)

$\boxed{\mathrm{FType}(h, \iota, f) = {}^{s}T}$ look up type of field in heap

$$\frac{h \vdash \iota :\ \_\ C\texttt{<\_>} \qquad \mathrm{FType}(C, f) = {}^{s}T}{\mathrm{FType}(h, \iota, f) = {}^{s}T}\ \text{RFT\_\_DEF}$$

2.3.4.2 Runtime Upper Bounds Look-up

To look up the runtime types of the upper bounds of a runtime type ${}^{r}T$ from the viewpoint ι, we first determine the static upper bound types and then use simple dynamization to determine the runtime types.

Definition 2.3.21 (Runtime Upper Bounds Look-up)

$\boxed{\mathrm{ClassBnds}(h, \iota, {}^{r}T, \overline{\sigma_l}) = \overline{{}^{r}T}}$ upper bounds of type ${}^{r}T$ from viewpoint ι

$$\frac{\mathrm{ClassBnds}(\mathrm{ClassOf}({}^{r}T)) = \overline{{}^{s}N} \qquad \mathrm{sdyn}\left(\overline{{}^{s}N}, h, \iota, {}^{r}T, \overline{\sigma_l}\right) = \overline{{}^{r}T}}{\mathrm{ClassBnds}(h, \iota, {}^{r}T, \overline{\sigma_l}) = \overline{{}^{r}T}}\ \text{RCB\_\_DEF}$$

We provide a sequence of owner addresses $\overline{\wp}$ that is used to substitute `lost` modifiers that might appear in the static upper bounds look-up.

The simple dynamization requires that `self` does not appear in the static type and that all type variables can be substituted by the runtime type; this is always the case for a well-formed class (see Def. 2.3.14).

2.3.4.3 Strictly Well-formed Runtime Type

By $h,\ \iota \vdash {}^{\wp}\iota\ C\texttt{<}\overline{{}^{r}T}\texttt{>}$ strictly OK we denote that the runtime type ${}^{\wp}\iota\ C\texttt{<}\overline{{}^{r}T}\texttt{>}$ is strictly well formed in heap h with viewpoint address ι. The owner ${}^{\wp}\iota$ has to be an address in the heap or one of the special addresses $\texttt{any}_a$ or $\texttt{root}_a$. The type arguments have to be strictly well formed and must be runtime subtypes of the corresponding upper bounds.

Definition 2.3.22 (Strictly Well-formed Runtime Type)

$\boxed{h,\ \iota \vdash {}^{r}T\ \text{ strictly OK}}$ strictly well-formed runtime type ${}^{r}T$

$$\frac{{}^{\wp}\iota \in \mathrm{dom}(h) \cup \{\texttt{any}_a\,,\,\texttt{root}_a\} \qquad \mathrm{ClassBnds}\big(h, \iota, {}^{\wp}\iota\ C\texttt{<}\overline{{}^{r}T_k}\texttt{>}, \emptyset\big) = \overline{{}^{r}T'_k} \qquad h,\ \_ \vdash \overline{{}^{r}T_k}\ \text{ strictly OK} \qquad h \vdash \overline{{}^{r}T_k} <:\ \overline{{}^{r}T'_k}}{h,\ \iota \vdash {}^{\wp}\iota\ C\texttt{<}\overline{{}^{r}T_k}\texttt{>}\ \text{ strictly OK}}\ \text{SWFRT\_\_DEF}$$

We call this judgment strictly well-formed runtime type because conceptually it corresponds to the strictly well-formed static type judgment. We did not find a need to define a weak form of well-formed runtime type.

This judgment uses a viewpoint address to express that the runtime type is well formed for a specific viewpoint address ι. The address ι is used to determine the runtime upper bound types. If the declared upper bounds contain the `rep` modifier, then ι will be used in the runtime upper bounds. This ensures that the `rep` upper bound is interpreted correctly. It is interesting to note that a class with a `rep` upper bound can never be instantiated, only a subclass of it could be instantiated. We never need to check strict runtime well formedness of such a type and in our uses the viewpoint address ι can be arbitrary.

The type arguments are also checked to be strictly well formed, but we use different viewpoint addresses $\overline{\iota}$. The type arguments are types that were potentially created in a different viewpoint, e.g., they are the result of substituting type arguments for a type variable. Using different viewpoints for the different type arguments allows for each type argument to be relative to a different point of instantiation.

Note that we use an empty sequence of substitutions for `lost` modifiers in ClassBnds. This forbids occurrences of `lost` in the declared upper bounds of a class. Our type rules ensure that we never use strictly well-formed runtime types for classes that use `lost` in upper bounds, because all corresponding static types are checked for strict well formedness.

2.3.4.4 Well-formed Address

An address ι is well formed in a heap h, denoted by $h \vdash \iota$ OK, if the runtime type of the object in the heap is strictly well formed, the root owner $\texttt{root}_a$ is in the set of transitive owners of the object (owners(h, ι) yields the set containing the owner address for ι, owner(h, ι), and all the transitive owners), and for all the fields that are declared in the corresponding class, the field type can be assigned to the field value. By mandating that all objects are (transitively) owned by $\texttt{root}_a$ and because each runtime type has one unique owner address, we ensure that ownership is a tree structure.

Definition 2.3.23 (Well-formed Address)

$\boxed{h \vdash \iota \ \ \mathsf{OK}}$ well-formed address

$$\frac{h(\iota)\downarrow_1 = \_\ C\texttt{<\_>} \qquad h,\ \iota \vdash h(\iota)\downarrow_1 \ \ \mathsf{strictly\ OK} \qquad \texttt{root}_a \in \mathrm{owners}(h,\iota) \\ \forall f \in \mathrm{fields}(C)\,.\ \exists\, {}^sT.\ (\mathrm{FType}(h,\iota,f) = {}^sT \ \wedge\ h,\iota \vdash h(\iota.f) :\ {}^sT)}{h \vdash \iota \ \ \mathsf{OK}}\ \text{WFA\_\_DEF}$$

This definition allows a field type with the `self` main modifier. The address is well formed, if the corresponding field value is the same address again. Also, field types can contain `lost` modifiers and can reference objects of a suitable type, because the static-type-to-value judgment chooses suitable owner addresses. However, the static type rules forbid that such fields are used in an update and therefore a well-formed program will never create such a heap.

2.3.4.5 Well-formed Heap

A heap h is well formed, denoted by $h \ \ \mathsf{OK}$, if all the addresses in the heap are well formed.

Definition 2.3.24 (Well-formed Heap)

$\boxed{h \ \ \mathsf{OK}}$ well-formed heap

$$\frac{\forall \iota \in \mathrm{dom}(h)\,.\ h \vdash \iota \ \ \mathsf{OK}}{h \ \ \mathsf{OK}}\ \text{WFH\_\_DEF}$$

2.3.4.6 Well-formed Environments

We need to express that the runtime information consisting of a heap h and a runtime environment ${}^r\Gamma$ are consistent with the static environment ${}^s\Gamma$, written as $h, {}^r\Gamma : {}^s\Gamma \ \ \mathsf{OK}$.

Definition 2.3.25 (Well-formed Environments)

$\boxed{h, {}^r\Gamma : {}^s\Gamma \ \ \mathsf{OK}}$ runtime and static environments correspond

$$\frac{\begin{array}{ll} {}^r\Gamma = \left\{\overline{X_l \mapsto {}^rT_l}\ ;\ \texttt{this} \mapsto \iota\,,\ \overline{pid \mapsto v_q}\right\} & \\ {}^s\Gamma = \left\{\overline{X_l \mapsto {}^sN_l}\,,\ \overline{X'_k \mapsto \_}\ ;\ \texttt{this} \mapsto \texttt{self}\ C\texttt{<}\overline{X'_k}\texttt{>}\,,\ \overline{pid \mapsto {}^sT_q}\right\} & \\ h \ \ \mathsf{OK} \qquad {}^s\Gamma \ \ \mathsf{OK} \qquad h,\ \iota \vdash \overline{{}^rT_l} \ \ \mathsf{strictly\ OK} & \\ \mathrm{dyn}\left(\overline{{}^sN_l}, h, {}^r\Gamma, \emptyset\right) = \overline{{}^rT'_l} & h \vdash \overline{{}^rT_l} <: \ \overline{{}^rT'_l} \\ h, {}^r\Gamma \vdash \iota :\ \texttt{self}\ C\texttt{<}\overline{X'_k}\texttt{>} & h, {}^r\Gamma \vdash \overline{v_q} :\ \overline{{}^sT_q} \end{array}}{h, {}^r\Gamma : {}^s\Gamma \ \ \mathsf{OK}}\ \text{WFRSE\_\_DEF}$$

The runtime environment only contains the method type variables $\overline{X_l}$ with their runtime type arguments $\overline{{}^rT_l}$, whereas the static environment contains the method type variables $\overline{X_l}$ and the class type variables $\overline{X'_k}$ with their respective static upper bound types. The type of the current object `this` has to match with the class type variables $\overline{X'_k}$ and the type of `this` must be assignable to the current object ι. The formal parameter types $\overline{{}^sT_q}$ must be assignable to the argument values $\overline{v_q}$.

The method type arguments $\overline{{}^rT_l}$ must be strictly well-formed runtime types and must be subtypes of the dynamization of the corresponding upper bounds. Note that the static upper bounds of the method type variables $\overline{{}^sN_l}$ are dynamized using dyn, because the $\overline{{}^sN_l}$ can contain other method type variables which need a runtime environment ${}^r\Gamma$ for dynamization and could also come from a supertype of the runtime type of the current object ι; sdyn would not be defined for such upper bounds. Note that we do not need to ensure that the ownership structure induced by the method type arguments and method arguments is correct, in particular, we do not need to ensure that $\texttt{root}_a$ is contained, as we do in Def. 2.3.24. We ensure that all

addresses are contained in the heap and therefore ensure the well formedness of the ownership structure using Def. 2.3.24.

Also, an empty substitution for `lost` modifiers is used; our type rules ensure that the upper bounds of method type variables do not contain `lost` modifiers (see TR_CALL, Def. 2.3.13; note that a well-formed method might contain `lost` modifiers in upper bounds and parameter types, but such a method will never be callable and therefore we never need to show a correspondence between the static and runtime environments).

The heap h and the static environment ${}^s\Gamma$ have to be well formed according to their respective well-formedness judgments, see Def. 2.3.24 and Def. 2.3.18.

2.3.5 Properties of the Topological System

This final subsection presents the main properties of the topological system and outlines their proofs. Additional properties can be found in App. A.1 and the detailed proofs are in Sec. A.2 on page 106.

2.3.5.1 Type Safety

Type safety of Generic Universe Types in particular ensures that the static ownership information is correctly reflected in the runtime system, which is expressed by the following theorem. If a well-typed expression e is evaluated in a well-formed environment (including a well-formed heap), then the resulting heap is well formed and e's static type can be assigned to the result of e's evaluation.

Theorem 2.3.26 (Type Safety)

$$\left.\begin{array}{l} \vdash P \ \ \mathsf{OK} \qquad h, {}^r\Gamma : {}^s\Gamma \ \ \mathsf{OK} \\ {}^s\Gamma \vdash e : {}^sT \\ {}^r\Gamma \vdash h, e \ \rightsquigarrow \ h', v \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} h', {}^r\Gamma : {}^s\Gamma \ \ \mathsf{OK} \\ h', {}^r\Gamma \vdash v : \ {}^sT \end{array}\right.$$

The proof of Theorem 2.3.26 runs by rule induction on the operational semantics. Lemma 2.3.28 is used to prove field reads and method results, whereas Lemma 2.3.29 is used to prove field updates and method argument passing.

We omit a proof of progress since this property is not affected by adding ownership to a Java-like language. The basic proof can easily be adapted from FGJ [99]. Extensions to include field updates and casts have also been done before [20, 80]. Only the additional check of the ownership information in a cast is different from these previous approaches; its treatment is analogous to a standard Java cast.

2.3.5.2 Validation and Creation of a New Viewpoint

The following judgment is convenient for the formulation of the viewpoint adaptation lemmas. It checks that all type variables that appear in the static type ${}^sT$ are either from the class of ${}^sN$ or from the method type variables $\overline{X_l}$. Also, in some static environment, ${}^sN$ needs to be well formed. This ensures that all type variables are substituted and therefore that the type after substitution of the class and method type variables is consistent with the new environment. Note that the types ${}^sN$ and ${}^sT$ are not part of the new viewpoint ${}^r\Gamma'$, but for brevity we want to include the check of this common side-condition here. The judgment determines the runtime types $\overline{{}^rT_l}$ for the method type arguments using the original environment ${}^r\Gamma$ and an empty substitution for `lost` modifiers. The new viewpoint ${}^r\Gamma'$ is constructed using the method type arguments, the new current object ι, and arbitrary method arguments.

Definition 2.3.27 (Validate and Create a New Viewpoint)

$\boxed{h, {}^r\Gamma \vdash {}^sN, {}^sT; \left(\overline{{}^sT}/\overline{X}, \iota\right) = {}^r\Gamma'}$ validate and create new viewpoint ${}^r\Gamma'$

$$\frac{\begin{array}{c} {}^s\Gamma \vdash {}^sN \ \mathsf{OK} \qquad \mathrm{ClassDom}(\mathrm{ClassOf}({}^sN)) = \overline{X} \qquad \mathrm{free}({}^sT) \subseteq \overline{X}, \overline{X_l} \\ \mathrm{dyn}\left(\overline{{}^sT_l}, h, {}^r\Gamma, \emptyset\right) = \overline{{}^rT_l} \qquad {}^r\Gamma' = \left\{\overline{X_l \mapsto {}^rT_l} \ ; \ \texttt{this} \mapsto \iota, \_\right\} \end{array}}{h, {}^r\Gamma \vdash {}^sN, {}^sT; \left(\overline{{}^sT_l}/\overline{X_l}, \iota\right) = {}^r\Gamma'} \ \text{NVP\_DEF}$$

2.3.5.3 Adaptation from a Viewpoint

The following lemma expresses that viewpoint adaptation from a viewpoint to `this` is correct. Consider the `this` object of a runtime environment ${}^r\Gamma$ and two objects o_1 and o_2. If from the viewpoint `this`, o_1 has the static type ${}^sN$, and from viewpoint o_1, o_2 has the static type ${}^sT$, then from the viewpoint `this`, o_2 has the static type ${}^sT$ adapted from ${}^sN$, ${}^sN \rhd {}^sT$. The following lemma expresses this property using the address ι and value v of the objects o_1 and o_2, respectively. (Note that v can be the $\texttt{null}_a$ value, because every static type (that does not contain `self`) can be assigned to the $\texttt{null}_a$ value.)

Lemma 2.3.28 (Adaptation from a Viewpoint)

$$\left.\begin{array}{l} h, {}^r\Gamma \vdash \iota : {}^sN \\ h, {}^r\Gamma' \vdash v : {}^sT \\ h, {}^r\Gamma \vdash {}^sN, {}^sT; \left(\overline{{}^sT}/\overline{X}, \iota\right) = {}^r\Gamma' \end{array}\right\} \Longrightarrow \begin{array}{l} \exists {}^sT'. \ ({}^sN \rhd {}^sT)\left[\overline{{}^sT}/\overline{X}\right] = {}^sT' \wedge \\ \qquad h, {}^r\Gamma \vdash v : {}^sT' \end{array}$$

This lemma justifies the type rule TR_READ and the method result in TR_CALL. Note how we can choose suitable substitutions for `lost` modifiers in the static types, i.e., the static type after viewpoint adaptation might contain more `lost` ownership information and suitable runtime types are chosen. The proof runs by induction on the shape of static type ${}^sT$.

2.3.5.4 Adaptation to a Viewpoint

The following lemma is the converse of Lemma 2.3.28. It expresses that viewpoint adaptation from `this` to an object o_1 is correct. If from the viewpoint `this`, o_1 has the static type ${}^sN$ and o_2 has the static type ${}^sN \rhd {}^sT$, then from viewpoint o_1, o_2 has the static type ${}^sT$. The lemma requires that the adaptation of ${}^sT$ from viewpoint ${}^sN$ does not contain `lost` ownership modifiers, because the lost ownership information cannot be recovered.

Lemma 2.3.29 (Adaptation to a Viewpoint)

$$\left.\begin{array}{l} h, {}^r\Gamma \vdash \iota : {}^sN \\ ({}^sN \rhd {}^sT)\left[\overline{{}^sT}/\overline{X}\right] = {}^sT' \qquad \texttt{lost} \notin {}^sT' \\ h, {}^r\Gamma \vdash v : {}^sT' \\ h, {}^r\Gamma \vdash {}^sN, {}^sT; \left(\overline{{}^sT}/\overline{X}, \iota\right) = {}^r\Gamma' \end{array}\right\} \Longrightarrow h, {}^r\Gamma' \vdash v : {}^sT$$

This lemma justifies the type rule TR_WRITE and the requirements for the types of the parameters in TR_CALL. The proof again runs by induction on the shape of static type ${}^sT$.

This concludes our discussion of the topological system. In summary, we presented the static and dynamic semantics of the topological system and proved type safety. The next section presents an encapsulation system that builds on top of the topological system.

2.4 Encapsulation System

On top of the topological system that we defined in the previous section, we now define an encapsulation system that enforces the owner-as-modifier discipline. Encapsulation is first defined for expressions and then for methods, classes, and programs. We conclude this section by stating the owner-as-modifier property formally and outlining its proof; the detailed proof can be found in Sec. A.2 on page 119.

The owner-as-modifier discipline allows an object o to be referenced from anywhere, but reference chains that do not pass through o's owner must not be used to modify o. This allows owner objects to control state changes of owned objects and supports program verification [69, 142, 116, 135]. It is also enforced in Spec♯'s dynamic ownership model [116] and Effective Ownership Types [122]. Note that all references, in particular also method parameters and local variables, are subjected to the same restrictions and cannot be used to circumvent the owner-as-modifier discipline.

2.4.1 Encapsulated Expression

The judgment ${}^s\Gamma \vdash e$ `enc`, given below, expresses that expression e is an encapsulated expression, that is, it is a topologically well-typed expression that constrains the possible field updates and method calls.

Definition 2.4.1 (Encapsulated Expression)

$\boxed{{}^s\Gamma \vdash e \ \texttt{enc}}$ encapsulated expression

$$\frac{{}^s\Gamma \vdash \texttt{null} : \_}{{}^s\Gamma \vdash \texttt{null} \ \texttt{enc}}\ \text{E\_\_NULL} \qquad \frac{{}^s\Gamma \vdash x : \_}{{}^s\Gamma \vdash x \ \texttt{enc}}\ \text{E\_\_VAR}$$

$$\frac{{}^s\Gamma \vdash \texttt{new}\ {}^sT\texttt{()} : \_}{{}^s\Gamma \vdash \texttt{new}\ {}^sT\texttt{()} \ \texttt{enc}}\ \text{E\_\_NEW} \qquad \frac{{}^s\Gamma \vdash \texttt{(}{}^sT\texttt{)}\ e : \_ \quad {}^s\Gamma \vdash e \ \texttt{enc}}{{}^s\Gamma \vdash \texttt{(}{}^sT\texttt{)}\ e \ \texttt{enc}}\ \text{E\_\_CAST}$$

$$\frac{{}^s\Gamma \vdash e_0.f : \_ \quad {}^s\Gamma \vdash e_0 \ \texttt{enc}}{{}^s\Gamma \vdash e_0.f \ \texttt{enc}}\ \text{E\_\_READ} \qquad \frac{{}^s\Gamma \vdash e_0.f = e_1 : \_ \quad {}^s\Gamma \vdash e_0 : {}^sN_0 \quad {}^s\Gamma \vdash e_0 \ \texttt{enc} \quad {}^s\Gamma \vdash e_1 \ \texttt{enc} \quad \mathrm{om}({}^sN_0) \in \{\texttt{self}, \texttt{peer}, \texttt{rep}\}}{{}^s\Gamma \vdash e_0.f = e_1 \ \texttt{enc}}\ \text{E\_\_WRITE}$$

$$\frac{{}^s\Gamma \vdash e_0\ .\ m\texttt{<}\overline{{}^sT}\texttt{>(}\overline{e}\texttt{)} : \_ \quad {}^s\Gamma \vdash e_0 : {}^sN_0 \quad {}^s\Gamma \vdash e_0 \ \texttt{enc} \quad {}^s\Gamma \vdash \overline{e} \ \texttt{enc} \quad \mathrm{om}({}^sN_0) \in \{\texttt{self}, \texttt{peer}, \texttt{rep}\} \ \vee \ \mathrm{MSig}\big({}^sN_0, m, \overline{{}^sT}\big) = \texttt{pure <\_> \_}\ m\texttt{(\_)}}{{}^s\Gamma \vdash e_0\ .\ m\texttt{<}\overline{{}^sT}\texttt{>(}\overline{e}\texttt{)} \ \texttt{enc}}\ \text{E\_\_CALL}$$

To enforce the owner-as-modifier discipline, the update of fields of objects in arbitrary contexts must be forbidden. Therefore, field updates are only allowed if the main modifier of the receiver type is `self`, `peer`, or `rep` (E__WRITE). For a method call, either the main modifier of the receiver type is `self`, `peer`, or `rep`, or the called method is pure (E__CALL). Pure methods can be called on arbitrary receivers, because they do not modify existing objects.

The encapsulation judgment prevents method `main` (Fig. 2.3) from updating field `key` of the object referenced by `n`, because the main modifier of `n` is `any`. The update would preserve the topology, but violate the owner-as-modifier discipline, because the object referenced by `n` is in a statically unknown context.

2.4.2 Pure Expression

To focus on the essentials of the type system, we under-specify what we mean with a pure expression. All we need for the proof of the owner-as-modifier property (Theorem 2.4.7 given later) is expressed in the following assumption: pure expressions do not modify objects that exist in the prestate of the expression evaluation; formally:

Assumption 2.4.2 (Pure Expression)

$$\left.\begin{array}{l} h, {}^{r}\Gamma : {}^{s}\Gamma \ \ \mathsf{OK} \\ {}^{s}\Gamma \vdash e : \_ \\ {}^{s}\Gamma \vdash e \ \ \mathsf{pure} \\ {}^{r}\Gamma \vdash h, e \ \rightsquigarrow \ h', \_ \end{array}\right\} \Longrightarrow \ \forall \iota \in \mathrm{dom}(h)\,, f \in h(\iota)\!\downarrow_2 \,.\, h(\iota.f) = h'(\iota.f)$$

As an example that satisfies this assumption we give a strict definition of pure expressions [63]. This definition forbids all field updates, and calls to non-pure methods.

Definition 2.4.3 (Strictly Pure Expression)

$\boxed{{}^{s}\Gamma \vdash e \ \ \mathsf{strictly\ pure}}$ strictly pure expression

$$\frac{{}^{s}\Gamma \vdash \texttt{null} : \_}{{}^{s}\Gamma \vdash \texttt{null} \ \ \mathsf{strictly\ pure}}\ \text{SP\_\_NULL} \qquad \frac{{}^{s}\Gamma \vdash x : \_}{{}^{s}\Gamma \vdash x \ \ \mathsf{strictly\ pure}}\ \text{SP\_\_VAR}$$

$$\frac{{}^{s}\Gamma \vdash \texttt{new}\ {}^{s}T\texttt{()} : \_}{{}^{s}\Gamma \vdash \texttt{new}\ {}^{s}T\texttt{()} \ \ \mathsf{strictly\ pure}}\ \text{SP\_\_NEW} \qquad \frac{\begin{array}{l}{}^{s}\Gamma \vdash \texttt{(}{}^{s}T\texttt{)}\ e : \_ \\ {}^{s}\Gamma \vdash e \ \ \mathsf{strictly\ pure}\end{array}}{{}^{s}\Gamma \vdash \texttt{(}{}^{s}T\texttt{)}\ e \ \ \mathsf{strictly\ pure}}\ \text{SP\_\_CAST}$$

$$\frac{\begin{array}{l}{}^{s}\Gamma \vdash e_0.f : \_ \\ {}^{s}\Gamma \vdash e_0 \ \ \mathsf{strictly\ pure}\end{array}}{{}^{s}\Gamma \vdash e_0.f \ \ \mathsf{strictly\ pure}}\ \text{SP\_\_READ}$$

$$\frac{\begin{array}{l}{}^{s}\Gamma \vdash e_0\ .\ m\texttt{<}\overline{{}^{s}T}\texttt{>(}\overline{e}\texttt{)} : \_ \\ {}^{s}\Gamma \vdash e_0 : {}^{s}N_0 \qquad {}^{s}\Gamma \vdash e_0 \ \ \mathsf{strictly\ pure} \qquad {}^{s}\Gamma \vdash \overline{e} \ \ \mathsf{strictly\ pure} \\ \mathrm{MSig}\big({}^{s}N_0, m, \overline{{}^{s}T}\big) \ = \texttt{pure <\_> \_ }m\texttt{(\_)}\end{array}}{{}^{s}\Gamma \vdash e_0\ .\ m\texttt{<}\overline{{}^{s}T}\texttt{>(}\overline{e}\texttt{)} \ \ \mathsf{strictly\ pure}}\ \text{SP\_\_CALL}$$

Also approaches that allow the modification of newly created objects [168] fulfill this assumption.

2.4.3 Encapsulated Method Declaration

For an encapsulated method, we require that for a pure method the method body is a pure expression and that for a non-pure method the method body is an encapsulated expression.

Definition 2.4.4 (Encapsulated Method Declaration)

$\boxed{{}^{s}\Gamma, C \vdash md \ \ \mathsf{enc}}$ encapsulated method declaration

$$\frac{\begin{array}{l}{}^{s}\Gamma, C \vdash p\ \texttt{<}\overline{X_l\ \texttt{extends}\ {}^{s}N_l}\texttt{>}\ {}^{s}T\ m\texttt{(}\overline{{}^{s}T_q\ pid}\texttt{)}\ \texttt{\{}\ e\ \texttt{\}} \ \ \mathsf{OK} \\ {}^{s}\Gamma = \big\{\overline{X'_k \mapsto {}^{s}N'_k}\ ;\ \texttt{this} \mapsto \texttt{self}\ C\texttt{<}\overline{X'_k}\texttt{>}, \_\big\} \\ {}^{s}\Gamma' = \big\{\overline{X'_k \mapsto {}^{s}N'_k},\ \overline{X_l \mapsto {}^{s}N_l}\ ;\ \texttt{this} \mapsto \texttt{self}\ C\texttt{<}\overline{X'_k}\texttt{>},\ \overline{pid \mapsto {}^{s}T_q}\big\} \\ \big(p = \texttt{pure} \implies {}^{s}\Gamma' \vdash e \ \ \mathsf{pure}\big) \qquad \big(p = \texttt{impure} \implies {}^{s}\Gamma' \vdash e \ \ \mathsf{enc}\big)\end{array}}{{}^{s}\Gamma, C \vdash p\ \texttt{<}\overline{X_l\ \texttt{extends}\ {}^{s}N_l}\texttt{>}\ {}^{s}T\ m\texttt{(}\overline{{}^{s}T_q\ pid}\texttt{)}\ \texttt{\{}\ e\ \texttt{\}} \ \ \mathsf{enc}}\ \text{EMD\_\_DEF}$$

The first requirement checks that the method is a topologically well-formed method. We then determine the static environments ${}^{s}\Gamma$ and ${}^{s}\Gamma'$ that we need to check the method body expression. The construction of these environments corresponds to the topological well-formed method judgment (Def. 2.3.16). Finally, a pure method is required to have an expression as method body that fulfills our assumption of a pure expression (Assumption 2.4.2) and a non-pure method needs a method body that fulfills the encapsulated expression judgment (Def. 2.4.1).

2.4.4 Encapsulated Class and Program Declaration

The final two judgments define encapsulated class, Cls `enc`, and encapsulated program, $\vdash P$ `enc`. They simply propagate the checks to the lower levels.

Definition 2.4.5 (Encapsulated Class Declaration)

$\boxed{Cls \texttt{ enc}}$ encapsulated class declaration

$$\frac{\texttt{class } Cid\texttt{<}\overline{X_k \texttt{ extends } {}^{s}N_k}\texttt{> extends } C\texttt{<}\overline{{}^{s}T}\texttt{> \{ } \overline{fd}\ \overline{md} \texttt{ \}} \quad \mathsf{OK} \qquad {}^{s}\Gamma = \left\{\overline{X_k \mapsto {}^{s}N_k} \texttt{ ; this} \mapsto \texttt{self } Cid\texttt{<}\overline{X_k}\texttt{>}, \_\right\} \qquad {}^{s}\Gamma, Cid \vdash \overline{md} \texttt{ enc}}{\texttt{class } Cid\texttt{<}\overline{X_k \texttt{ extends } {}^{s}N_k}\texttt{> extends } C\texttt{<}\overline{{}^{s}T}\texttt{> \{ } \overline{fd}\ \overline{md} \texttt{ \}} \texttt{ enc}}\ \text{EC\_\_DEF}$$

$$\frac{}{\texttt{class Object \{\} enc}}\ \text{EC\_\_OBJECT}$$

A class declaration is correctly encapsulated, if it fulfills the topological rules of well formedness and all methods are encapsulated in the corresponding environment ${}^{s}\Gamma$ (the construction of ${}^{s}\Gamma$ again corresponds to the construction of the topological judgment, this time from Def. 2.3.14). Class `Object` is always correctly encapsulated.

Definition 2.4.6 (Encapsulated Program Declaration)

$\boxed{\vdash P \texttt{ enc}}$ encapsulated program

$$\frac{\vdash \overline{Cls},\ C,\ e \quad \mathsf{OK} \qquad \overline{Cls_k \texttt{ enc}} \qquad \{\emptyset \texttt{ ; this} \mapsto \texttt{self } C\texttt{<>}\} \vdash e \texttt{ enc}}{\vdash \overline{Cls},\ C,\ e \texttt{ enc}}\ \text{EP\_\_DEF}$$

A program is encapsulated, if the program is topologically well formed, all classes are correctly encapsulated, and the main expression is correctly encapsulated.

2.4.5 Examples

2.4.5.1 Purity Examples

Consider the following examples about method purity. Method `getData` below is pure, in both the weak (Assumption 2.4.2) and strict definitions (Def. 2.4.3):

```
class C {
  rep Data f;

  pure rep Data getData() {
    return f;
  }
}
```

The method simply returns a reference to the internal state and does not modify any state.

On the other hand, consider:

```
class D {
  pure rep Data computeData() {
    rep Data x = new rep Data();
    x.addInfo(...);
    return x;
  }
}
```

assuming some non-pure method `addInfo` in class `Data`. Method `computeData` is pure according to the weak definition of purity from Assumption 2.4.2, because it does not modify objects that exist in the pre-state of the method call. It creates a new object, modifies only this new object, and finally returns it. However, the strict definition from Def. 2.4.3 forbids method `computeData`, because all non-pure method calls are forbidden.

Finally, the following method is non-pure by both definitions:

```
class E {
  rep Data f;
  rep Data cachedData() {
    if( f==null ) {
      f = new rep Data();
      f.addInfo(...);
    }
    return f;
  }
}
```

The modification of field `f` violates both purity definitions. However, caching and lazy initialization are common in query methods. The current research on observational purity [54, 146, 18] tries to remedy this problem, but the relation to the owner-as-modifier discipline has not been investigated yet.

2.4.5.2 Encapsulation Examples

The following examples illustrate the encapsulation judgments. The code fragment below is well encapsulated:

```
peer Data dp = ...;
dp.addInfo(...);
dp.count = 4;

rep Data dr = ...;
dr.addInfo(...);

any Data da = ...;
any Info ia = da.getInfo();
int count = da.count;
```

Calls of the non-pure method `addInfo` are on receivers with a `peer` or `rep` type; also, the field update is on a receiver with a `peer` type. We have no static knowledge of the ownership of `da`; therefore, only the `pure` method `getInfo` can be called on it and the field can only be read.

Note that the encapsulation judgment is concerned only with the main modifier of the receiver type. For example, consider:

```
peer List<any Data> pla = new peer List<any Data>();
pla.add(new peer Data());
```

This code is well encapsulated. We know that the list referenced by `pla` has the same owner as the current object and the current object can therefore modify the object referenced by `pla`. We have no static knowledge of the ownership of the type arguments, but this is not needed to enforce the owner-as-modifier discipline.

Static types always represent static approximations of the runtime behavior. Consider:

```
any Data d;
if (...) {
  d = new rep Data();
} else {
  d = new peer Data();
}
d.addInfo(...);
```

This code is not well encapsulated. At runtime, variable `d` will always reference an object that is either owned by the current object or by the owner of the current object. However, the static type loses this ownership information and the encapsulation system rejects the call to the non-pure method `addInfo`. The code must be rewritten to retain the static knowledge.

2.4.6 Properties of the Encapsulation System

The owner-as-modifier discipline is expressed by the following theorem. The evaluation of an encapsulated expression e in an encapsulated program P and a well-formed environment can only modify those objects that are (transitively) owned by the owner of `this`.

Theorem 2.4.7 (Owner-as-Modifier)

$$\left.\begin{array}{l} \vdash P \ \ \mathsf{enc} \\ {}^{s}\Gamma \vdash e \ \ \mathsf{enc} \\ h, {}^{r}\Gamma : {}^{s}\Gamma \ \ \mathsf{OK} \\ {}^{r}\Gamma \vdash h, e \ \rightsquigarrow \ h', \_ \end{array}\right\} \Longrightarrow \begin{array}{l} \forall \iota \in \mathrm{dom}(h)\,.\, \forall f \in \mathrm{dom}(h(\iota)\downarrow_2)\,. \\ \quad h(\iota\,.\,f) = h'(\iota\,.\,f) \ \vee \\ \quad \mathrm{owner}(h, {}^{r}\Gamma(\texttt{this})) \in \mathrm{owners}(h, \iota) \end{array}$$

The proof of Theorem 2.4.7 runs by rule induction on the operational semantics. The interesting cases are field updates and calls of non-pure methods. In both cases, the encapsulation rules enforce that the receiver expression does not have the main modifier `lost` or `any`. That is, the receiver object is owned by `this` or the owner of `this`. The case for pure method calls relies only on Assumption 2.4.2 and not on a more restrictive definition of purity.

2.5 Discussion

This section discusses interesting aspects of the formalization in more detail and compares the previous formalization of GUT [60] with our current work using an example. To distinguish the two systems, we call the system from [60] GUT_1 and the current formalization GUT_2.

2.5.1 Reasonable Programs

2.5.1.1 Motivation

The goal of our formalization of Generic Universe Types is to impose the minimal set of restrictions necessary for soundness. This highlights the necessity of each restriction and makes the formalization elegant, but on the other hand makes the proofs more involved and allows programs which are sound, but not meaningful for programmers to write.

For instance, we define a well-formed static type judgment (Def. 2.3.11) and a strictly well-formed static type judgment (Def. 2.3.12). Using only the strict judgment would be overly restrictive, as it would forbid the appearance of `lost` in the types of subexpressions. Consider the following example:

```
class L<X extends peer Object> {
  X getFirst() {...}
}
class C {
  peer L<peer D> f;
}
class E {
  void m(any C p) {
    any D x = p.f.getFirst();
  }
}
```

The type of field `f` is strictly well formed; however, the type of the field access `p.f` is not strictly well formed, because the type after viewpoint adaptation is `lost L<lost D>` and the upper bound after viewpoint adaptation contains `lost`. However, we can consider the type well formed and we can call method `getFirst`, because it does not contain `lost` in the parameter types. If we were to always enforce strict well-formedness, we would forbid all access to objects where we only have partial knowledge of the ownership.

On the other hand, our well-formed type judgment only considers type soundness and does allow types which are not meaningful for a programmer to write. For example, consider this code:

```
class C<X extends rep D> {
  void add(X p) {...}
}
class E {
  void m(peer C<peer D> p) {
    p.add(new peer D());
  }
}
```

The type of parameter `p` is well formed, but not strictly well formed, because the viewpoint adapted upper bound of class `C` contains `lost`. This example is sound, because there will never be a call of method `m` with an argument other than $\texttt{null}_a$. It would be useful to warn the programmer that such programs are sound, but probably not what was intended.

In work on non-generic Universe Types [55], the syntax forbids the use of `self` and `lost` by the programmer and only allows these two modifier in derivations. However, in the generic case this restriction is not enough. In the example above, the type of parameter `p` does not contain `self` or `lost`, but still is not meaningful. In the generic system we need a stronger check that also ensures that the viewpoint adapted upper bounds are meaningful.

As an aside, let us consider a similar issue with wildcards in Java 5. The bound of a wildcard can be a supertype of the declared bound of the corresponding type variable. Consider this example:

```
class A {}
class B extends A {}

class C<X extends B> {}

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

This is a valid Java 5 program and field `f` will always reference an object that fulfills the intersection of the declared bound and the wildcard bound.

Now consider this slight extension:

```
class A {}
class B extends A {
  void crash() { }
}

class C<X extends B, Y extends X> {
  void m(Y y) { y.crash(); }
}

public class D {
  void foo(C<? extends A, A> p) {
    p.m(new A());
  }
}
```

Also this program type-checks as valid Java 5 using Sun javac 1.5 and 1.6. However, if method `foo` could be called with an instance of class `C` as argument, we would receive a "Message Not Understood" exception, as the argument to method `m` is an instance of class `A`, which does not provide method `crash`. However, method `foo` can never be called with an argument other than $\texttt{null}_a$, because whenever an instance of class `C` is created, the declared bound is checked and the parameter type is not a creatable type.

In work on non-generic Universe Types [34], we compared the `lost` and `any` ownership modifiers to existential quantification in a parametric ownership type system and showed their equivalence. It will be interesting further work to see how wildcards can be combined with Generic Universe Types.

In the following subsections, we will discuss interesting examples, illustrate the soundness of the weak rules, and motivate how stricter rules would aid programmers. We then present our "reasonable" judgments, which still allow meaningful programs but forbid unreasonable cases.

2.5.1.2 Examples

Use of `self` in declared types. We allow the `self` modifier to be used in field and method parameter types; however, such fields can only be updated on `this` and such methods are only callable on `this` and the right-hand side, respectively the argument in the method call, has to be the `this` object. The return type of a method could use `self` and the method would still be callable on something other than `this`, but the viewpoint-adapted return type would be `lost`, if the receiver is not `this`.

In the proof of soundness we need to show that, if a value can be typed with `self`, it corresponds to the current object in the runtime environment. Because of subsumption, we need Lemma A.1.21 to show that if we showed the absence of `self` in a subtype, also the supertype does not contain `self`. Also, in Def. 2.3.25 there is no way to ensure that a type argument is actually a subtype of a `self` upper bound, as there is no object that can be compared with the current object. We would need to make the runtime model more complicated to support this and decided to stay with the current runtime model.

The use of `self` in the superclass instantiation and in object creations is forbidden by the strict well-formedness check of the type. Finally, the type of a cast could use `self`; such a use of `self` can be replaced with a comparison against the current object `this`.

In conclusion, for programmers the possibility to use `self` in a program does not add a lot of expressiveness. We will therefore forbid all explicit uses of `self` in reasonable programs. This will also make it possible to simplify the "assigning a static type to a value" judgment (Def. 2.2.15) by not checking whether the object is the current object `this`, which would simplify casts.

Use of `rep` in declared upper bounds. A class C's upper bounds express ownership relative to the current C instance. If such an upper bound contains a `rep` modifier, clients of C cannot instantiate C. The ownership modifiers of an actual type argument are relative to the client's viewpoint. From this viewpoint, none of the ownership modifiers expresses that an object is owned by the C instance. We do not forbid a class from having a `rep` modifier in an upper bound, but ensure that when instantiating an object, the upper bounds do not contain `lost`. It is still possible to have subclasses of classes with `rep` in an upper bound. This allows class `C` from Fig. 2.10 and class `D` is a valid subclass. Class `C` will never be usable in an object creation expression and a corresponding warning could be issued for the class declaration.

In GUT_1 we had to forbid classes with upper bounds that contain a `rep` modifier, because there was no possibility to distinguish between an upper bound that was declared to be `any` and one that is `any` after viewpoint adaptation. In GUT_2 we make the handling of this case more flexible and uniform and allow `rep` in upper bounds.

Use of `rep` in superclass instantiations. A superclass can be instantiated with `rep` types, but the use of such classes is limited, as clients of the class can only refer to the superclass with `lost` type arguments. For example, consider the decorator pattern introduced in Fig. 2.4 and the following decoration class:

```
class MyDecoration implements Decoration<rep Data> {
  rep Data f;
  void set(rep Data d) {
    f = d;
  }
}
```

We consider this class well formed, but it cannot be used together with class `Decorator`. Type variable `O` of method `decorate` in class `Decorator` has `peer Decoration<V>` as upper bound. A `peer MyDecoration` is only a subtype of `peer Decoration<lost Data>`, therefore mandating that type variable V has type `lost Data`. However, the type rules forbid method calls with type arguments that contain `lost` ownership modifiers.

Forbidding the use of `rep` in superclass instantiations would also forbid valid uses; therefore, we do not constrain this case further.

Superclass instantiations are strictly well formed. The instantiation of a superclass can only use strictly well-formed types as type arguments. There are two places where static types are dynamized into corresponding runtime types, object creation and the method type arguments in a method call. In both places we need to ensure that the resulting runtime type is a strictly well-formed runtime type and therefore need to make sure that the corresponding static type is strictly well formed.

The instantiation of the superclass needs to use strictly well-formed type arguments because a method in the superclass could use the type variable in an object creation or as a method type argument. For example, assume that we would not check for strictly well formed superclass instantiation and consider the following code:

```
class L<Z extends rep Object> {}

class B<X extends peer Object> {
   X f;
   X foo() {
     this.f = new X();
     return this.f;
   }
}
class C extends B<peer L<peer Object>> {}

class D {
  peer C x = new peer C();
  peer L<peer Object> o = x.foo();
}
```

The type `peer L<peer Object>` is well formed, but not strictly well formed, because the upper bound after viewpoint adaptation contains `lost`. In class `D` we could instantiate an instance of class `C`, because `peer C` is a strictly well-formed type. Then the call `x.foo` would create a new object that would not respect the upper bound of type variable `Z` of class `L`. Ensuring that the superclass instantiation uses strictly well-formed type arguments forbids class `C` and prevents this problem.

The same issue arises with method type arguments. Consider this example:

```
class C {
   <Y extends peer Object> Y bar() {
      return new Y();
   }
}

class D {
  peer C x = new peer C();
  peer L<peer Object> o = x.bar<peer L<peer Object>>();
}
```

If we would not check for strict well-formedness of the method type arguments, we would consider the above code well formed, but it would result in an ill-formed heap. By enforcing strict well-formedness of the method type arguments we prevent this problem.

The topological system already enforces strict well-formedness and we therefore do not need additional checks for this case.

Use of `lost` in declared types. The well-formed class and method rules allow most declared types to contain `lost`. However, if a declared type contains `lost`, the type after viewpoint adaptation also contains `lost` (see Lemma A.1.4). Therefore:

Object creation of non-variable type: the type has to be strictly well-formed, which checks for `lost` in the type and in the viewpoint-adapted upper bounds. Therefore, a class with `lost` in an upper bound can never be instantiated.

Superclass instantiations: the well-formed class judgment (Def. 2.3.14) checks that the type arguments to the superclass are strict subtypes of their respective upper bounds, and thereby that the upper bounds do not contain `lost`. Therefore, a class with `lost` in an upper bound can never be subclassed.

Field updates: the viewpoint-adapted field type must not contain `lost`. Therefore, a field with `lost` in its declared type can never be updated and will always be $\texttt{null}_a$.

Method calls: the method type arguments, the viewpoint-adapted upper bounds of the method type variables, and the viewpoint-adapted parameter types must not contain `lost`. Therefore, a method with `lost` in any of these declared types can never be called.

The only places where `lost` could be used in a type is in a cast, in a method return type, and for local variables. Therefore, we only allow the use of `lost` in these types and provide additional checks to ensure that all declared types are reasonable.

Note that Def. 2.3.25 and Def. 2.3.22 use dyn with an empty substitution for `lost` when upper bounds are dynamized. The static checks ensure that there is no `lost` in the corresponding types.

Expressiveness without `lost` ownership. How can we declare a class, method, or field that references some arbitrary kind of `List`, regardless of the ownership of the type argument? In GUT_1 the type `any List<any Object>` can be used. As explained in Sec. 2.3.8, the type `any List<lost Object>` is the most general `List` type in GUT_2, but as this most general type contains `lost`, we cannot use it as a declared type and still instantiate the class, call the method, or update the field. Consider the following class that can be parameterized by an arbitrary `List`:

```
class List<X extends any Object> {
  X get();
  void add(X p);
}

class C<Y extends any List<lost Object>> {
  Y store;

  void init(Y p) {
    store = p;
  }

  peer List<lost Object> sort() { ok; }
  void addElem(Object o) {
    store.add(o); // forbidden
  }
}
```

Then types like

```
peer C<peer List<peer Data>> f;
peer C<any List<rep Data>> g;
```

could be used to manage different kinds of lists. However, as the upper bound of `C` contains `lost` we can never create instances of class `C`. For upper bounds of class and method type variables and method parameter and return types, one can simply introduce another type variable that varies over the element type:

```
class C<Z extends any Object, Y extends any List<Z>> {
  ...
}
```

which is the same approach as one would use in a Java 5 program. Then the types for references `f` and `g` from above can be expressed as:

```
peer C<peer Data, peer List<peer Data>> f;
peer C<rep Data, any List<rep Data>> g;
```

and corresponding instances of class `C` are creatable.

This is a workable solution, but it introduces an extra, redundant type variable. Moreover, for a field type we cannot introduce a separate type variable to vary over the element type. The solution to both issues in Java 5 is to use a wildcard as element type in field and method parameter and return types.

We think that there are three possible solutions to this for Generic Universe Types:

1. **Use-site variance:** wildcards as found in Java 5:

   ```
   any List<? extends any Object> l;
   ```

   The wildcard abstracts away the concrete ownership of the list elements. Unlike `lost`, a wildcard can be used in a method signature and field type and the method is still callable and the field still updatable. However, the uses of a type that uses wildcards are limited by the upper or lower bound of the wildcard.

2. **Declaration site variance:** as found, e.g., in Scala:

   ```
   class List<+X extends any Object> {...}
   ```

   The uses of type variable `X` within the class declaration are checked to conform to its variance annotation. This again allows us to abstract from unknown ownership information in a class declaration and eases the use of the class by clients.

3. **Additional `some` ownership modifier:** the problem with using `lost` is that one cannot distinguish between the loss of ownership that results from viewpoint adaptation and intended covariance that is declared in the field or parameter type. A new modifier `some` could be introduced that basically behaves like an upper bounded wildcard that only varies over the ownership information. Viewpoint adaptation changes an ownership modifier to `lost`, when a concrete main modifier is needed.

   For example we could imagine:

   ```
   class C {
     any List<some Data> l;

     void m(any C o) {
       o.l = new peer List<peer Data>(); // OK
       int x = o.l.length();             // OK
       any Data h = o.l.head();          // OK
       o.l.add( new peer Data() ); // Error: capture
     }
   }
   ```

   The use of `some` in the type of `l` allows us to distinguish between a deliberate abstraction over ownership and the inadvertent loss of ownership by viewpoint adaptation. We can update field `l` and query information from the referenced object, but cannot modify it.

However, in all three solutions one still needs the `lost` modifier to distinguish the loss of ownership by viewpoint adaptation from intended variance. For example, consider this class:

```
class E {
  peer List<rep Data> l;
}
```

and a reference `x` of type `peer E`. The type of `x.l` is `peer List<lost Data>`; it would not be sound to use `peer List<? extends any Data>` or `peer List<some Data>` as the type of `x.l`, as we could update the field `l` without respecting the declared field type. If only declaration site variance is supported, the field access would be illegal, if class `List` is not using the correct variance annotation, even though save read access would be possible using `lost`.

In GUT_2 we leave the exploration of these options as future work. `lost` is used as internal modifier for derived types to highlight the loss of ownership information that might occur in viewpoint adaptation and to cleanly separate the topological system from the encapsulation system.

Method overriding. The method overriding rule (Def. 2.3.17) ensures that the method selected at runtime is compatible with the method arguments.

As an example, consider the following classes:

```
class Super<X> {
  void m(X p) {
  }
}

class Sub1 extends Super<peer Data> {
  void m(peer Data p) { ... }
}

class Sub2 extends Super<rep Data> {
  void m(rep Data p) { ... }
}

class Sub3 extends Super<rep Data> {
  void m(peer Data p) { ... }
}
```

According to Def. 2.3.17, classes `Sub1` and `Sub2` correctly override method `m`, as the method parameter is consistent with the instantiation of the superclass. On the other hand, `Sub3` does not use the correct parameter type and is rejected.

It is important to note the relation between subtyping and method selection. In the above example, we can derive `peer Sub1` <: `peer Super<peer Data>`. The look-up of the method signature in the supertype is consistent with the signature in the subtype and any call that typechecks on the supertype will also typecheck on the subtype. We also have `any Sub1` <: `any Super<lost Data>`; for both the sub- and the supertype, the method signature look-up yields `lost Data` as parameter type and the method call rule (TR_CALL in Def. 2.3.13) forbids an invocation of the method. It is important that `any Sub1` <: `any Super<peer Data>` does not hold; otherwise, we could provide a `peer` reference as argument to a call on the supertype, even though the object referenced at runtime does not expect a `peer` argument. For class `Sub2`, we can deduce `peer Sub2` <: `peer Super<lost Data>` and again, the method signatures look-up for sub- and supertype match.

Pure methods. Pure methods are intended to be callable on any object to query their state, e.g., in specifications. If a pure method uses a type variable as parameter type, the type might use `lost` as type argument (e.g., as the result of viewpoint adaptation) and the method would not be callable any more. Also, non-variable types that change under viewpoint adaptation with `any` would not be callable on an arbitrary receiver.

The soundness of GUT_1 depended on requiring that pure methods use only `any` in their signature, because this system could not distinguish between loss of ownership information from viewpoint adaptation and a declared type that only uses `any`. In GUT_2 the topological system enforces the constraints that are needed for soundness, regardless of whether the method is pure or non-pure. If the method parameter types or upper bounds after viewpoint adaptation contain `lost`, the call has to be forbidden. However, the goal of the encapsulation system is to use pure methods as queries that can be used on any receiver type. Therefore, additional checks on pure methods can ensure that no ownership information is lost under viewpoint adaptation.

For example, consider:

```
class C<X extends any Data> {
  pure boolean foo(X p) {...}
  pure boolean bar(peer Object p) {...}
}

any C<lost Data> x = ...;
if( x.foo(blub) ) ... // illegal, signature contains lost
if( x.bar(blub) ) ... // illegal, signature contains lost
```

The topological system considers class `C` well formed and forbidds both calls on receiver `x`, because the viewpoint adapted parameter type contains `lost`. The example is more flexible, if it is rewritten as:

```
class C<X extends any Data> {
  pure boolean foo(any Data p) {...}
  pure boolean bar(any Object p) {...}
}

any C<lost Data> x = ...;
if( x.foo(blub) ) ... // ok
if( x.bar(blub) ) ... // ok
```

That is, appearances of type variables in the signature of pure methods are replaced by (supertypes of) their upper bounds and non-variable types do not change under viewpoint adaptation, i.e., they only use the `any` modifier.

2.5.1.3 Reasonable Well-formedness Judgments

In the following we discuss judgments that check the static types in a program to be reasonable, i.e., that forbid more types than Def. 2.3.11, but are more flexible than Def. 2.3.12. As these judgments are only used to reject unreasonable programs, we do not prove properties of these judgments. Possible properties that could be investigated are 1.) no useful types, which could reference well-formed objects at runtime, are forbidden, 2.) all unreasonable types are forbidden, i.e., if a type is reasonable, it can reference a well-formed object. We first present a judgment that checks a static type to be reasonable and then judgments to propagate the checks to methods, classes, and programs.

Reasonable Static Type. Consider this code:

```
class C<X extends rep Data> { }

class D {
  peer C<peer Data> foo() {...}
  peer C<lost Data> bar() {...}
}
```

Both return types are well formed according to Def. 2.3.11. The simplest solution to deciding what a reasonable type is, would be to use the strict well-formedness judgment (Def. 2.3.12) on all declared types in a program. This would forbid both return types above. In the following, we define a reasonable type judgment that forbids the return type of `foo`, but allows the return type of `bar`. The return type of `foo`, `peer C<peer Data>` is unreasonable, because no object of this type can ever exist. On the other hand, the return type of `bar`, `peer C<peer Data>`, is reasonable, because there exists an object that can be referenced by this type. However, because such reasonable types might still contain `lost` ownership information, they are not usable in all positions. We separately forbid the use of `lost` in method parameter types and upper bounds, class upper bounds, and types in casts. One of the three solutions discussed above (wildcards, variance annotations, or a `some` ownership modifier) needs to be used additionally to make types usable in all positions. In our example, we could imagine:

```
class E {
  void blub(peer C<? extends any Data> p) {...}
  void blob(peer C<some Data> q) {...}
}
```

as valid method signatures; the signature of `blub` allows a subclass of `Data` with arbitrary ownership as type argument, whereas `blob` allows only `Data` instances with arbitrary ownership as type argument.

The judgment ${}^s\Gamma \vdash {}^sT$ prg OK expresses that type ${}^sT$ is a reasonable type in type environment ${}^s\Gamma$. Type variables are reasonable, if they are contained in the type environment (PWFT_VAR). A non-variable type $u\ C\texttt{<}\overline{{}^sT_k}\texttt{>}$ is reasonable (PWFT_NVAR) if its type arguments $\overline{{}^sT_k}$ are reasonable, each type argument is a subtype of the corresponding upper bound, using the well-formed type argument judgment (see Def. 2.5.2 below), and the type does not contain `self`.

Definition 2.5.1 (Reasonable Static Type)

${}^s\Gamma \vdash {}^sT$ prg OK — reasonable static type

$$\frac{X \in {}^s\Gamma}{{}^s\Gamma \vdash X \text{ prg OK}}\ \text{PWFT\_VAR}$$

$$\frac{\begin{array}{cc} {}^s\Gamma \vdash \overline{{}^sT} \text{ prg OK} & \texttt{self} \notin u\ C\texttt{<}\overline{{}^sT}\texttt{>} \\ \text{ClassBnds}(C) = \overline{{}^sN} & u\ C\texttt{<}\overline{{}^sT}\texttt{>} \triangleright \overline{{}^sN} = \overline{{}^sN'} \\ {}^s\Gamma, u\ C\texttt{<}\overline{{}^sT}\texttt{>} \vdash \overline{{}^sT} <: \overline{{}^sN}, \overline{{}^sN'} & \end{array}}{{}^s\Gamma \vdash u\ C\texttt{<}\overline{{}^sT}\texttt{>} \text{ prg OK}}\ \text{PWFT\_NVAR}$$

A type argument ${}^sT$ of non-variable type $u\ C\texttt{<}\overline{{}^sT}\texttt{>}$ is valid with regard to the un-adapted upper bound ${}^sN$ and the upper bound after viewpoint adaptation ${}^sN'$, written as
${}^s\Gamma, u\ C\texttt{<}\overline{{}^sT}\texttt{>} \vdash {}^sT <: {}^sN, {}^sN'$.

The viewpoint adaptation is necessary because the type arguments describe ownership relative to the `this` object where $u\ C\texttt{<}\overline{{}^sT}\texttt{>}$ is used, whereas the upper bounds are relative to the object

of type $u\ C\texttt{<}\overline{{}^{s}T}\texttt{>}$. The type argument ${}^{s}T$ has to be a subtype of the viewpoint adapted upper bound ${}^{s}N'$ and we construct a static environment ${}^{s}\Gamma'$, which uses the type variables and upper bounds of class C. Finally, the judgment has to show that there exists a type ${}^{s}T_0$, which after viewpoint adaptation from $u\ C\texttt{<}\overline{{}^{s}T}\texttt{>}$ equals to ${}^{s}T$ and is a subtype of the declared upper bound ${}^{s}N$.

Definition 2.5.2 (Reasonable Static Type Argument)

$\boxed{{}^{s}\Gamma, {}^{s}N'' \vdash {}^{s}T <: {}^{s}N, {}^{s}N'}$ reasonable static type argument

$$\frac{\begin{array}{l}{}^{s}\Gamma \vdash {}^{s}T <:\ {}^{s}N' \\ \quad \text{ClassDom}(C) = \overline{X_k} \qquad \text{ClassBnds}(C) = \overline{{}^{s}N_k} \\ {}^{s}\Gamma' = \left\{\overline{X_k \mapsto {}^{s}N_k}\ ;\ \texttt{this} \mapsto \texttt{self}\ C\texttt{<}\overline{X_k}\texttt{>}, \_\right\} \\ \exists\, {}^{s}T_0.\ \left(u\ C\texttt{<}\overline{{}^{s}T}\texttt{>} \rhd {}^{s}T_0\ =\ {}^{s}T\ \wedge\ {}^{s}\Gamma' \vdash {}^{s}T_0 <:\ {}^{s}N\right)\end{array}}{{}^{s}\Gamma, u\ C\texttt{<}\overline{{}^{s}T}\texttt{>} \vdash {}^{s}T <: {}^{s}N, {}^{s}N'}\ \text{PWFTA\_\_DEF}$$

Given class `C` from before:

```
class C<X extends rep Data> { }
```

Type `peer C<peer Data>` is not reasonable, because there is no type ${}^{s}T_0$ that after adaptation from `peer C<peer Data>` equals to `peer Data` which is a subtype of the declared upper bound `rep Data`. On the other hand, `peer C<lost Data>` is reasonable, because there exists the type `rep Data`, which is a subtype of the declared upper bound and after viewpoint adaptation is equal to `lost Data`.

Why do we need to check for subtyping twice, once for the type argument and the viewpoint-adapted upper bound and once with type ${}^{s}T_0$ and the declared upper bound? Consider this use of class `C`:

```
class D<Y extends peer Object> {
  peer C<Y> f;
}
```

We can find a type ${}^{s}T_0$ that is a subtype of the declared upper bound: the type variable `X`; it is by definition a subtype of the upper bound and after viewpoint adaptation with `peer C<Y>` it equals to `Y`. However, the viewpoint adapted upper bound for class `C` is `lost Data` and `Y` is not a subtype of `lost Data`. Therefore, without both checks we would not ensure that types are correct.

Other Reasonable Well-formedness Conditions. The definitions for classes, fields, methods, and programs check all occurrences of types literally used in the program to conform to the reasonable type judgment and to forbid `lost`, if its use would restrict the applicability of the method.

A class is reasonable, if it is well formed, the types used as upper bounds of the class type variables are reasonable types and do not use `lost`, and the field and method declarations are reasonable.

Definition 2.5.3 (Reasonable Class Declaration)

$\boxed{Cls\ \ \mathsf{prg\ OK}}$ reasonable class declaration

$$\frac{\begin{array}{l}\texttt{class}\ Cid\texttt{<}\overline{X_k\ \texttt{extends}\ {}^{s}N_k}\texttt{>}\ \texttt{extends}\ C\texttt{<}\overline{{}^{s}T}\texttt{>}\ \texttt{\{}\ \overline{fd}\ \overline{md}\ \texttt{\}}\ \ \mathsf{OK} \\ {}^{s}\Gamma = \left\{\overline{X_k \mapsto {}^{s}N_k}\ ;\ \texttt{this} \mapsto \texttt{self}\ Cid\texttt{<}\overline{X_k}\texttt{>}, \_\right\} \\ \quad {}^{s}\Gamma \vdash \overline{{}^{s}N_k}\ \ \mathsf{prg\ OK} \qquad \texttt{lost} \notin \overline{{}^{s}N_k} \\ \quad {}^{s}\Gamma \vdash \overline{fd}\ \ \mathsf{prg\ OK} \qquad {}^{s}\Gamma, Cid \vdash \overline{md}\ \ \mathsf{prg\ OK}\end{array}}{\texttt{class}\ Cid\texttt{<}\overline{X_k\ \texttt{extends}\ {}^{s}N_k}\texttt{>}\ \texttt{extends}\ C\texttt{<}\overline{{}^{s}T}\texttt{>}\ \texttt{\{}\ \overline{fd}\ \overline{md}\ \texttt{\}}\ \ \mathsf{prg\ OK}}\ \text{PC\_\_DEF}$$

$$\frac{}{\texttt{class Object \{\}}\ \ \mathsf{prg\ OK}}\ \text{PC\_\_OBJECT}$$

Note that the instantiation of the superclass is checked to use strictly well-formed types by the topological judgment.

A field declaration is reasonable, if the field type is reasonable and `lost` is not contained in the type.

Definition 2.5.4 (Reasonable Field Declaration)

$\boxed{{}^s\Gamma \vdash {}^sT\ f\texttt{;}\ \ \mathsf{prg\ OK}}$ reasonable field declaration

$$\frac{{}^s\Gamma \vdash {}^sT\ \ \mathsf{prg\ OK} \qquad \texttt{lost} \notin {}^sT}{{}^s\Gamma \vdash {}^sT\ f\texttt{;}\ \ \mathsf{prg\ OK}}\ \text{PFD\_\_DEF}$$

A method declaration is reasonable, if the types used as upper bounds of the method type variables, the return type, and the parameter types are all reasonable types, `lost` is not contained in the method upper bounds and parameter types, and the method body is reasonable in a corresponding new environment ${}^s\Gamma'$.

Definition 2.5.5 (Reasonable Method Declaration)

$\boxed{{}^s\Gamma, C \vdash md\ \ \mathsf{prg\ OK}}$ reasonable method declaration

$$\frac{\begin{array}{l}{}^s\Gamma, C \vdash p\ \texttt{<}\overline{X_l\ \texttt{extends}\ {}^sN_l}\texttt{>}\ {}^sT\ m\texttt{(}\overline{{}^sT_q\ pid}\texttt{)}\ \texttt{\{}\ e\ \texttt{\}}\ \ \mathsf{OK}\\ {}^s\Gamma = \left\{\overline{X'_k \mapsto {}^sN'_k}\ ;\ \texttt{this} \mapsto \texttt{self}\ C\texttt{<}\overline{X'_k}\texttt{>}, \_\right\}\\ {}^s\Gamma' = \left\{\overline{X'_k \mapsto {}^sN'_k}, \overline{X_l \mapsto {}^sN_l}\ ;\ \texttt{this} \mapsto \texttt{self}\ C\texttt{<}\overline{X'_k}\texttt{>}, \overline{pid \mapsto {}^sT_q}\right\}\\ {}^s\Gamma' \vdash \overline{{}^sN_l}, {}^sT, \overline{{}^sT_q}\ \ \mathsf{prg\ OK} \qquad \texttt{lost} \notin \overline{{}^sN_l}, \overline{{}^sT_q} \qquad {}^s\Gamma' \vdash e\ \ \mathsf{prg\ OK}\\ p = \texttt{pure} \implies \left(\text{free}\left(\overline{{}^sN_l}, \overline{{}^sT_q}\right) \subseteq \emptyset\ \wedge\ \texttt{any} \rhd \overline{{}^sN_l},\ \overline{{}^sT_q}\ =\ \overline{{}^sN_l},\ \overline{{}^sT_q}\right)\end{array}}{{}^s\Gamma, C \vdash p\ \texttt{<}\overline{X_l\ \texttt{extends}\ {}^sN_l}\texttt{>}\ {}^sT\ m\texttt{(}\overline{{}^sT_q\ pid}\texttt{)}\ \texttt{\{}\ e\ \texttt{\}}\ \ \mathsf{prg\ OK}}\ \text{PMD\_\_DEF}$$

Pure methods must not use type variables in parameter types and method upper bounds, and these types must not change by viewpoint adaptation with `any`. This ensures that pure methods are callable on any receiver type.

In expressions, note that the type in an object creation and the method type arguments are already checked to be strictly well formed by the topological judgment. The only remaining explicit type written by the programmer is the type of a cast expression, which is checked to be reasonable; we do allow `lost` in the type of a cast. The types of subexpressions are not checked to be reasonable, as the `this` expression might introduce `self` as a valid type or viewpoint adaptation might introduce `lost` in a type. Subsumption will be used to assign such types to `peer` or `any` references, respectively.

Definition 2.5.6 (Reasonable Expression)

$\boxed{{}^s\Gamma \vdash e\ \ \mathsf{prg\ OK}}$ reasonable expression

$$\frac{}{{}^s\Gamma \vdash \texttt{null} : \_}\ \text{PE\_\_NULL} \qquad \frac{}{{}^s\Gamma \vdash x : \_}\ \text{PE\_\_VAR}$$

$$\frac{}{{}^s\Gamma \vdash \texttt{new}\ {}^sT\texttt{()} : \_}\ \text{PE\_\_NEW} \qquad \frac{{}^s\Gamma \vdash \texttt{(}{}^sT\texttt{)}\ e : \_ \quad {}^s\Gamma \vdash e\ \ \mathsf{prg\ OK} \qquad {}^s\Gamma \vdash {}^sT\ \ \mathsf{prg\ OK}}{{}^s\Gamma \vdash \texttt{(}{}^sT\texttt{)}\ e\ \ \mathsf{prg\ OK}}\ \text{PE\_\_CAST}$$

$$\frac{{}^s\Gamma \vdash e_0.f : \_ \quad {}^s\Gamma \vdash e_0\ \ \mathsf{prg\ OK}}{{}^s\Gamma \vdash e_0.f\ \ \mathsf{prg\ OK}}\ \text{PE\_\_READ} \qquad \frac{{}^s\Gamma \vdash e_0.f\ \texttt{=}\ e_1 : \_ \quad {}^s\Gamma \vdash e_0\ \ \mathsf{prg\ OK} \qquad {}^s\Gamma \vdash e_1\ \ \mathsf{prg\ OK}}{{}^s\Gamma \vdash e_0.f\ \texttt{=}\ e_1\ \ \mathsf{prg\ OK}}\ \text{PE\_\_WRITE}$$

$$\frac{{}^s\Gamma \vdash e_0\ .\ m\texttt{<}\overline{{}^sT}\texttt{>(}\overline{e}\texttt{)} : \_ \quad {}^s\Gamma \vdash e_0\ \ \mathsf{prg\ OK} \qquad {}^s\Gamma \vdash \overline{e}\ \ \mathsf{prg\ OK}}{{}^s\Gamma \vdash e_0\ .\ m\texttt{<}\overline{{}^sT}\texttt{>(}\overline{e}\texttt{)}\ \ \mathsf{prg\ OK}}\ \text{PE\_\_CALL}$$

```
class Stack<X extends any Object> {
    void push(X p) {...}
    X pop() {...}
}

class C {
    void m() {
        peer Stack<rep Data> x = new peer Stack<rep Data>();
        x.push( new rep Data() );
        rep Data y = x.pop();
    }
}
```

Figure 2.12: A simple stack and client.

```
class Stack {
  void push(any Object p) {...}
  any Object pop() {...}
}

class C {
  void m() {
    peer Stack x = new peer Stack();
    x.push( new rep Data() );
    rep Data y = (rep Data) x.pop();
  }
}
```

Figure 2.13: Erasure of the stack from Fig. 2.12 to a non-generic Universe Types program.

Finally, a program is reasonable, if all classes and the main expression are reasonable.

Definition 2.5.7 (Reasonable Program)

$\boxed{\vdash P \ \ \mathsf{prg\ OK}}$ reasonable program

$$\frac{\begin{array}{l}\vdash \overline{Cls_i},\ C,\ e\ \ \mathsf{OK} \\ \overline{Cls_i\ \ \mathsf{prg\ OK}} \\ \{\emptyset\ ;\ \texttt{this} \mapsto \texttt{self}\ C\texttt{<}\emptyset\texttt{>}\} \vdash e\ \ \mathsf{prg\ OK}\end{array}}{\vdash \overline{Cls_i},\ C,\ e\ \ \mathsf{prg\ OK}}\ \text{PP\_DEF}$$

This concludes our discussion of reasonable programs. We expect these checks to support the programmer in writing meaningful programs without limiting the expressiveness of GUT.

2.5.2 Erasure and Expansion of Type Arguments

It is possible to erase a GUT program into a Universe Types program without generics [55], using casts. The interpretation of casts and type arguments is the same: both are from the viewpoint of the current object. Therefore, the casts inserted into the erased program use the same types that are used as type arguments. The simple example from Fig. 2.12 can be erased to the UT program in Fig. 2.13.

However, we cannot translate a GUT program into a UT program by "expansion", i.e., generating a new class for each type where the type arguments are substituted for the type variables. In our example, this expansion would produce the code in Fig. 2.14.

```
class StackrepData {
  void push(rep Data p) {...}
  rep Data pop() {...}
}

class C {
  void m() {
    peer StackrepData x = new peer StackrepData();
    x.push( new rep Data() ); // error
    rep Data y = x.pop();  // error
  }
}
```

Figure 2.14: The stack from Fig. 2.12 after "expansion" of the type argument.

This does not work, because the declared types will be adapted from the type of the receiver to the current viewpoint. In the above example, this viewpoint adaptation will result in lost ownership information, which will forbid the call of method `push` and return less precise information for the result of `pop`. For some combinations of receiver and parameter types this expansion is possible, but in general, as illustrated by the example above, such an expansion is not possible.

We could adapt the runtime model to be closer to Java: keeping the runtime type variables in the environment and in the heap is not done in Java. If we forbid creation of type variables and casts with type arguments, as is done in Java, they are not needed. We could still store the main owner in the heap, in order for downcasts from `any` to `rep` or `peer` to be checked at runtime.

2.5.3 Arrays

Early work on the UTS did not treat arrays. In languages like Java, arrays are also objects and therefore also need ownership modifiers. For the integration of the UTS into JML [63] we also discussed how arrays are handled and later formalized the approach [108, 115].

An array of reference type always has two ownership modifiers, the first for the array object itself and the second for the elements. Both modifiers express ownership relative to the receiver object and both modifiers can be any of the ownership modifiers. For example, the type `rep any Object[]` says that the array object itself is owned by the receiver object, but the elements are in an arbitrary ownership context. A `peer rep Object[]` type says that the array object has the same owner as the receiver object and that the array elements are owned by the receiver object.

All array objects in a multi-dimensional array of a reference type are in the same context, which is determined by the first ownership modifier. For example, if an instance field, `f`, has type `rep peer Object[][]`, then `f` and `f[3]` are both owned by the receiver and `f[3][1]` has the same owner as the receiver object.

For one-dimensional arrays of primitive types, the second ownership modifier is omitted. Primitive types are not owned and do not take an ownership modifier. A one-dimensional array of primitive types is one object that needs to specify ownership information. For example, the type `any int[]` expresses that the array object can belong to any context. A `rep int[]` references an array object that is owned by the receiver object and that manages `int` values.

Multi-dimensional arrays of primitive types have two ownership modifiers, the first for the array objects at higher levels and the second for the one-dimensional array at the "lowest" level.

All array objects in a multidimensional array, except the array objects at the lowest level, are in the same context, which is determined by the first ownership modifier.

For example, if an instance field, `g`, has type `rep peer int[][][]`, then:

- `g` references a `rep peer int[][][]` array object that is owned by the receiver and the array manages `rep peer int[][]` references.
- `g[3]` references a `rep peer int[][]` array object that is owned by the receiver and the array manages `peer int[]` references.
- `g[3][1]` references a `peer int[]` array object that has the same owner as the receiver and the array manages `int` values.
- `g[3][1][0]` is an `int` value.

Note how the first modifier changes when going from a two- or more-dimensional array of a primitive type to a one-dimensional array of a primitive type.

Also note that `java.lang.Object` is a supertype of arrays, in particular also of arrays of primitive type. A `peer int[]` can be assigned to a `peer Object` reference. Then a `rep peer Object[][]` type behaves consistently with the `rep peer int[][][]` type.

Following the convention in Java, array types support covariant subtyping that needs runtime checks on write accesses. For example, a `peer rep Object[]` is a subtype of a `peer any Object[]` and when an element is inserted it needs to be checked that it is owned by the receiver object. Note how in Generic Universe Types no runtime checks are needed for the limited covariance that we support for generic types.

Our handling of arrays differs from our earlier interpretation [63]. Previously, the second ownership modifier was relative to the referenced array object, not relative to the current object. During our work on Generic Universe Types we decided to uniformly treat all ownership modifiers relative to the current object. This change makes the interpretation of the ownership modifiers in the array type `peer rep Object[]` equivalent to the modifiers in `peer List<rep Object>` and also allows previously forbidden combinations, i.e., the use of `rep` as second ownership modifier.

In a formalization of the Universe type system that includes arrays [108], we extend the syntax to allow a separate ownership modifier for each array dimension and show the soundness of this system. The simple syntax with two ownership modifiers can be expanded to the syntax with an ownership modifier for each dimension. Our experience so far suggests that the use of two modifiers is sufficient in many cases; allowing both kinds of array annotations might be an option.

2.5.4 Exceptions

The handling of exceptions in ownership type systems poses several problems [62]. The exception object might reference application objects, i.e., to reference with which object an exception occurred. The object that creates an exception and the object that will eventually handle the exception might be in different contexts.

We discuss and compare the expressiveness, applicability, and implementation complexity of four possible ways to handle exceptions in ownership type systems [62]: 1) clone the exception object whenever it needs to cross a context boundary; 2) transfer the exception object to the context of the handling object; 3) only use global exceptions; or 4) use `any` references to the exception object.

Consider the code from Fig. 2.15. An object of class `Person` in Fig. 2.15 owns its `Car` object. `Car` objects refer to a global object representing the manufacturer's company.

Note that for ownership systems that enforce the owner-as-dominator discipline a reference into the representation of a different object is forbidden. Therefore, either the exception object is cloned or transferred to the handling context, creating overhead and limiting the use of exceptions, or the exception object is created in the root context, again limiting the use of references to application objects. All three cases cannot efficiently handle the example in Fig. 2.15, as a direct reference to the `Engine` is forbidden in owner-as-dominator systems.

In the Universe type system we use `any` references to propagate exceptions. This comes with no additional runtime overhead, because the objects do not need to be transfered or cloned to the handling context. It also allows the sharing of application objects, because `any` references can cross context boundaries. This allows the exception object to reference the `Engine` with a peer type. From the point of view of the exception handler in class `Person`, the access to the `Engine` is limited to an `any` reference.

If the owner-as-modifier discipline is enforced, then the modification of exception objects in exception handlers is forbidden, as they are in an unknown context. Such exception handlers should be rewritten to use exception chaining, that is, create a new exception object locally that contains a reference to the original exception.

Java 5 forbids the use of generics for exceptions. Therefore, we do not need to revisit the handling of generics for Generic Universe Types and can continue to use `any` as modifier for exceptions.

2.5.5 Static Fields

Our discussion so far focused on instance fields and methods and assumed that a current object `this` is always available. However, an extension to handle static fields and methods is possible [135, 115].

Static fields exist per class, not per instance. A private static field can be accessed from any instance of the class, independent of the location of the instance in the ownership tree. The only statically safe ownership modifier for static fields is therefore `any`, as it expresses exactly that the referenced object can be in an arbitrary context. The use of `peer` and `rep` is forbidden, as we do not treat the class object as a separate object in the ownership tree.

Consider this example:

```
class Global {
  public static any Data global;

  static {
    global = new peer Data();
  }
}
```

The static field `global` provides some global information. It can be accessed as usual as `Global.global`. Note that also updates of the field are allowed, e.g., as the field is public, any object could execute `Global.global = ...` to modify it. If possible the field should be private and proper information hiding should be applied.

Additional modifiers could be added to the Universe type system to allow a more flexible handling of global data [91]. However, the handling of static fields for modular verification is unclear. Recent work suggests possible refinements [182].

```
class Person {
  rep Car mycar;

  void drive() {
    try {
      ...
      mycar.start();
      ...
    } catch( any CarException ce ) {
      any Object e = ce.getOrigin();
      ...
    }
  }
}

class Engine {
  void start() throws any CarException {
    ...
    throw new peer CarException( this );
    ...
  }
}

class Car {
  rep  Engine engine;
  any Company manufacturer;

  void start() throws any CarException {
    ...
    engine.start();
    ...
  }
}

class CarException extends Exception {
  peer Object origin;
  ...
}
```

Figure 2.15: A `Person` object owns its `Car` object. Method `Person.drive` starts the person's car. A `CarException` is thrown if there is a problem. Such exceptions store a reference to the origin of the exception.

2.5.6 Static Methods

A static method declaration cannot use the `rep` modifier in the method signature or implementation, as there is no current object available during execution. The `any` modifier is interpreted as usual as referring to an object in an arbitrary context.

The interpretation of `peer` modifiers is slightly adapted: instead of meaning that the referenced object has the same owner as the current object, it is interpreted as being in the same context as the calling context, where the calling context depends on the call of the static method. Assume a static method `m` in class `C`. If the method is called as `peer C.m()`, then the calling context of the static method is the current calling context, if the calling method is static, or the context that contains the `this` object, if the calling method is an instance method. If the method is called as `rep C.m()`, then the calling context is the context owned by the current object `this`; note that this is only possible in an instance method. A call of the form `any C.m()` is forbidden, as it would not set a calling context.

The use of static methods allows us to express examples in which the context of the returned object should depend on the calling context. Consider the following code:

```
class Element {
  any String val;
  private Element(any String arg) { val = arg; }

  static peer Element createA() {
    return new peer Element("A");
  }
  static peer Element createB() {
    return new peer Element("B");
  }
}

class User {
  void m() {
    peer Element pe = peer Element.createA();
    rep Element re = rep Element.createB();
  }
}
```

This use is very similar to having additional constructors, but if one requires multiple constructors that take the same parameter types, using different static construction methods is an alternative.

The next example presents a class `Collections` that provides sorting features similar to `java.util.Collections`:

```
class Collections {
  static void sortInPlace(peer List inout) {
    // somehow sort elements of inout
  }

  static peer List sort(any List in) {
    peer List out = new peer List(in.size());
    // set all elements of out to the elements of in
    peer Collections.sort( out );
    return out;
  }
}
```

```
class ListUser {
  void m() {
    rep List unsrt = ...;
    rep Collections.sortInPlace(unsrt);
    any List al = ...;
    peer List sorted = peer Collections.sort(al);
  }
}
```

Method `sortInPlace` requires that the argument is in the calling context and can then modify the referenced collection directly. Alternatively, method `sort` can sort a list in an arbitrary context. It first creates a local duplicate of the input list and then sorts this list; of course, some other, more direct, sorting from `in` into `out` might be implemented.

Our interpretation of static methods gives us the possibility to delegate the creation of objects to a different class. For more general patterns, e.g., the factory method pattern, ownership transfer is necessary [143].

2.5.7 Map Example

In the following we discuss the main differences between GUT_1 [60] and our current work on GUT_2 by an example. The example is basically the same as the one used in GUT_1 with minor modifications and also similar to the example in Sec. 2.1.

GUT_1 used the $\triangleright_m$ function to recursively check whether the main modifier of a type needs to be changed to `any` to prevent the possible unsoundness of a covariant type argument adaptation. This change relied on the fact that the owner-as-modifier discipline is always enforced to ensure type soundness. In GUT_2 we use the `lost` modifier to express that ownership information was lost locally and do not need to change other ownership modifiers. We can now cleanly separate the topological system from the owner-as-modifier discipline.

Class `Map` (Fig. 2.16) implements a generic map from keys to values. Key-value pairs are stored in singly-linked `Node` objects, where class `Node` extends the superclass `MapNode` (both Fig. 2.17). In contrast to Fig. 2.2, we parameterize class `MapNode` by the key, the value, and the location of the next node and class `Node` instantiates `MapNode` with `peer Node<K, V>` as corresponding argument. This ensures that the maps are owned by the same object and allows an elegant iterator implementation, which we discuss later.

Class `Client` (Fig. 2.18) stores two references to map objects: `mapr`, to a map in the same context that manages keys that are of type `rep ID` and values of type `any Data`; the other field, `mapa`, to a map in an arbitrary context that manages keys that are of type `any ID` and values of type `any Data`.

Now consider the example code using GUT_1 that is shown in Fig. 2.19. In `Appl1`, the ownership of the first type argument of `c.mapr` cannot be expressed from the current viewpoint, because it is neither owned by the current object nor shares the same owner with the current object. In GUT_1 the viewpoint adaptation of `peer Map<rep ID, any Data>` from the viewpoint `peer Client` needs to be `any Map<any ID, any Data>`. The only available ownership modifier for `ID` is `any` and because of the covariant change in a type argument, the main modifier of the enclosing type also has to be changed to `any`. The enforcement of the owner-as-modifier discipline ensures that the reference can only be used to read information and the covariant change in the type argument does not impact type soundness. We are still able to call pure functions on `mmapr`, but cannot call the non-pure `deleteX` methods.

```
class Map<K, V> {
  rep Node<K, V> first;

  void put(K key, V value) {
    rep Node<K, V> newfirst = new rep Node<K, V>();
    newfirst.init(key, value, first);
    first = newfirst;
  }

  pure V get(any Object key)
    { /* omitted */ return null; }

  void deleteK(K key) { /* omitted */ }
  void deleteV(V val) { /* omitted */ }

  pure peer Iter<K, V> getIter() {
    peer IterImpl<K, V, rep Node<K, V>> it =
      new peer IterImpl<K, V, rep Node<K, V>>();
    it.setCurrent(first);
    return it;
  }
}
```

Figure 2.16: `Map` implementation.

```
class MapNode<K, V,
          X extends any MapNode<K, V, X> > {
  K key; V value; X next;

  void init(K k, V v, X n)
    { key = k; value = v; next = n; }
}

class Node<K, V> extends MapNode<K, V,
                            peer Node<K, V> > {}
```

Figure 2.17: `MapNode` and `Node` implementations.

```
class Client {
    peer Map<rep ID, any Data> mapr = new peer Map<rep ID, any Data>();
    any Map<any ID, any Data> mapa = new peer Map<any ID, any Data>();
}
```

Figure 2.18: The `Client` class stores two references to maps.

```
class Appl1 {
  void m(peer Client c) {
    any Map<any ID, any Data> mmapr = c.mapr; // OK
    c.mapr = new peer MapIter<peer ID, any Data>(); // Error: update forbidden
    mmapr.get(new peer ID());         // OK: get is pure
    mmapr.deleteK(new peer ID());     // Error: deleteK not pure
    mmapr.deleteV(new peer Data());   // Error: deleteV not pure
  }
}
```

Figure 2.19: GUT_1 example.

```
class Appl2 {
  void m(peer Client c) {
    peer Map<lost ID, any Data> mmapr = c.mapr; // OK
    c.mapr = new peer MapIter<peer ID, any Data>(); // Error: c.mapr contains lost
    mmapr.get(new peer ID());         // topol. and encaps. OK
    mmapr.deleteK(new peer ID());     // Error: lost in parameter type
    mmapr.deleteV(new peer Data());   // topol. and encaps. OK

    any Map<any ID, any Data> mmapa = c.mapa; // OK
    c.mapa = new peer MapIter<any ID, any Data>(); // OK
    mmapa.get(new peer ID());         // topol. and encaps. OK
    mmapa.deleteK(new peer ID());     // topol. OK, not encapsulated
    mmapa.deleteV(new peer Data());   // topol. OK, not encapsulated
  }
}
```

Figure 2.20: GUT_2 example.

However, in order to preserve soundness, this solution lost ownership information that is statically available. We statically know that the `Client` object referenced by `c` has the same owner as the current `Appl1` object and furthermore know that the `Map` object referenced by `mapr` and the current `Client` object have the same owner. Therefore, it is statically known that the current `Appl1` object and the `Map` object referenced by `mmapr` have the same owner. We have to change the main modifier of `mmapr` to `any` in order to prevent modifications of the map object (the owner-as-modifier discipline forbids modifications through `any` references).

Also, in GUT_1, separating the topology from the encapsulation discipline is not possible. From the viewpoint `Appl1` the fields `c.mapr` and `c.mapa` have the same type `any Map<any ID, any Data>`. Updates on `c.mapa` would be type safe, as the referenced object actually was created with `any` as type argument. The GUT_1 type system does not distinguish these two different kinds of `Map` objects.

Our goal in designing GUT_2 was to separate the topology and the encapsulation aspects cleanly, to improve the expressiveness, and to reduce the loss of ownership information. In GUT_2 the adaptation of `peer Client` and `peer Map<rep ID, any Data>` results in `peer Map<lost ID, any Data>`. We preserve that the `Map` is `peer` and express the loss of ownership information by the new modifier `lost`. The example code using GUT_2 is shown in Fig. 2.20.

The ownership modifier `lost` is used to express that ownership information regarding this type might have been lost. In GUT_1 we had to change all enclosing ownership modifiers, if a covariant change happens. A change to `lost` is local and does not influence the enclosing modifiers.

How can we now ensure that the covariant change in the type argument does not create a soundness problem? Instead of relying on the owner-as-modifier discipline we strengthened the conditions for field updates and method calls. The adaptation of the type of the target of the field update and of the declared field type must not contain the modifier `lost`. This ensures that we can only update a field for which all ownership information is expressible from the local viewpoint. (Note that no subtyping is applied to the type of the left-hand side, that is, even though `lost` is a subtype of `any`, we only need to forbid updates if the viewpoint-adapted type contains `lost`.) This rule again forbids the unsafe update `c.mapr = ...` (on line 4 in Fig. 2.20).

Similarly, for a method call we have to ensure that the adaptation of the type of the receiver with the declared parameter types does not contain `lost`. However, the combined return type may contain `lost`, as this is only used to read information. This rule again forbids the call to `deleteK`, as the ownership information for `K` was lost (line 6 in Fig. 2.20). On the other hand, we can now allow the call to method `deleteV` (line 7 in Fig. 2.20). Even though some modifiers in the type of `mmapr` are `lost`, the adaptation with the parameter type, `V`, does not contain `lost`, so it is safe to call `deleteV`.

In GUT_2 we can now also modify `c.mapa` (line 10 in Fig. 2.20). The type does not contain lost ownership information and it is therefore type safe to update the field and to call methods on it. The distinction between the two fields `mapr` and `mapa` is now preserved by viewpoint adaptation. Note that the encapsulation system allows the call of non-pure methods on `mapr`, but forbids them on `mapa`, because the receiver has an unknown owner.

GUT_2 further improves the combination of generic types and ownership type systems. The clean separation of topology and encapsulation system eases the formalization and presentation of the system, and more static ownership information is preserved.

Iterators. Fig. 2.21 presents an iterator for the map. Note how type variable `X` is used to parameterize the iterator with the concrete location of the nodes. Method `next` is valid, because in class `MapNode` field `next` has type variable `X` as type. If we were to use the node implementation from Fig. 2.2, this parameterization would not work and field `current` would need to use the type `any Node<K, V>`, giving us no guarantee about the location of the nodes. Method `getIter` in class `Map` uses `rep Node<K, V>` as corresponding argument and thereby ensures that the iterator can only iterate over its representation objects.

Consider the use of the iterator in Fig. 2.22. This class is correct with regard to the topological system and maintains the encapsulation. Note that the return type of the call `itr.getKey()` is `lost ID`, because the concrete ownership of the keys was lost by viewpoint adaptation. Finally, consider the similar example in Fig. 2.23. The topology of this example is also well formed, but the encapsulation system is violated. Method `getIter` in class `Map` is pure and we can therefore receive a reference for a map that is in an unknown context. The calls to `isValid`, `getKey`, and `getValue` are correctly encapsulated, because these methods are pure. However, the call `ita.next()` violates the encapsulation system, because it is a non-pure method call on a receiver in an unknown context, and is forbidden; such a call would modify an object in an unknown context and thereby possibly break unknown invariants. One could write an alternative iterator that can iterate over a map in an unknown context. When this alternative iterator is in a `peer` or `rep` context, the iterator itself can be modified by calling `next`. However, note that such an iterator could not modify the map over which it iterates, e.g., a `remove` method could not be implemented.

This concludes our discussion of Generic Universe Types and the previous formalization [60]. The next section discusses related work and compares GUT to other ownership type systems.

```
interface Iter<K, V> {
  pure K getKey();
  pure V getValue();
  pure boolean isValid();
  void next();
}
class IterImpl<K, V, X extends any MapNode<K, V, X>>
implements Iter<K, V> {
  X current;

  void setCurrent(X c) { current = c; }
  pure K getKey() { return current.key; }
  pure V getValue() { return current.value; }
  pure boolean isValid() { return current != null; }
  void next() { current = current.next; }
}
```

Figure 2.21: Iterator interface and implementation.

```
class Appl3 {
  void foo(peer Client c) {
    peer Map<lost ID, any Data> mmapr = c.mapr;
    peer Iter<lost ID, any Data> itr = mmapr.getIter();
    while(itr.isValid()) {
      // use itr.getKey() and itr.getValue()
      itr.next();
    }
  }
}
```

Figure 2.22: Application using the iterator; the topology and encapsulation discipline are correct.

```
class Appl4 {
  void bar(peer Client c) {
    any Map<any ID, any Data> mmapa = c.mapa;
    any Iter<any ID, any Data> ita = mmapa.getIter();
    while(ita.isValid()) {
      // use itr.getKey() and itr.getValue()
      ita.next(); // (1)
    }
  }
}
```

Figure 2.23: Application using the iterator; the topology is correct, but the call on line (1) violates the encapsulation system and is forbidden.

2.6 Related Work

Early work on object-oriented programming already discussed the problems of object aliasing, for example, see the descriptions of Meyer [128, 129]. Guides to secure programming, for example for Java [183], also recognize the problems of aliasing.

The "Geneva convention on the treatment of object aliasing" [98] illustrates the problems and outlines four possible treatments: detection, advertisement, prevention, and control.

The Islands system [97] was the first approach to combat aliasing; however, it has a high annotation overhead. Balloon types [10, 11] use a type system and static analysis to give strong encapsulation guarantees. Both severely restrict the expressiveness and forbid many useful programs.

We structure the rest of this section as follows. In Sec. 2.6.1 we discuss the relation to other ownership type systems, in Sec. 2.6.2 we give an overview of the work on the Universe type system, and in Sec. 2.6.3 we discuss type systems that support read-only references and immutable objects. Finally, in Sec. 2.6.4 we briefly discuss object-oriented verification.

2.6.1 Ownership Type Systems

2.6.1.1 Ownership Types

Flexible alias protection [150] presents a mode system and discusses its use to protect against the negative effects of aliasing. Clarke et al. [48, 45] developed an ownership type system for flexible alias protection. It enforces the owner-as-dominator property and uses context parameters to express role separation and to allow an object, o, to reference objects in ancestor contexts of the context that contains o. Such references violate neither the owner-as-dominator nor the owner-as-modifier property. Still, we require references to ancestor contexts to be `any` to prevent methods from modifying objects in ancestor contexts because such modifications are difficult to handle by specification techniques for frame properties [141].

Context parameters allow a fine-grained specification of the ownership relationship. In contrast, the combination of type parameters and the `any` modifier allow GUT to choose between parameterizing the class and using an abstraction of the ownership. The abstract `any` types can replace context parameters in many situations, impose less annotation overhead, and lead to programs that are easier to read and reason about. Using type parameters allows one to parameterize a class by both, ownership and class information. For many examples we believe that the type parameters found in GUT will be expressive enough to model the desired ownership structures. Furthermore, type parameters and `any` references allow multiple objects to reference one representation, which is not supported by the owner-as-dominator model used in ownership types. However, such non-owning references to a representation are used in common implementations such as iterators or shared data structures.

In ownership types, the static visibility function SV is used to protect `rep` references from exposure. However, it forbids all access to representation from non-owners. In contrast, the viewpoint adaptation function used in GUT (see Sec. 2.3.1) will introduce `lost` ownership information if the exact ownership relation cannot be expressed. This still allows limited access to the representation of other objects. The substitution of context parameters also roughly corresponds to viewpoint adaptation in Universe types, which adapts ownership information and replaces type arguments for type parameters. Clarke's PhD thesis [45] gives a detailed development of an object calculus with ownership types and proves a containment invariant.

For a detailed comparison of parametric ownership type systems to non-generic Universe types see Sec. 2.6.2.3 and [34]. Extending this work to GUT is future work.

Clarke and Drossopoulou [46] extended the original ownership type system to support inheritance. Their type system is ownership parametric and enforces the owner-as-dominator property. Therefore, it suffers from the same problems as the original ownership type system. Based on their type system, Clarke and Drossopoulou present an effects system and use it to reason about aliasing and non-interference.

Multiple Ownership for Java-like Objects (MOJO) [37] is an ownership type system that enforces a more flexible topology, supporting more than one owner per object and path types. The type system does not enforce an encapsulation topology. The wildcard owner "?" provides ownership abstraction similar to `any` references in Universe types. This system is only parametric in ownership contexts, not in types.

2.6.1.2 SafeJava

SafeJava [23, 26] is very similar to ownership types, but supports a model that is slightly less restrictive than owner-as-dominator: An object and all associated instances of inner classes can access a common representation. For instance, iterators can be implemented as inner class of the collection and, therefore, directly reference the collection's representation. However, more general forms of sharing are not supported. SafeJava is more flexible than ownership types, but the Universe type system is both syntactically simpler and more expressive. SafeJava has been applied to prevent data races and deadlocks [24, 27].

Boyapati et al. [25] present a space-efficient implementation of downcasts in SafeJava. Their implementation inspects each class, C, to determine whether downcasts for C objects potentially require dynamic ownership information. If not, ownership information is not stored for C objects. "Anonymous owners", marked as "-" in class declarations, are used to mark owner parameters that are not used in the class body and do not need runtime representation. This optimization does not work for the Universe type system, where `any` references to objects of any class can be cast to `peer` or `rep` references and, therefore, objects of every class potentially need runtime ownership information.

SafeJava [23] supports type parameters and ownership parameters independently, but does not integrate both forms of parametricity. This leads to significant annotation overhead. Also, the combination with type parameterization is not formalized. No implementation is available.

2.6.1.3 Ownership Domains

Ownership domains [7] support a model that is less restrictive than owner-as-dominator. Contexts can be structured into several domains. Domains can be declared public, which permits reference chains to objects in the public domain that do not pass through their owner. Programmers can control whether objects in different domains can reference each other. For instance, iterators in a public domain of a collection are accessible for clients of the collection. They can be allowed to reference the representation of the collection stored in another domain. However, the use of public domains and linking of domains can lead to complex systems [145].

Ownership domains have been used to visualize software architectures [1, 3]. They have also been encoded in Java 5 annotations [2], however, the limited capabilities of Java 5 annotations required a complex encoding.

The concept of ownership domains is powerful and allows many forms of sharing. However, its suitability to support verification of functional correctness properties is unclear. Supporting verification has been the main motivation behind the Universe type system. Another drawback of ownership domains is the annotation overhead they impose. Like ownership types, ownership

domains impose the annotation overhead of context parameters; in addition link declarations are necessary to define the desired sharing of the system.

Ownership Domains [7] combine type parameters and domain parameters into a single parameter space and thereby reduce the annotation overhead. However, type parameters are not covered by their formalization. Ownership Domains are integrated in the ArchJava compiler [6].

2.6.1.4 Systems by Lu and Potter

The Acyclic Region Type System (ARTS) [120] separates the heap into regions and ensures that reference cycles can only occur within a region. The core language does not use ownership and provides a strong encapsulation discipline that prohibits common patterns.

Effective ownership types [122] provide effect encapsulation and enforce the owner-as-modifier discipline. It uses `any` and `unknown` modifiers, corresponding to the `any` and `lost` modifiers in our work. However, they always enforce the owner-as-modifier discipline and do not separate the system into a topology and an encapsulation system.

Variant ownership types [121] support both ownership and accessibility modifiers, allowing a fine-grained access scheme. Context variance allows to abstract over ownership information.

None of the three systems is type parametric and no implementations are available.

2.6.1.5 Ownership Generic Java

Ownership Generic Java (OGJ) [162, 161] allows programmers to attach ownership information through type parameters. OGJ enforces the owner-as-dominator discipline. It piggybacks ownership information on type parameters. In particular, each class C has a type parameter to encode the owner of a C object. This encoding allows OGJ to use a slight adaptation of the normal Java type rules to also check ownership, which makes the formalization very elegant. Similarly to OGJ the Generic Confinement system [163, 161] encodes package-level ownership on top of a generic type system. WOGJ [38] presents an encoding of ownership on top of a generic type system that supports wildcards and requires less changes to the underlying type system than OGJ.

We believe that adapting OGJ to separate the topological system from the encapsulation system or to support the owner-as-modifier discipline is not easily accomplished. One needs to loosen up the static ownership information by allowing certain references to point to objects in any context. In OGJ, the subtype relation between any types and other types would require covariant subtyping, for instance, that `Node<This>` is a subtype of `Node<Any>`. In OGJ there is no covariant subtyping, because the underlying Java (or C♯) type system is non-variant. Therefore, piggybacking ownership on the standard Java type system is not possible in the presence of `any` ownership.

The formalization of OGJ [161] leaves certain points about the handling of upper bounds unclear. The prototype compiler for OGJ in our experiments accepted many invalid programs.

2.6.1.6 Existential Types

Higher-order functional ownership by Krishnaswami and Aldrich [111] allows the abstraction of ownership information similar to `any` references.

Existential owners for ownership types [191] provides a mechanism that allows ownership types to support some downcasts without requiring a runtime representation of ownership. This system does not provide the full flexibility of `any` references.

Jo∃ [35, 33] combines the theory on existential types with a parametric ownership type system. Ownership information is passed as additional type parameters, and existential types can be used to allow subtype variance. Jo$\exists_{deep}$ provides optional enforcement of the owner-as-dominator discipline. The Jo∃ system provides theoretical underpinnings and builds a theoretically sound basis. It does not provide a practical language design and no language implementation.

We discussed the relationship between a subset of Jo∃, called Jo$\exists^{-}$, and the non-generic Universe type system [34], see our discussion in Sec. 2.6.2.3 below. Analyzing the correspondence between GUT and Jo∃ will provide interesting future work.

A formalization of wildcards [36] uses existential quantification to model Java wildcards. It could also provide insights into how to model `lost` and `any` in future systems.

2.6.1.7 Other Ownership Type Systems

Confined types [22] guarantee that objects of a confined type cannot be referenced in code declared outside the confining package. Confined types have been designed for the development of secure systems. They do not support representation encapsulation on the object level.

A confinement system has also been used to ensure correct behavior of Enterprise Java Beans [49].

Banerjee and Naumann use ownership to prove a representation independence result for object-oriented programs [13, 14]. Their ownership model requires that for a given pair of classes C, D, all instances of D are owned by some instance of C. This is clearly too restrictive for many implementations. For instance, lists are typically used as internal representation by many classes. Similarly, it is unclear how arrays can be supported by such a model. Banerjee and Naumann present a static analysis to check whether a program satisfies the ownership model for a pair of classes C, D.

The work on Simple Loose Ownership Domains and Boxes [170, 171] provide a model of encapsulation that is based on Ownership Domains [7] but allows "loose" references to representation domains, abstracting multiple domains with a single type. It was also adapted to active objects in CoBoxes [172]; a compiler for JCoBox, the realization of CoBoxes for Java, is available from `http://softech.informatik.uni-kl.de/Homepage/JCoBox`.

Pedigree types [119] provide additional ownership modifiers that allow a finer description of ownership relations, similar to path types supported in other systems. They also present an interesting inference system.

Ownership has received considerable attention for real-time and concurrent applications, for example, the work on SafeJava [24, 27, 23] mentioned above, scoped types for real-time applications [12], the use for components and process calculi [96], the use of an ownership topology for concurrency [56], the use of a dynamic ownership model for concurrency in Spec♯ [103], and an ownership system for object race detection [186].

The work on gradual encapsulation and decapsulation [94] in the context of ObjectTeams [95] provides interesting discussions.

Also the much broader aims of shape, alias, and static analysis in general can be considered related work and investigating relationships between ownership systems and these more general approaches will provide interesting future work.

2.6.2 Universe Type System

The original design goals of the Universe type system, according to [137], are:

1. have simple semantics,

2. be easy to apply,
3. be statically checkable,
4. guarantee an invariant that is strong enough for modular reasoning, and
5. be flexible enough for many useful programming patterns.

These goals were also our guiding principles for the development of Generic Universe Types.

In the following we first discuss different formalizations of the Universe type system, then discuss the relation to dependent type systems, and finally compare Universe types to parametric ownership systems that support existential quantification.

2.6.2.1 Formalizations

The Universe type system (UTS) was first introduced by Müller and Poetzsch-Heffter [137, 138]. The early syntax was different to the current syntax and Type Universes were later removed. These early formalizations already use a "type combinator" to adapt a declared type to a changed viewpoint. The syntax using the three ownership modifiers `peer`, `rep`, and `readonly` was first used by Müller and Poetzsch-Heffter [140]. The UTS was used by Müller to develop a modular verification methodology for object-oriented programs [134, 135].

In [63] we present the integration of the Universe type system into the Java Modeling Language (JML). We implemented a type checker, runtime checks, and bytecode information for GUT in the JML tools ; see Chapter 3 for a detailed discussion of the tools that are available for GUT.

Universe Types with Transfer [143] realize ownership transfer for non-generic Universe types.

All formalizations mentioned above always enforce the owner-as-modifier discipline. For some applications of ownership, e.g., for concurrency [56], only enforcing the heap topology is enough.

In a master's project [108] we first developed the separation of the UTS into a *topological* system and an *encapsulation* discipline. We renamed the modifier `readonly` to `any`, because this ownership modifier now signifies only that the object is in an arbitrary ownership context and not necessarily that it is used only for reading. The `any` modifier is a "don't care" modifier, it expresses that for the annotated reference the ownership of the referenced object is of no concern. We also introduce a new ownership modifier `unknown` that signifies that static ownership information for a reference is not available. In contrast to `any`, this is a "don't know" modifier: from the current viewpoint we cannot express the ownership relation, but the declared type might have a constraint.

The master's thesis [108] also provides a type soundness proof in the automatic theorem prover Isabelle [148]. It defines a Java subset, based on Featherweight Java [99, 81] and Jinja [109, 110], extends it with Universe modifiers, and shows type soundness. To the best of our knowledge, this is the first soundness proof of an ownership type system in an automatic theorem prover.

In recent work, non-generic Universe types were separated into a topological system and an encapsulation system [55]. We use `lost` as main modifier to signify that ownership information cannot be expressed, similarly to the `unknown` modifier in [108]. Using `lost` in the non-generic system allowed the clean separation of the topology and the encapsulation system and simplified the rules and their presentation. This thesis provides an independent formalization of Generic Universe Types with the separation into a topological system and an encapsulation system. The encapsulation system in [55] made the distinction between an encapsulation property

and an owner-as-modifier property. In this terminology, our owner-as-modifier Theorem 2.4.7 corresponds to the encapsulation property; a corresponding owner-as-modifier property could be proved using a small-step semantics or using execution traces.

A previous version of Generic Universe Types [60, 59] always enforces the owner-as-modifier discipline. A particularly interesting aspect of that version is how generics and ownership can be combined in the presence of an `any` modifier, in particular, how a restricted form of ownership covariance can be permitted without runtime checks. For this ownership covariance to be safe, the enforcement of the owner-as-modifier discipline and the rules for subtyping and viewpoint adaptation are tightly coupled. In this current work, we allow ownership covariance using the `lost` modifier and thereby cleanly separate viewpoint adaptation and subtyping from the enforcement of an encapsulation system.

The separation of topology and encapsulation is simplified by distinguishing between a reference that can refer to an arbitrary object and a reference that points to a specific context, but where this specific context is not expressible in the type system. We distinguish between the "don't care" modifier `any` that can reference an arbitrary object and the "don't know" modifier `lost` that references an object for which the precise ownership information cannot be expressed statically. Updates of `any` variables are always possible, since the owner of their value is not of interest. Updates of `lost` variables must be forbidden, since the ownership information required for type-safe updates is not statically known.

2.6.2.2 Dependent Types

Ownership type systems structure the heap and enforce restrictions on the behavior of a program. Virtual classes [126, 71, 72] express dependencies between objects. Similar to virtual methods, a class A can declare a dependent class B by nesting class B within the definition of A. Virtual classes can be overridden in subclasses, and the runtime type of an object determines the concrete definition of a virtual class. Recent work [154, 73, 47, 100, 167] formalized and extended virtual classes. All of these systems have in common that dependency is expressed by nesting of classes.

Dependent classes [84, 85] are a generalization of virtual class systems that allows one class to depend on multiple objects. Dependency is expressed by explicit declaration of the depended-upon objects as class parameters. This allows one to declare dependencies independently of the nesting of classes, which increases the expressive power and reduces the coupling between classes. Even more generally, constrained types [152, 41] can express multiple constraints and is parametric in the constraint system.

Ownership type systems, in particular the Universe type system and Ownership Domains [7], can be expressed using dependent classes [66]. The ownership structure is made explicit by adding a dependency on an immutable `owner` field, similarly to the dynamic encoding presented in [63]. The ownership modifiers of the UTS can be directly expressed as constraints on this `owner` field. Even more fine-grained relationships can be expressed, for example, that an object is in an unknown context, but in the same context as some other object.

In the dependent classes [84] syntax, we can declare the following class:

```
class OwnedObject( owner: Object ) {}
```

We can then express the topology of the following simple program that uses Universe types:

```
class C {...}
class D {
  rep C f = new rep C();
}
```

The field `f` may reference only `C` objects that have the current `D` object as owner. We can express this in the dependent classes encoding as:

```
class C( Object owner ) extends OwnedObject {...}
class D( Object owner ) extends OwnedObject {
  C(owner: this) f = new C(owner=this);
}
```

In the above program, we make explicit that the owner of the referenced `C` object is the current `this` object. Similarly, a `peer` reference is translated into the constraint that the referenced object has the same owner as the current object.

2.6.2.3 Parametric Ownership with Existential Types

Parametric ownership types [150, 48, 45, 46, 162] discussed above and the non-generic Universe type system [63] are two ownership type systems that describe an ownership hierarchy and statically check that this hierarchy is maintained. They both provide (different) encapsulation disciplines.

Ownership types can describe fine-grained heap topologies, whereas Universe types are more flexible and easier to use. No direct encoding of one type system in the other has been possible: the abstraction provided by `any` references in the Universe type system could not be modeled with parametric ownership types.

Recently, parametric ownership has been extended with existential quantification of contexts [35, 33]. This extension, called Jo∃, provides the possibility to abstract from concrete ownership information—similarly to `any` references in Universe types.

We show in [34] that the descriptive parts of the Universe type system [55] and a variant of Jo∃, which we call Jo$\exists^{-}$, are equivalent. In Jo$\exists^{-}$ we use a single owner parameter that corresponds to the single ownership modifier of the UTS. Note that full Jo∃ allows multiple owner parameters and is thus more expressive than non-generic Universe types.

We formalize this correspondence as encodings between the two systems. We have proved that the encodings from Universe types to Jo$\exists^{-}$ and from Jo$\exists^{-}$ to Universe types are sound; thus, we have shown that the two systems are equivalent with respect to type checking. As an intermediate step in the encoding we give an alternative formalization of the UTS which is closer to the underlying existential types model.

Consider the program P_1 using Universe types:

```
class C {
  peer Object f1;
  any Object f2;

  void m(any C x) {
    this.f1 = new peer Object(); // 1: OK
    x.f1 = new peer Object();   // 2: error
    x.f2 = new peer Object();   // 3: OK
  }
}
```

and the program P_2 using Jo$\exists^{-}$ types:

```
class C<owner> {
  Object<owner> f1;
  ∃o. Object<o> f2;
```

```
  void m(∃o. C<o> x) {
    this.f1 = new Object<owner>(); // 1: OK
    x.f1 = new Object<owner>();   // 2: error
    x.f2 = new Object<owner>();   // 3: OK
  }
}
```

These two programs are equivalent, that is, both describe the same topology and type checking in both systems rejects expression 2.

In P_1 the field update `x.f1` in expression 2 is forbidden, as the viewpoint adaptation `peer Object` from `any` results in `lost Object` and `lost` is forbidden in the adapted field type. On the other hand, the field update `x.f2` in expression 3 is allowed, as `any Object` from `any` results in `any Object` and the right-hand side is a correct subtype.

In expression 2 of P_2, the type of `x` must be unpacked before it can be used. Therefore, the field type lookup for field `f1` in type `C<o1>` is performed, where `o1` is a fresh context variable. This lookup gives the type `Object<o1>`. There is no subtype relationship between `Object<owner>` and `Object<o1>` because their parameters do not match and subtyping of un-quantified types is invariant.

In expression 3, the lookup for field `f2` in type `C<o1>` results in `∃o. Object<o>`, which is a supertype of `Object<owner>`, because of the variance of existential types, and the assignment is allowed.

The investigation of encoding ownership type systems in dependent types and the relationship between Universe types and a parametric ownership system with existential quantification gave us valuable insights into these type systems and gave us promising ideas for future research for combining these systems with GUT.

2.6.3 Read-only References and Immutability

Skoglund [177, 178] as well as Birka and Ernst [21] present type systems for readonly types that are similar to readonly references when the owner-as-modifier discipline is enforced. Birka and Ernst's type system is more flexible than ours as it allows one to exclude certain fields or objects from the immutable state. Neither Skoglund nor Birka and Ernst combined readonly types with ownership. The combination with ownership gives more context to decide when a downcast to a read-write reference is valid.

Our readonly types leave the owner of an object unspecified. Whenever precise information about the owner is needed, a downcast with a dynamic type check is used. This approach is similar to soft typing [39, 40], where a compiler does not reject programs that contain potential type errors, but rather introduces runtime checks around suspect statements. In soft typing, these runtime checks are added automatically by the compiler whereas we require programmers to introduce casts manually.

Immutability Generic Java (IGJ) [193] allows the covariant change of type arguments of readonly and immutable types. The unsoundness of the covariant change is prevented by forbidding modifications through readonly and immutable types. However, the erased-signature rule is needed to ensure that overriding methods cannot introduce an unsoundness; this rule also requires that type variables that appear in the parameter types of pure methods are marked as non-variant. In contrast, in GUT the loss of ownership information in a covariant argument change is detected for the method call and we can therefore safely allow more methods.

Unique references and linear types [28, 29, 78, 187] can be used for a very restrictive form of alias control. For ownership type systems, a weaker form of uniqueness [50, 190] is sufficient to

enable ownership transfer. Universe Types with Transfer [143] realize ownership transfer for non-generic Universe types.

Ensuring that an object is immutable cannot be checked by the GUT type system. Immutability is present in other type systems (e.g., Javari [185], Jimuva [90], IGJ [193], and Joe_3 [155]) and in a dynamic encoding of ownership called frozen objects [117]. Investigating the combination with Universe types is interesting future work.

2.6.4 Object-Oriented Verification

Work on the formal verification of object-oriented programs is of particular interest to this thesis. Verification tools for the Java Modeling Language (JML) [30, 32, 43, 44, 53, 79, 102, 104] and Spec♯ [17] all have to deal with the problems of aliasing for program verification. Preventing representation exposure [57] and cross-type aliasing [58] also provide a discussion of aliasing problems and possible solutions.

A recent technical report provides an overview and comparison of different behavioral interface specification languages [93]. Specification and verification challenges were also recently summarized [114]. A framework for verification techniques for object invariants [69] describes different techniques using seven parameters.

Müller's thesis [135] provides the basis for the owner-as-modifier discipline that we enforce in GUT. Ownership was also used to reason about frame properties in JML [141]. Spec♯'s [17] dynamic ownership model [116] (also called the Boogie methodology) for reasoning about invariants is based on a dynamic ownership encoding similar to the one described in [63]. In this methodology, any reference can be used to modify an object, provided that all transitive owners of this object are made mutable by applying a special unpack operation. In practice, this requirement is typically met by following the owner-as-modifier policy: the owner unpacks itself before initiating the modification of an owned object. The Boogie methodology supports ownership transfer. It is future work to investigate how the topological system of GUT can be used together with the Boogie methodology. Reasoning using ownership [136] is a very promising approach, but some remaining obstacles need to be overcome.

Lu, Potter, and Xie [123] divide their system into validity invariants and the validity effect to describe which objects need to be revalidated. The system is based on effective and variant ownership types [122, 121].

Poetsch-Heffter and Schäfer [159, 160] describe the modular specification of components based on their boxes type system [170, 171, 172].

Alternative approaches to the formal verification of systems are, for example, separation logic [166], regional logic [15], and (implicit) dynamic frames [105, 179]. It will be interesting future work to investigate the relationships between these different approaches and ownership type systems.

Chapter 3

Tool Support

This chapter is split into two parts: Sec. 3.1 discusses type checkers for Generic Universe Types and Sec. 3.2 discusses defaulting and the automatic inference of ownership modifiers.

3.1 Type Checkers

3.1.1 MultiJava and JML

We integrated Generic Universe Types into the MultiJava compiler [51] on which the JML compiler [112, 113, 115] is built. In this section, we describe the JML implementation without distinguishing whether a feature is actually implemented in MultiJava or JML.

The implementation of GUT in the JML compiler is well tested. We provide test cases for non-generic Universe types [144], the runtime support [174], the bytecode representation [184], and for Generic Universe Types [127, 194]. In total there are over 400 test cases for the different aspects of the Universe type system. We also developed a testing tool that helps in managing the test cases for different compilers [173].

In the rest of this section we use Universe type system to refer to both the generic and non-generic Universe type system.

3.1.1.1 Backwards Compatibility

The JML compiler recognizes the ownership modifiers `any` and `readonly` and treats them as synonyms.

To be able to use the JML compiler for *all* Java programs, the Universe type checking can be controlled in a fine-grained way by command line switches. Users can choose between four modes of operation: (1) the Universe type system is switched off completely; (2) the ownership annotations are parsed, but type checking is switched off; (3) Universe type checking is switched on; (4) Universe type checking is switched on and the necessary runtime checks are generated. In the context of JML, we always want to enforce the owner-as-modifier discipline to support the formal verification of the code. Therefore, we do not provide a separate switch to enable or disable the enforcement of the owner-as-modifier discipline.

With the concrete syntax of ownership modifiers used so far, some standard Java programs cannot be type-checked with the Universe type system because they use the keywords `peer`, `rep`, `readonly`, or `any` as identifiers. To avoid syntactic conflicts with the keywords `peer`, `rep`, `readonly`, and `any`, programmers can use an alternative syntax for ownership modifiers, where the keywords are preceded by a backslash, for example `/*@ \peer @*/`. Such modifiers are ignored by the compiler if the Universe type system is switched off (modes 1 and 2). This version of the modifiers can be used for Java API classes that should be usable either with or without enabled Universe type system. Finally, it is possible to use `peer`, `rep`, `readonly`,

`any`, and `pure` without enclosing comments, as is done in the examples in this thesis. However, programs with this concise syntax cannot be compiled by standard Java compilers.

3.1.1.2 Bytecode Representation of Ownership Information

To support separate compilation of programs, we store the ownership information in the bytecode [184]. We support two formats: (1) as custom bytecode annotations, which can store ownership format in a custom format; (2) as Java 5 annotations, which can be stored in the bytecode in a standardized format. The advantage of format (1) is that it works for all versions of Java. Format (2) is only compatible with Java 5 and greater, but has the advantage that the storage format is standard.

3.1.1.3 Runtime Checks

Like standard Java, the Universe type system requires runtime checks for downcasts and array updates. Besides the plain Java types, these checks have to compare the ownership information. Ownership information at runtime is also necessary to evaluate `instanceof` expressions.

The necessary runtime checks can be expressed as assertions in terms of the ghost field `owner`. The JML runtime assertion checker handles ghost fields by adding a normal field to the declaring class and appropriate get and set methods. However, this approach works only for ghost fields of classes that are compiled by the JML compiler, which is not the case for `Object`. Neither modifying `Object`'s source code nor patching its class file is an option since the distribution of a modified version of `Object` violates the Sun license terms.

We solve this problem by storing ownership information externally in a global hashtable. This table maps objects to their owner object. For array objects, we also store the ownership modifier of the element type in the hashtable. This information is used in the runtime checks of array updates. We use Java's weak references to ensure that storing a reference in the hashtable does not affect garbage collection (see [174] for details).

Conceptually the `owner` field is set by `Object`'s default constructor, which would take the owner object as argument. Since we cannot modify the implementation of `Object`, we add a new object to the ownership hashtable at two places: (1) before the first statement of the constructor, which ensures that the ownership information of the new object is available during the execution of the constructor; (2) after the `new` expression, which ensures that the new object is added even if the constructor was not compiled by the JML compiler. With this solution, it is still possible to create objects that are not added to the ownership hashtable if the `new` expression occurs in a class that is not compiled by the JML compiler. As the ownership of an object is determined by the `new` expression, if the `new` expression is not compiled by the JML compiler, no owner object can be determined. A Java system property is used to control whether runtime checks for such objects always pass or always fail.

We implemented the runtime checks for downcasts, array updates, and `instanceof` expressions as additional bytecode instructions generated by the compiler. As future work, we plan to adapt JML's runtime assertion checker to map accesses to the `owner` ghost field to accesses to the ownership hashtable.

For downcasts and array updates, the result of a failed check can be controlled by Java system properties. The options range from throwing an exception to only reporting the error on the console.

The checks for downcasts and `instanceof` are comparisons of the corresponding entries in the ownership hashtable. Like Java, the Universe type system has covariant array subtyping (see Sec. 2.5.3). For instance, `rep peer Object[]` is a subtype of `rep any Object[]`. Therefore,

an array variable with a static any element type could, at runtime, contain an array with peer element type. Consequently, updating such an array requires a runtime check that the reference assigned to the array element actually is a peer reference. The ownership hashtable stores the element ownership modifier of each array. This modifier is used to check whether the owner of the object on the right-hand side of the update conforms to the element type of the array.

3.1.1.4 Generic Types in JML

MultiJava was developed before Java 5 was available and only supported earlier versions of Java. A master's project [42] adapted a draft release of generic types in MultiJava and JML.

We improved the support for Java 5 in a semester project [127]. One big problem was that the MultiJava and JML compiler could not be executed on a Java 5 virtual machine. Also, some details about the handling of generic types changed between the draft release handled by [42] and the final release of Java 5. The semester project also added support for wildcards, including capture conversion, raw types, and the necessary unchecked warnings.

This support for genericity in MultiJava allowed us to implement Generic Universe Types in a master's project [194]. We also improved the runtime checks to support GUT [156].

3.1.1.5 Eclipse Integration

Setting up a MultiJava and JML installation is a quite complex task and difficult for newcomers. Also, the command line interface is not attractive to developers used to modern Integrated Development Environments (IDE).

To remedy this situation, we developed a JML plug-in for the Eclipse IDE (see `http://www.eclipse.org/`). This tool was developed as a semester project [19] and later improved, for example during the master's thesis [82].

The JML checker, runtime assertion checking (RAC) compiler, and specification tester can be invoked from within Eclipse and comfortable configuration dialogs can be used to set the different options. Error messages are parsed and displayed in a separate window and as markers in the code. Code with RAC and the specification tester can be executed directly from Eclipse. We also provide code templates that make entering specifications easy. The JML reference manual [115], the command-line help, and other introductory documents are integrated into the Eclipse help system.

3.1.2 Other Compilers and Languages

The annotation syntax of Java 5 is not flexible enough to allow type annotations on every occurrence of a type. The Java Specification Request (JSR) 308 [76] is remedying this problem and is scheduled to be included in Java 7. With JSR 308 it is possible to annotate all occurrences of a type. The developers of JSR 308 also provide a framework that simplifies the development of type checkers [157].

In a semester project [151] we developed a type checker for Generic Universe Types using the checker framework for JSR 308. Using options to the checker the owner-as-modifier discipline can be optionally enforced.

The annotation syntax of Scala [153] is very powerful and writing pluggable annotation processors for the compiler is possible. Using this annotation mechanism we investigated using finer grained Universe annotations [175]. We also developed an annotation processor for Generic Universe Types, including runtime checks for casts and instanceofs [180].

The Extended Static Checker for Java version 2 (ESC/Java2) [53] is a tool to find errors in Java source code that uses JML annotations. In a semester project [188] we extended the type checker of ESC/Java2 with checks for the Universe type system. As ESC/Java2 does not support genericity, we could not implement Generic Universe Types in it.

The language Eiffel [129] is popular for the formal development of software. In a semester project [169] we investigated the advanced features of Eiffel, e.g., expanded types and agents, and concluded that an adapted version of the Universe type system is possible.

3.1.3 Experience

We successfully use the JML plug-in for Eclipse in courses at ETH Zurich and it is also integrated into the Mobius Program Verification Environment. It is also the basis for the integration of the inference tools into Eclipse, which we will discuss later in this chapter.

We compared the expressiveness of the Universe type systems and other type systems [145] using the popular design patterns [83]. The flexibility of of `any` references proved to be very valuable in many design patterns, for example, in the flyweight pattern where the instances can be shared and protected by the owner-as-modifier discipline.

We also performed a case study on a commercial application [92]; it was possible to apply ownership encapsulation for the data structures of the application.

Finally, we were also able to use a system inspired by the owner-as-modifier discipline to enforce the applet isolation for JavaCard systems [139, 67].

The Generic Universe Types checker is integrated into current releases of MultiJava (`http://www.multijava.org/`) and JML (`http://www.jmlspecs.org/`).

The other tools we developed are available at the website `http://www.pm.inf.ethz.ch/research/universes/tools/`.

3.2 Universe Type Inference

Helping software engineers to transition from unannotated programs to code that uses an ownership type system is a very important task to allow ownership type systems to be used in everyday programs.

This task is simplified by default modifiers that are used wherever ownership modifiers are missing. We discuss defaulting in Sec. 3.2.1.

Our defaulting helps in annotating existing programs, but creates a “flat” structure. Usual type inference is concerned with finding the most general typing for the expressions in a program and there is a best solution: the most general typing that is valid for the expressions. Inferring ownership annotations is different: ownership annotations express design intent and there are multiple different ownership structures that are all correctly typed, but express different encapsulation properties. For realistic examples, there are many possible annotations and deciding which annotation is the best one depends on the intended design. Ownership inference needs to rely on heuristics to find desirable structures and needs to easily allow the developer to guide the inference.

In Sec. 3.2.2 we discuss static and runtime inference of ownership modifiers that help a programmer to find a desirable ownership topology. Finally, in Sec. 3.2.3 we discuss related work.

3.2.1 Default Ownership Modifiers

If the ownership modifiers are omitted in a type declaration, then a default is used. This default is normally `peer`, but there are a few exceptions, described below.

Using `peer` creates a "flat" ownership structure: all objects are owned by the root object. Viewpoint adaptation will never result in lost ownership and field updates and non-pure method calls are possible on `peer` references. Using `peer` as the default modifier has the advantage that most existing Java programs compile without changes.

Local Variables. The ownership modifier of local variable declarations is propagated from the initializer expression. If no initializer is present, the other defaults are applied.

Immutable Types. The ownership modifier of immutable types defaults to `any`. As such objects cannot be modified, their location in the heap is irrelevant. Currently, the set of immutable types includes only the Java wrapper types for primitive types (e.g., `java.lang.Integer` and `java.lang.Long`), and the classes `java.lang.String`, `java.lang.Class`, and `java.math.BigInteger`. An additional class modifier could be introduced to mark immutable types.

Pure Methods. The default modifier for explicit formal parameters to a `pure` method (but not for the implicit receiver, `this`) is `any`. Pure methods should be callable on any receiver type; using a modifier other than `any` would limit the applicability of the method. Pure constructors use the same defaulting as non-pure methods; as constructors are always called on an object that is either in the same context or in the representation context, there is no need to use `any` as default.

Arrays. If, for a type that is an array of references, one of the two ownership modifiers is omitted, then the element type is used to determine the meaning of the ownership modifier. If the element type is a mutable type, then the specified modifier is taken to be the element modifier, and the array's modifier defaults to `peer`. If the element type is an immutable type, then the specified modifier is taken to be the array modifier, and the element modifier defaults to `any`.

For example, the type `any Object[]` is the same as `peer any Object[]`. A type `rep Integer[]` is the same as `rep any Integer[]`. Note that if one wants to specify a `rep` or `any` array of mutable references, one is thus forced to use two ownership modifiers, for example, `rep any Object[]`.

One-dimensional arrays of primitive types default to `peer`. For multi-dimensional arrays of primitive types a single ownership modifier is always taken to be the element modifier.

Casts and Instanceof. The type in casts and instanceof expressions is defaulted to the static type of the sub-expression. This default allows one to test only for Java types, if the ownership modifiers are irrelevant for the cast.

Exception Handling. The only possible type for `throws` and `catch` clauses is `any`. Missing ownership modifiers will be interpreted as `any`; explicit ownership modifiers other than `any` are forbidden. Exception handlers might need to be rewritten, if they directly modify an exception object. Such handlers should be rewritten to use exception chaining, that is, create a new exception object locally that contains a reference to the original exception.

Upper Bounds. The most consistent rule is to also use the `peer` ownership modifier for all missing modifiers in an upper bound.

However, this simplest scheme makes the transition to partially annotated programs harder by requiring annotations on the upper bounds, in order for the bounds to be flexible for general use. For class and method upper bounds, `any` could be used as default for all missing ownership modifiers. This would allow the more general use of these classes and methods. However, viewpoint adaptation might now introduce `lost` and field updates and non-pure method calls on type variables are then forbidden, if the owner-as-modifier discipline is enforced; therefore existing programs might no longer compile.

Currently, `any` is used as default modifier in upper bounds; a compiler option or class annotation could be introduced that tells the compiler to choose `peer` as default modifier.

Purity. A separate issue is determining which methods are side-effect free, according to Assumption 2.4.2. The default annotation is that a method might have side effects, i.e., that it is non-pure. A purity inference tool [87, 168] enables the inference of side-effect free methods.

Summary. More elaborate propagation of modifiers can further ease the partial annotation of programs, e.g., local type inference, as found in Scala and other languages, could further simplify the annotation process.

Using the above defaults allows us to compile most existing Java programs as GUT programs without changes. They also ease the compilation of partially annotated programs.

Defaulting eases the process of annotating existing source files, but results in a flat ownership structure; it does not help the programmer in finding desirable annotations that correspond to a correct topology or encapsulation discipline. We discuss ownership inference in the next section.

3.2.2 Universe Type Inference

In the following we sketch the approaches to Universe type inference that we investigated so far, but leave a detailed development as future work. We discuss relevant related work in Sec. 3.2.2.

3.2.2.1 Static Universe Type Inference

Static inference builds a constraint system and uses a SAT solver to find a possible ownership annotation. A traversal of the abstract syntax tree (AST) determines all the locations for which Universe modifiers need to be inferred. Then the AST is used to build a boolean formula that encodes the constraints between the different variables, for example, by recording that the type of the right-hand side of an assignment has to be a subtype of the type of the left-hand side. Then a SAT solver is used to find a possible assignment to the variables that satisfies the constraints. Different assignments to the variables represent different ownership structures. By assigning different weights, the SAT solver can be instructed to try to infer a more desirable structure. Static inference has the advantage of covering the whole source code, but has to rely on heuristics to try to find good ownership structures.

3.2.2.2 Runtime Universe Type Inference

Runtime inference traces program executions and finds ownership structures in the heap. During a program execution the references between objects are monitored and also the modifications between the objects are traced. This information is then used to build an Extended Object

Graph that contains the accumulated information of the program execution. A dominator calculation [5, 118] then produces the first approximation to the topology of the ownership hierarchy. Then the additional constraints of the owner-as-modifier discipline are used to structure the object graph. Finally, the ownership hierarchy in the Extended Object Graph is used to determine Universe modifiers that can be used as annotations in the source code. This approach gives the deepest ownership hierarchy that is possible for the observed object graph. However, good code coverage is needed to produce results that are valid for the whole program.

3.2.2.3 Tool Support

We first investigated static inference using a Prolog database [106] and later based the tool on a SAT solver [147].

The runtime inference uses a Java Virtual Machine Tooling Interface tracing agent to monitor the execution of programs and then build the Extended Object Graph [124, 16, 65].

The static and runtime inference of Universe modifiers can be combined [82]. The runtime inference is used to find the deepest possible ownership structure, which is then used as weights for the static inference, ensuring that we have both, complete code coverage and a deep ownership structure.

Both tools automatically find possible annotations that the programmer can review and insert into the source code. Visualization of the inferred ownership structures improves the understanding of the behavior and structure of a program. These Eclipse GEF plug-ins [19, 130] are integrated with the static and runtime Universe type inference to visualize the inference results.

Finally, the implementation of a purity inference tool [87, 168] enables the inference of side-effect free methods. The static, runtime, and purity inference tools use a common XML schema for storing their results. A command line tool can be used to insert the annotations into existing Java sources and integration into Eclipse allows the easy modification of inference results.

3.2.3 Related Work

Ownership inference is recognized as an important research problem, and there were many recent contributions. However, recent work has not yet provided a satisfactory solution to infer ownership type annotations.

Alisdair Wren's work on inferring ownership [189] provided a theoretical basis for our work on runtime inference. It developed the idea of the Extended Object Graph and how to use the dominator as a first approximation of ownership. It builds on ownership types [46, 7, 23, 48] which uses parametric ownership and enforces the owner-as-dominator discipline. Several attempts at inferring ownership types [8, 4, 189] showed that the complexity of parametric ownership type systems makes inference difficult. The number of ownership parameters for parametric type systems is not fixed and is usually determined by the programmer, as is the number of type parameters for a class. Trying to automatically infer a good number of ownership parameters makes their system complex. No implementation is provided.

SafeJava [23] provides intra-procedural type inference and default types to reduce the annotation overhead. Agarwal and Stoller [4] describe a run-time technique that infers even more annotations. AliasJava [8] uses a constraint system to infer alias annotations. Another static analysis for ownership types resulted in a large number of ownership parameters [133]. In contrast, the simplicity of Universe types makes the mapping to static annotations possible. As

discussed in Sec. 3.2.1 the Universe type system also supports defaulting of ownership modifiers, allowing existing Java programs to be compiled without changes.

Milanova [131] presents preliminary results for the static inference of Universe Types. The tool applies the idea of [65] to static alias graphs instead of to runtime object graphs. Abi-Antoun and Aldrich [3, 1] present how runtime object graphs can be extracted from programs with ownership domain annotations.

Rayside et al. [165] present a dynamic analysis that infers ownership and sharing, but they do not map the results back to an ownership type system. Mitchell [132] analyzes the runtime structure of Java programs and characterizes them by their ownership patterns. The tool can work with heaps with 29 million objects and creates succinct graphs. The tool does not distinguish between read and write references and the results are not mapped to an ownership type system. The work on uniqueness and ownership inference [125] presents a static analysis to infer these program properties, without mapping the results to a type system. General type qualifier inference [88] presents relevant work for qualifier inference; however, applied to ownership, it would not help in the inference of the deepest or most desirable ownership structure, but infers a solution that satisfies all constraints, possibly a flat structure.

Daikon [74, 75, 77] is a tool to detect likely program invariants from program traces. Invariants are only enforced at the beginning and end of methods and therefore also snapshots are only taken at these spots. From these snapshots we cannot infer which references were used for reading and which were used for writing. Therefore we could not directly use Daikon, but our runtime inference tool has a similar architecture.

Work on the dynamic inference of abstract types [89] uses the flow of values in a program execution to infer abstract types. Yan et al. [192] use state machines to map implementation events to architecture events and thereby deduce architectures. Both approaches do not seem to be applicable to infer ownership information. Recent work by Poetzsch-Heffter et al. [158] and Quinonez et al. [164] on inferring ownership, respectively, immutability provide interesting research directions.

Chapter 4

Future Work

We split the future work into four areas: the formalization of the type system in Sec. 4.1, its expressiveness in Sec. 4.2, the inference of ownership information in Sec. 4.3, and, finally, the tool support in Sec. 4.4.

4.1 Formalization

In this thesis, we use the tool Ott [176] to ensure that our notation is consistent and that all operations are defined. We disambiguated the Ott input to enable the generation of the LaTeX code used throughout this dissertation and also to generate Isabelle [148] definitions for the formalization. The definition of GUT consists of 140 rules and 361 rule clauses and is disambiguated enough to create LaTeX and Isabelle output. However, the proofs that we present in App. A.2 are done manually, without the support of a mechanical proof checker.

Featherweight Java (FJ) [99] was proven sound in Isabelle [81]; we do not know of a mechanized proof of Featherweight Generic Java (FGJ). FJ and FGJ are purely functional models of Java. ClassicJava [80], Middleweight Java [20], and Lightweight Java [181] are two formalizations of subsets of Java that do model state. Lightweight Java is formalized in Ott and soundness is proved in Isabelle. However, we did not find a mechanical proof of soundness of an extension of FGJ with state.

It would be interesting future work to extend a formalization like Lightweight Java to include type genericity and then model Generic Universe Type on top of this formalization. Our existing definition in Ott should be a good starting point. An alternative might be to use SASyLF [9], a proof assistant particularly geared towards proofs in language theory. The advantage would be that instead of doing proofs in a lower-level proof assistant like Isabelle, the proofs are done in the same language as the language definition. The input language to SASyLF looks similar to Ott, and a conversion might be feasible.

Also further analysis using other formalizations is interesting. For example, modeling Generic Universe Types using parametric ownership type systems that support existential quantification [35], extending our existing work on non-generic Universe types [34], or expressing GUT using dependent classes [84] or constrained types [152], following our previous work [66].

Generic Universe Types establish an ownership topology and can enforce the owner-as-modifier encapsulation discipline. It will be very interesting future work to compare the benefits and restrictions of ownership type systems like GUT to other systems aimed at the formal verification of software, namely Spec♯'s dynamic ownership model [116, 17], separation logic [166], regional logic [15], and (implicit) dynamic frames [105, 179].

4.2 Expressiveness

The combination of JML and GUT allows us to use the type system to express common ownership patterns and resort to JML specifications for cases where the type system is not expressive enough. Using Generic Universe Types allows us to statically check more ownership situations that previously needed casts. It is interesting future work to find useful ownership patterns and make them efficiently checkable using a lightweight syntax.

Our work on dependent classes [66] and the comparison with parametric ownership type systems that support existential quantification [34] sparked our interest in combining these type systems. The Universe type system is lightweight, requiring less annotation overhead; dependent classes, constrained types, and parametric ownership require more annotations, but support finer-grained ownership structures. Creating a hybrid language that provides lightweight annotations for common ownership patterns, but allows to also use the flexibility of parametric ownership type systems is challenging future work.

Universe Types with Transfer (UTT) [143] support changes of ownership at runtime. Investigating the interaction between ownership transfer and type genericity will provide possible future work.

In this thesis we deal with an ownership system where every object is owned by at most one other object and where we have a single representation context per object. Multiple ownership [37] allows one object to have multiple owners and ownership domains [7] support multiple contexts per object. Encapsulation systems range from simple systems enforcing the owner-as-dominator or owner-as-modifier disciplines to fine-grained permission systems that allow the programmer to have a detailed control about the effects of a program. The investigation of other topological systems and encapsulation disciplines builds interesting future work.

The expressiveness of ownership type systems also depends on the underlying programming language. We imagine that adapting the Universe type system to the Bytecode Modeling Language (BML) [31] will provide interesting experience with low-level languages, whereas adapting it to the more advanced typing features of languages like Scala [153] will provide experience with the other end of the abstraction spectrum.

The owner-as-modifier discipline enforces that `any` references are not used to modify an object. However, other aliases to the same object may be used for modifications. Implementation patterns can be used to ensure that an object is immutable, i.e., directly after creating and initializing an object the only reference to it has to be assigned to a `readonly` variable, all read-write references to the object have to be abandoned, and casts involving this object have to be restricted. Ensuring that an object is immutable cannot be checked by our type system. Immutability is present in other type systems (e.g., Javari [185], Jimuva [90], IGJ [193], and Joe_3 [155]) and in a dynamic encoding of ownership called frozen objects [117]. The combination of GUT and immutability presents interesting future work.

Our assumption of method purity in Assumption 2.4.2 forbids all modifications of objects that exist in the pre-state of a pure method call. For some examples, e.g., caching of results, this is too strict. The work on *observational purity* [18, 146, 54] relaxes the definition of purity and allows the implementation of more patterns. It will be interesting to combine the work on observational purity with the owner-as-modifier discipline and try to derive an *owner-as-observable-modifier* discipline.

4.3 Ownership Inference

Type inference for ownership type systems will significantly ease their use. In Sec. 3.2 we discussed our first investigations of how Universe types can be inferred using a static and a runtime inference approach. Fully developing these techniques presents interesting future work.

Runtime Inference. Currently, the monitoring of realistic applications is prohibitively expensive due to the monitoring overhead and memory requirements to build the Extended Object Graph. The goal is to develop the runtime inference approach to successfully handle programs of a realistic size with reasonable time and memory requirements.

Current Java Virtual Machines do not preserve generic type information at runtime. It will be interesting to investigate whether some information from the Extended Object Graph can still be used to infer the ownership of type arguments.

Static Inference. Our static inference technique represents Universe types and the properties of a program as boolean formulas. There are different possibilities how to create these formulas and they influence the time and memory requirements of the inference and the quality of the results. The goal is to systematically deduce the best boolean representation for the static inference problem, possibly parameterized by time/memory trade-offs.

Also, currently one large boolean representation is generated for a program. This might have a negative impact on the time needed by the SAT solver. Different parts of a program can be known to be independent, for example, the type of a local variable that does not depend on a parameter or return type. It would be desirable to split the boolean representation into smaller independent parts that can be solved by the SAT solver, maybe even by parallel instances of the solver. This would also improve the performance in interactive applications of the program in an IDE.

The previous version of Generic Universe Types [60] adapted the main modifier of a type if a covariant change in a type argument occurred. This model proved to also be difficult to handle for the static inference, where additional constraints need to be introduced [194]. The formalization of GUT presented in this thesis does not create a dependency between the different ownership modifiers in a type and therefore should be easier to infer statically.

Combining Runtime and Static Inference. Runtime inference needs high code coverage to produce valid results and static inference can only use heuristics to produce deep ownership structures. The combination will allow us to get the best of both worlds: deep ownership structures found from execution traces and perfect coverage from the static inference.

Applying the inference tools to realistic case studies will reveal what inherent ownership structures exist in software and will allow one to further improve ownership type systems.

4.4 Tool Support

In Chapter 3 we discussed the extensive tool support that exists for the Universe type system. In this section we want to discuss some possible future developments.

Type Checkers. The current type checker for JML is implemented in the Common JML tools version 2, which only has limited support for Java 5 features. At the moment, multiple new implementations for JML are under development, for instance, to adapt it to type genericity [52]

and to support modern development environments and compilers. The type checker for Universe types using JSR 308 [151] might prove to be easily integrated into a new implementation of JML. The checker for Scala [180] was developed for version 2.6 of the Scala compiler. It needs to be adapted to newer language and compiler versions.

Inference. We currently have first prototypes for the static and runtime inference of Universe types. Once the theory is fully developed, developing research prototypes is an important task and will allow us to perform evaluations on realistic applications.

Visualization. Ownership annotations express design intent about what encapsulation boundaries are desired. The runtime and static inference methods provide two approaches to support the software developer in finding good annotations. Visualizing the inferred ownership structure will allow the developer to decide whether the solution is correct and adjust it easily. The current tools provide simple visualizations of the ownership tree that need to be adapted to support realistic examples.

Work on ownership domains [1, 3] helps visualize the software architecture of a program. The visualization of the static and runtime inference provides similar results and soundness could be investigated. Also, how to visualize object structures using ownership can provide interesting challenges [149].

Annotation Handling. The different inference tools need to communicate with each other and the software developer in a simple and effective way. We need a format that will be used as the low-level information interchange format, possibly also across other tools that use Java annotations.

On top of this, one could provide tools for easy editing and merging of the ownership information with source code. Possible output formats are Java source code with special keywords that are only recognized by the Universe type system compiler, as Java source code using Java Modeling Language stylized comments, the JSR 308 [76] extended annotation syntax, and separate annotation files supported by some compilers.

Evaluation. Applying the inference techniques to realistic applications will provide insights into the inference techniques and for ways to improve ownership type systems. Examples for which the tools cannot determine an ownership structure can have two results: either the inference method needs to be improved or the ownership type system needs to be extended to express the ownership structure.

An evaluation will have to range from hand-crafted examples that contain a desired ownership structure to large case studies of existing software, for example taken from Java benchmarks like the DaCapo suite or SPECjvm2008.

It will also be interesting to investigate new uses of ownership type systems. For example, how they can be applied to component models like OSGi or the new Java Module mechanism.

Chapter 5

Conclusion

Practical encapsulation of object structures is an important challenge in programming language research. This thesis contributes to the design and rigorous formalization of ownership type systems, provides usable implementations, and reports on our experience of their usage.

We present Generic Universe Types (GUT), a lightweight ownership type system that combines type genericity and ownership. We separate the topological structure from the enforcement of the owner-as-modifier discipline, allowing for the separate development and reuse of these parts. The GUT system is rigorously developed and proven sound.

The goal in our formalization is to impose the minimal set of restrictions that are needed to produce a sound system. We allow the use of the modifiers `self` and `lost` in programs and develop a separate judgment to decide whether a program is reasonable. This minimalistic approach has the advantage of highlighting the different aspects of the system and the requirements needed for soundness. Alternatively, one could decide to use the modifiers `self` and `lost` only internally by forbidding them in the syntax. We chose this approach in our work on non-generic Universe types [55]. This alternative approach has the advantage that one could assume basic well-formedness conditions, which would simplify some proofs of GUT.

We support a limited form of covariance, using the `lost` modifier in type argument positions. As our formal language does not support local variables, the use of this additional subtyping relationship is limited. The limited covariance would be unnecessary in a system that supports variance, e.g., via wildcards or variance annotations.

We developed the type checker and runtime support for Generic Universe Types for JML and Scala, and developed a type checker using the JSR 308 annotation mechanism and checker framework. We also enabled the use of JML with Java 5 and integrated JML into the Eclipse IDE, two measures that make JML more widely usable. In particular, this allowed us to use GUT in the classroom.

Aliasing is a problem that future developers need to be aware of and thinking about ownership structures, even without using an ownership type system, helps to understand and structure code. In classes at ETH Zurich we have had positive experiences from presenting the Universe type system to students and we supervised many semester and master projects making students aware of the effects of aliasing and letting them help to work on a solution.

During these projects we also did case studies to evaluate the use of Generic Universe Types and the tools we developed. We conclude that GUT allows the elegant expression of many interesting ownership structures.

Appendix A

Properties and Proofs

A.1 Properties

This section gives additional lemmas that are needed for the proofs. See Sec. 2.3.5 and Sec. 2.4.6 for the main properties of the topological system and the encapsulation system, respectively.

To simplify the notation, the intended scope of a quantifier is from the point of its introduction to the end of the formula; for example, the formula $\exists x.A \wedge \exists y.B \wedge \exists z.C$ can be read as $\exists x.(A \wedge (\exists y.(B \wedge \exists z.C)))$.

A.1.1 Viewpoint Adaptation

A.1.1.1 Adaptation from a Viewpoint Auxiliary Lemma

For the proof of Lemma 2.3.28 we use the following auxiliary lemma. According to Def. 2.2.15 the static type assignment judgment $h, {}^{r}\Gamma \vdash v : {}^{s}T'$ consists of two parts: the dynamization of the static type is assignable to the value and if the static type has `self` as main modifier, the address corresponds to the current object in the environment. The following lemma is used in the proof of the first part:

Lemma A.1.1 (Adaptation from a Viewpoint Auxiliary Lemma)

$$\left.\begin{array}{l} h, {}^{r}\Gamma \vdash \iota : {}^{s}N \\ h, {}^{r}\Gamma \vdash {}^{s}N, {}^{s}T; \left(\overline{{}^{s}T}/\overline{X}, \iota\right) = {}^{r}\Gamma' \end{array}\right\} \Longrightarrow \begin{array}{l} \exists \overline{o_\iota}, {}^{r}T.\ \mathrm{dyn}\left({}^{s}T, h, {}^{r}\Gamma', \overline{o_\iota}\right) = {}^{r}T \wedge \\ \exists {}^{s}T'.\ \left({}^{s}N \rhd {}^{s}T\right)\left[\overline{{}^{s}T}/\overline{X}\right] = {}^{s}T' \wedge \\ \exists \overline{o_\iota}', {}^{r}T'.\ \mathrm{dyn}\left({}^{s}T', h, {}^{r}\Gamma, \overline{o_\iota}'\right) = {}^{r}T' \wedge \\ \qquad {}^{r}T = {}^{r}T' \end{array}$$

This lemma expresses that the dynamizations of the static types ${}^{s}T$ and ${}^{s}T'$ in the two different viewpoints result in the same runtime types. Note how we can choose suitable substitutions for the `lost` modifier, i.e., the static type after viewpoint adaptation might contain more `lost` ownership information.

The proof of Lemma A.1.1 runs by induction on the shape of static type ${}^{s}T$. The base case deals with type variables and non-generic types. The induction step considers generic types, assuming that the lemma holds for the actual type arguments. Each of the cases is done by a case distinction on the main modifiers of ${}^{s}N$ and ${}^{s}T$. The proof can be found in Sec. A.2 on page 123.

A.1.1.2 Adaptation to a Viewpoint Auxiliary Lemma

The proof of Lemma 2.3.29 is analogous to the proof for Lemma 2.3.28 and the static type to value judgment is again split into two parts. The following auxiliary lemma is used for the first part:

Lemma A.1.2 (Adaptation to a Viewpoint Auxiliary Lemma)

$$\left.\begin{array}{l} h, {}^{r}\Gamma \vdash \iota : {}^{s}N \\ ({}^{s}N \rhd {}^{s}T)\left[\overline{{}^{s}T}/\overline{X}\right] \;=\; {}^{s}T' \\ \texttt{lost} \notin {}^{s}T' \\ h, {}^{r}\Gamma \vdash {}^{s}N, {}^{s}T; \left(\overline{{}^{s}T}/\overline{X}, \iota\right) \;= {}^{r}\Gamma' \end{array}\right\} \Longrightarrow \begin{array}{l} \exists {}^{r}T.\;\; \mathrm{dyn}\left({}^{s}T, h, {}^{r}\Gamma', \emptyset\right) \;= {}^{r}T \;\wedge \\ \exists {}^{r}T'.\;\; \mathrm{dyn}({}^{s}T', h, {}^{r}\Gamma, \emptyset) \;= {}^{r}T' \;\wedge \\ \qquad {}^{r}T = {}^{r}T' \end{array}$$

This lemma expresses that the dynamization of the static types ${}^{s}T$ and ${}^{s}T'\left[\overline{{}^{s}T}/\overline{X}\right]$ in the two different viewpoints results in the same runtime types. Note that in this lemma we use empty substitutions for `lost` when determining the runtime types (the requirement $\texttt{lost} \notin {}^{s}T'$ implies that also $\texttt{lost} \notin {}^{s}T$, see Lemma A.1.4). The proof can be found in Sec. A.2 on page 127.

A.1.1.3 Viewpoint Adaptation and `self`

If after viewpoint adaptation a type contains `self` we know that the original type had to contain `self` also and that the viewpoint has `self` as main modifier.

Type ${}^{s}N$ needs to be well formed, because otherwise ${}^{s}T$ could be a type variable that is substituted by the type argument from ${}^{s}N$ and an ill-formed type could use `self` as type argument.

Lemma A.1.3 (Viewpoint Adaptation and `self`)

$$\left.\begin{array}{l} {}^{s}\Gamma \vdash {}^{s}N \;\; \mathsf{OK} \\ {}^{s}N \rhd {}^{s}T \;=\; {}^{s}T' \\ \texttt{self} \in {}^{s}T' \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} \texttt{self} \in {}^{s}T \\ {}^{s}N = \texttt{self \_<\_>} \end{array}\right.$$

Proof: a simple analysis of viewpoint adaptation. Requiring that ${}^{s}N$ is well formed ensures that the type arguments do not contain `self`. Type ${}^{s}T$ does not need to be well formed, all we need is that the viewpoint adaptation is defined. □

This lemma is used in the proofs of Lemma 2.3.28 and Lemma 2.3.29.

A simple consequence is that if a type does not contain `self`, then it will also not contain `self` after viewpoint adaptation with a well-formed type ${}^{s}N$.

A.1.1.4 Viewpoint Adaptation and `lost`

If a type does not contain `lost` after viewpoint adaptation, the type cannot contain `lost` before viewpoint adaptation.

Lemma A.1.4 (Viewpoint Adaptation and `lost`)

$$\left.\begin{array}{l} {}^{s}N \rhd {}^{s}T \;=\; {}^{s}T' \\ \texttt{lost} \notin {}^{s}T' \end{array}\right\} \Longrightarrow \texttt{lost} \notin {}^{s}T$$

Proof: a simple analysis of viewpoint adaptation. □

Note that in this case we do not need well-formed types.

This lemma is used in the proof of Lemma A.1.2 (see Sec. A.2 on page 127) to show that the type before viewpoint adaptation cannot contain `lost` if the type after does not.

A simple consequence is that if a type contains `lost` before viewpoint adaptation, the type will also contain `lost` after viewpoint adaptation.

A.1.2 Well-formedness Properties

A.1.2.1 Well-formedness and Viewpoint Adaptation

Well-formed types stay well formed after viewpoint adaptation, if certain conditions about the new viewpoint hold.

Lemma A.1.5 (Well-formedness and Viewpoint Adaptation)

$$\begin{array}{l}
\mathrm{ClassDom}(C) = \overline{X_k} \qquad \mathrm{ClassBnds}(C) = \overline{{}^sN_k} \\
{}^s\Gamma = \left\{ \overline{X_k \mapsto {}^sN_k}\,,\ \overline{X'_l \mapsto {}^sN'_l}\ ;\ \texttt{this} \mapsto \texttt{self}\ C\texttt{<}\overline{X_k}\texttt{>}, _ \right\} \\
{}^s\Gamma \vdash {}^sT\ \ \mathsf{OK} \\
{}^s\Gamma' \vdash {}^sN\ \ \mathsf{OK} \qquad \mathrm{ClassOf}({}^sN) = C \\
\left({}^sN \rhd \overline{{}^sN'_l}\right) \left[\overline{{}^sT_l}/\overline{X'_l}\right] = \overline{{}^sN''_l} \\
{}^s\Gamma' \vdash \overline{{}^sT_l}\ \ \mathsf{strictly\ OK} \qquad {}^s\Gamma' \vdash \overline{{}^sT_l} <:\ \overline{{}^sN''_l} \\
\Longrightarrow \\
\exists {}^sT'.\ \ ({}^sN \rhd {}^sT) \left[\overline{{}^sT_l}/\overline{X'_l}\right] = {}^sT' \wedge \\
\qquad {}^s\Gamma' \vdash {}^sT'\ \ \mathsf{OK}
\end{array}$$

The proof can be found in Sec. A.2 on page 130 and runs by induction on the shape of ${}^sT$.

This lemma is used to prove Corollary A.1.6, Corollary A.1.7, and Corollary A.1.8.

A.1.2.2 Well-formedness and VP Adaptation for Fields

Field types are well formed after viewpoint adaptation.

Corollary A.1.6 (Well-formedness and VP Adaptation for Fields)

$$\left.\begin{array}{l}
\vdash P\ \ \mathsf{OK} \\
\mathrm{FType}(C, f) = {}^sT \\
{}^s\Gamma \vdash {}^sN\ \ \mathsf{OK} \qquad \mathrm{ClassOf}({}^sN) = C
\end{array}\right\} \Longrightarrow \begin{array}{l} \exists {}^sT'.\ \ \mathrm{FType}({}^sN, f) = {}^sT' \wedge \\ \qquad {}^s\Gamma \vdash {}^sT'\ \ \mathsf{OK} \end{array}$$

This corollary is a simple application of Lemma A.1.5: from $\vdash P\ \ \mathsf{OK}$ we know that the declared field type ${}^sT$ was checked for well-formedness in a suitable environment (defined in rule WFC_DEF, Def. 2.3.14). There are no method type variables and we therefore use empty sequences for them. □

This corollary is used in the soundness proof (Sec. A.2 on page 109 and Sec. A.2 on page 111) to show that the viewpoint-adapted field types are well formed and in the proof of Lemma A.1.16 to argue that expression types are well formed.

A.1.2.3 Well-formedness and VP Adaptation for Methods

Method signatures are well formed after viewpoint adaptation.

Corollary A.1.7 (Well-formedness and VP Adaptation for Methods)

$$\begin{array}{l}
\vdash P\ \ \mathsf{OK} \\
\mathrm{MSig}(C, m) = _\ \texttt{<}\overline{X_l\ \texttt{extends}\ {}^sN_l}\texttt{>}\ {}^sT_1\ m\texttt{(}\overline{{}^sT'_q\ pid}\texttt{)} \\
{}^s\Gamma \vdash {}^sN\ \ \mathsf{OK} \qquad \mathrm{ClassOf}({}^sN) = C \\
\left({}^sN \rhd \overline{{}^sN_l}\right) \left[\overline{{}^sT_l}/\overline{X_l}\right] = \overline{{}^sN'_l} \\
{}^s\Gamma \vdash \overline{{}^sT_l}\ \ \mathsf{strictly\ OK} \qquad {}^s\Gamma \vdash \overline{{}^sT_l} <:\ \overline{{}^sN'_l} \\
\Longrightarrow \\
\exists {}^sT'_1, \overline{{}^sT''_q}.\ \ \mathrm{MSig}\left({}^sN, m, \overline{{}^sT_l}\right) = _\ \texttt{<}\overline{X_l\ \texttt{extends}\ {}^sN'_l}\texttt{>}\ {}^sT'_1\ m\texttt{(}\overline{{}^sT''_q\ pid}\texttt{)} \wedge \\
\qquad {}^s\Gamma \vdash \overline{{}^sN'_l}\,,\ {}^sT'_1\,,\ \overline{{}^sT''_q}\ \ \mathsf{OK}
\end{array}$$

This corollary is a simple application of Lemma A.1.5: from $\vdash P\ \ \mathsf{OK}$ we know that the method signature was checked for well-formedness in a suitable environment (defined in rule WFMD_DEF, Def. 2.3.16). The conditions for the method type variables are the same as needed for the lemma. Then the lemma is simply instantiated for the upper bound, return, and parameter types. □

This corollary is used in the soundness proof (Sec. A.2 on page 113) to show that the types in the viewpoint-adapted method signature are well formed.

A.1.2.4 Well-formedness and VP Adaptation for Class Upper Bounds

Class upper bounds are well formed after viewpoint adaptation.

Corollary A.1.8 (Well-formedness and VP Adaptation for Class Upper Bounds)

$$\left.\begin{array}{l} \vdash P\ \ \mathsf{OK} \\ \mathrm{ClassBnds}(C) = \overline{{}^{s}N} \\ {}^{s}\Gamma \vdash {}^{s}N\ \ \mathsf{OK} \qquad \mathrm{ClassOf}({}^{s}N) = C \end{array}\right\} \Longrightarrow \begin{array}{l} \exists \overline{{}^{s}N'}.\ \mathrm{ClassBnds}({}^{s}N) = \overline{{}^{s}N'} \wedge \\ \qquad {}^{s}\Gamma \vdash \overline{{}^{s}N'}\ \ \mathsf{OK} \end{array}$$

This corollary is a simple application of Lemma A.1.5: from $\vdash P\ \ \mathsf{OK}$ we know that the upper bound types $\overline{{}^{s}N}$ were checked for well-formedness in a suitable environment (defined in rule WFC_DEF, Def. 2.3.14). There are no method type variables and we therefore use empty sequences for them. □

This corollary is used in Sec. A.2 on page 132 to show that the viewpoint-adapted class bounds are well formed.

A.1.2.5 Static Well-formedness Implies Dynamization

A well-formed static type can be dynamized into a runtime type. However, this runtime type might not be strictly well formed, because the substitution for `lost` might not be consistent with the upper bounds.

Lemma A.1.9 (Static Well-formedness Implies Dynamization)

$$\left.\begin{array}{ll} 1. & \vdash P\ \ \mathsf{OK} \\ 2. & h, {}^{r}\Gamma : {}^{s}\Gamma\ \ \mathsf{OK} \\ 3. & {}^{s}\Gamma \vdash {}^{s}T\ \ \mathsf{OK} \end{array}\right\} \Longrightarrow\ \exists \overline{o_l}.\ \exists {}^{r}T.\ \ \mathrm{dyn}({}^{s}T, h, {}^{r}\Gamma, \overline{o_l}) = {}^{r}T$$

From 3. we know that the free variables in ${}^{s}T$ are contained in ${}^{s}\Gamma$. From 2. we know that those variables are either found in the heap or the runtime environment. From 1. we get general well-formedness of the program, in particular, that subtyping is well formed. We can choose an arbitrary substitution for the occurrences of `lost` in ${}^{s}T$ and arrive at a runtime type ${}^{r}T$. □

However, note that this runtime type is not guaranteed to be well formed, as the substitutions we choose for `lost` might not fulfill the requirements of the corresponding upper bounds!

This lemma is used to show soundness of the `null` expression and object creation, where well-formed static types are assigned to the $\mathtt{null}_a$ value.

A.1.2.6 Strict Well-formedness and dyn

A strictly well-formed static type can be turned into a runtime type, without a substitution for `lost`. The resulting runtime type is a strictly well-formed runtime type.

Note that an arbitrary viewpoint address for the strict well-formedness of ${}^{r}T$ can be used; a strictly well-formed static type does not use `rep` in an upper bound and therefore at runtime does not depend on a viewpoint. In particular, the viewpoint address does not need to be in the domain of the heap h.

Lemma A.1.10 (Strict Well-formedness) Strict static well-formedness implies dynamization and runtime well-formedness:

$$\left.\begin{array}{l} \vdash P \ \ \mathsf{OK} \\ h, {}^{r}\Gamma : {}^{s}\Gamma \ \ \mathsf{OK} \\ {}^{s}\Gamma \vdash {}^{s}T \ \ \mathsf{strictly\ OK} \end{array}\right\} \Longrightarrow \begin{array}{l} \exists {}^{r}T.\ \ \mathrm{dyn}({}^{s}T, h, {}^{r}\Gamma, \emptyset) = {}^{r}T \wedge \\ \qquad h,\ \_ \vdash {}^{r}T \ \ \mathsf{strictly\ OK} \end{array}$$

The proof can be found in Sec. A.2 on page 131 and runs by induction on the shape of the static type ${}^{s}T$.

This lemma is used to show soundness of object creations in Sec. A.2 on page 108 and method calls in Sec. A.2 on page 113.

A.1.2.7 Correct Checking of Class Upper Bounds

The static checks we perform for class type arguments ensure that the corresponding runtime type arguments are subtypes of their upper bounds.

Lemma A.1.11 (Correct Checking of Class Upper Bounds)

$$\left.\begin{array}{l} \vdash P \ \ \mathsf{OK} \\ h, {}^{r}\Gamma : {}^{s}\Gamma \ \ \mathsf{OK} \\ {}^{s}\Gamma \vdash {}^{s}N \ \ \mathsf{strictly\ OK} \\ \mathrm{dyn}({}^{s}N, h, {}^{r}\Gamma, \emptyset) = {}^{o}\iota \ \ C\texttt{<}\overline{{}^{r}T_k}\texttt{>} \\ \mathrm{ClassBnds}\big(h, \_, {}^{o}\iota \ \ C\texttt{<}\overline{{}^{r}T_k}\texttt{>}, \emptyset\big) = \overline{{}^{r}T'_k} \end{array}\right\} \Longrightarrow h \vdash \overline{{}^{r}T_k} <: \ \overline{{}^{r}T'_k}$$

The proof can be found in Sec. A.2 on page 132.

This lemma is used to prove Lemma A.1.10 (see Sec. A.2 on page 131).

A.1.2.8 Correct Checking of Method Upper Bounds

The static checks we perform for method type arguments ensure that the corresponding runtime type arguments are subtypes of the declared upper bounds.

Lemma A.1.12 (Correct Checking of Method Upper Bounds)

$$\left.\begin{array}{l} \vdash P \ \ \mathsf{OK} \\ h, {}^{r}\Gamma : {}^{s}\Gamma \ \ \mathsf{OK} \\ ({}^{s}N \rhd {}^{s}T_0)\left[\overline{{}^{s}T}/\overline{X}\right] \ = \ {}^{s}T' \\ \texttt{lost} \notin {}^{s}T' \\ {}^{s}\Gamma \vdash {}^{s}T,\ {}^{s}T' \ \ \mathsf{OK} \\ {}^{s}\Gamma \vdash {}^{s}T <: \ {}^{s}T' \\ h, {}^{r}\Gamma \vdash \iota : \ {}^{s}N \\ h, {}^{r}\Gamma \vdash {}^{s}N, {}^{s}T_0; \left(\overline{{}^{s}T}/\overline{X}, \iota\right) = {}^{r}\Gamma' \end{array}\right\} \Longrightarrow \begin{array}{l} \exists \overline{{}^{o}\iota}, {}^{r}T.\ \ \mathrm{dyn}({}^{s}T, h, {}^{r}\Gamma, \overline{{}^{o}\iota}) = {}^{r}T \wedge \\ \exists {}^{r}T'.\ \ \mathrm{dyn}\big({}^{s}T_0, h, {}^{r}\Gamma', \emptyset\big) = {}^{r}T' \wedge \\ \qquad h \vdash {}^{r}T <: \ {}^{r}T' \end{array}$$

The proof can be found in Sec. A.2 on page 133 and builds on Lemma A.1.23 and Lemma A.1.2.

This lemma is used to show soundness of method calls in Sec. A.2 on page 113.

A.1.2.9 Properties of Strictly Well-formed Static Types

A strictly well-formed static type does not use `rep` in the upper bounds of the class. If we know that the main modifier is `any`, then we can also exclude `peer` as modifier in the upper bound.

Lemma A.1.13 (Properties of Strictly Well-formed Static Types)

$$\left.\begin{array}{l} {}^{s}\Gamma \vdash u\ C\texttt{<}\overline{{}^{s}T}\texttt{>}\ \ \mathsf{strictly\ OK} \\ \mathrm{ClassBnds}(C)\ = \overline{{}^{s}N} \end{array}\right\} \Longrightarrow \texttt{rep} \notin \overline{{}^{s}N}$$

$$\left.\begin{array}{l} {}^{s}\Gamma \vdash \texttt{any}\ C\texttt{<}\overline{{}^{s}T}\texttt{>}\ \ \mathsf{strictly\ OK} \\ \mathrm{ClassBnds}(C)\ = \overline{{}^{s}N} \end{array}\right\} \Longrightarrow \texttt{peer} \notin \overline{{}^{s}N}$$

Note that the strictly well-formed static type judgment already forbids `lost` to appear in the upper bounds. The appearance of `self` in upper bounds is forbidden by the well-formed class declaration judgment.

The proof can be found in Sec. A.2 on page 134 and runs by an analysis of the definitions of strictly well-formed static type and the class bounds look-up function.

This lemma is used in the proof of Lemma A.1.10 (see Sec. A.2 on page 131).

A.1.2.10 Free Variables

The following two properties describe what type variables can be used in a superclass instantiation and in the upper bounds of a class.

The type variables that can be used in the instantiation of a superclass are a subset of the type variables of the subclass.

The type variables that can be used in the upper bounds of a class are a subset of the type variables of the class.

Lemma A.1.14 (Free Variables) in upper bounds and superclass instantiations

$$\left.\begin{array}{l} \vdash P\ \ \mathsf{OK} \\ C\texttt{<}\overline{X}\texttt{>} \sqsubseteq C'\texttt{<}\overline{{}^{s}T}\texttt{>} \end{array}\right\} \Longrightarrow \mathrm{free}\big(\overline{{}^{s}T}\big)\ \subseteq\ \overline{X}$$

$$\left.\begin{array}{l} \vdash P\ \ \mathsf{OK} \\ \mathrm{ClassDom}(C)\ = \overline{X} \\ \mathrm{ClassBnds}(C)\ = \overline{{}^{s}N} \end{array}\right\} \Longrightarrow \mathrm{free}\big(\overline{{}^{s}N}\big)\ \subseteq\ \overline{X}$$

The proof can be found in Sec. A.2 on page 134 and runs by an induction on the shape of the derivation of subclassing relation, respectively a simple analysis of class well-formedness.

This lemma is used in Sec. A.2 on page 138 and Sec. A.2 on page 132 to show that the applications of sdyn are defined.

A.1.2.11 Strict Well-formedness Implies Well-formedness

A strictly well-formed static type is always also a well-formed static type.

Lemma A.1.15 (Strict Well-formedness Implies Well-formedness)

$${}^{s}\Gamma \vdash {}^{s}T\ \ \mathsf{strictly\ OK} \quad\Longrightarrow\quad {}^{s}\Gamma \vdash {}^{s}T\ \ \mathsf{OK}$$

A simple investigation of Def. 2.3.12 and Def. 2.3.11 gives that the checks for strict well-formedness are stricter than the corresponding checks of the well-formedness check. □

A.1.2.12 Expression Types are Well Formed

If a static type can be assigned to an expression, we know that the type is well formed.

Lemma A.1.16 (Expression Types are Well Formed)

$$\left.\begin{array}{ll} 1. & \vdash P \ \ \mathsf{OK} \\ 2. & {}^{s}\Gamma \ \ \mathsf{OK} \\ 3. & {}^{s}\Gamma \vdash e : {}^{s}T \end{array}\right\} \Longrightarrow {}^{s}\Gamma \vdash {}^{s}T \ \ \mathsf{OK}$$

This is done by a simple analysis of the derivation tree of 3. The type rules TR_SUBSUM, TR_NULL, TR_NEW, and TR_CAST have an explicit check for well formedness of the assigned type. For TR_VAR we use the knowledge from 2. to deduce that the assigned type is well formed. For rules TR_READ, TR_WRITE, respectively TR_CALL we apply Corollary A.1.6 respectively Corollary A.1.7 to deduce that the field type respectively method return type are well formed. □

This lemma is used in the proof of Theorem 2.3.26. In general, the supertype of a well-formed type is not necessarily also well formed (see Sec. A.1.6.2 for a discussion). In the soundness proof we need that the type that is assigned to an expression is well formed.

A.1.3 Ordering Relations

This section is a collection of properties about the static and runtime ordering relations.

A.1.3.1 Subclassing: Superclass Instantiation Uses Strictly Well-formed Types

In a subclassing relationship, the instantiation of a superclass uses strictly well-formed types.

Lemma A.1.17 (Subclassing: Strict Superclass Instantiation)

$$\left.\begin{array}{l} \vdash P \ \ \mathsf{OK} \\ C\texttt{<}\overline{X}\texttt{>} \sqsubseteq C'\texttt{<}\overline{{}^{s}T}\texttt{>} \end{array}\right\} \Longrightarrow \ {}^{s}\Gamma \vdash \overline{{}^{s}T} \ \ \mathsf{strictly\ OK}$$

$$\begin{array}{l} \text{where } {}^{s}\Gamma = \left\{\overline{X_k \mapsto {}^{s}N_k} \ ; \ \texttt{this} \mapsto \texttt{self}\ C\texttt{<}\overline{X_k}\texttt{>}, \_\right\} \\ \quad \text{and } \mathrm{ClassDom}(C) \ = \overline{X_k} \text{ and } \overline{X} = \overline{X_k} \\ \quad \text{and } \mathrm{ClassBnds}(C) \ = \overline{{}^{s}N_k} \end{array}$$

The proof can be found in Sec. A.2 on page 135 and runs by an induction on the derivation of subclassing.

The lemma is used in the proof of Lemma A.1.10.

As a simple corollary we have that the superclass instantiation does not use `self`:

Corollary A.1.18 (Subclassing does not Introduce `self`)

$$\left.\begin{array}{l} \vdash P \ \ \mathsf{OK} \\ C\texttt{<}\overline{X}\texttt{>} \sqsubseteq C'\texttt{<}\overline{{}^{s}T}\texttt{>} \end{array}\right\} \Longrightarrow \ \texttt{self} \notin \overline{{}^{s}T}$$

This is a simple consequence of Lemma A.1.17 and Def. 2.3.12.

This corollary is used in the proof of Lemma A.1.21 (see Sec. A.2 on page 137).

A.1.3.2 Subclassing: Superclass Instantiation Uses Subtypes of the Upper Bounds

In a subclassing relationship, the instantiation of a superclass uses types that are subtypes of the upper bounds.

Lemma A.1.19 (Subclassing: Bounds Respected)

$$\left.\begin{array}{l} \vdash P \ \mathsf{OK} \\ C\texttt{<}\overline{X}\texttt{>} \sqsubseteq C'\texttt{<}\overline{{}^{s}T}\texttt{>} \\ \mathrm{ClassBnds}\big(\texttt{self}\ C'\texttt{<}\overline{{}^{s}T}\texttt{>}\big) = \overline{{}^{s}N} \end{array}\right\} \Longrightarrow \ {}^{s}\Gamma \vdash \overline{{}^{s}T} <: \ \overline{{}^{s}N}$$

$$\text{where } {}^{s}\Gamma = \big\{\overline{X_k \mapsto {}^{s}N_k}\ ;\ \texttt{this} \mapsto \texttt{self}\ C\texttt{<}\overline{X_k}\texttt{>}, \_\big\}$$
$$\text{and } \mathrm{ClassDom}(C) = \overline{X_k} \text{ and } \overline{X} = \overline{X_k}$$
$$\text{and } \mathrm{ClassBnds}(C) = \overline{{}^{s}N_k}$$

The proof can be found in Sec. A.2 on page 136 and runs by induction on the derivation of subclassing.

The lemma is used in the proof of Lemma A.1.10.

A.1.3.3 Subclassing: Superclass Instantiation has Upper Bounds without `lost`

In a subclassing relationship, the upper bounds of the superclass do not contain `lost`.

Lemma A.1.20 (Subclassing: Bounds do not Contain `lost`*)*

$$\left.\begin{array}{l} \vdash P \ \mathsf{OK} \\ C\texttt{<}\overline{X}\texttt{>} \sqsubseteq C'\texttt{<}\overline{{}^{s}T}\texttt{>} \\ C \neq C' \\ \mathrm{ClassBnds}\big(\texttt{self}\ C'\texttt{<}\overline{{}^{s}T}\texttt{>}\big) = \overline{{}^{s}N} \end{array}\right\} \Longrightarrow \ \texttt{lost} \notin \overline{{}^{s}N}$$

The proof can be found in Sec. A.2 on page 136 and runs by induction on the derivation of subclassing.

The lemma is used in the proof of Lemma A.1.10.

Note that the lemma would not hold, if $C = C'$ was allowed. The uses of the lemma handle this case separately. Also note that from Lemma A.1.17 and Lemma A.1.4 we can also conclude that the declared upper bounds of class C' do not contain `lost`, i.e., $\mathrm{ClassBnds}(C') = \overline{{}^{s}N}'$ and $\texttt{lost} \notin \overline{{}^{s}N}'$.

A.1.3.4 Subtyping and `self`

If the `self` modifier does not appear in a subtype it can also not appear in a supertype. If `self` is the main modifier of the supertype, then it is also the main modifier of a subtype.

Lemma A.1.21 (Subtyping and `self`*)*

$$\left.\begin{array}{l} \vdash P \ \mathsf{OK} \\ {}^{s}\Gamma \ \mathsf{OK} \\ {}^{s}\Gamma \vdash {}^{s}T <: \ {}^{s}T' \\ \texttt{self} \notin {}^{s}T \end{array}\right\} \Longrightarrow \texttt{self} \notin {}^{s}T'$$

$$\left.\begin{array}{l} \vdash P \ \mathsf{OK} \\ {}^{s}\Gamma \ \mathsf{OK} \\ {}^{s}\Gamma \vdash {}^{s}T <: \ {}^{s}T' \\ \mathrm{om}({}^{s}T', {}^{s}\Gamma) = \texttt{self} \end{array}\right\} \Longrightarrow \mathrm{om}({}^{s}T, {}^{s}\Gamma) = \texttt{self}$$

The proof can be found in Sec. A.2 on page 137 and runs by induction on the derivation of subtyping.

The lemma is used in the soundness proof (Sec. A.2 on page 106), when we need to argue that applying the subsumption rule cannot introduce `self`, and in Sec. A.2 on page 139 to show that sub- and supertype agree on ${}^{r}\Gamma(\texttt{this})$.

A.1.3.5 Static Type Assignment to Values Preserves Subtyping

If a static type can be assigned to a value, also a supertype can be assigned to the value.

Lemma A.1.22 (Static Type Assignment to Values Preserves Subtyping)

$$\left.\begin{array}{l} \vdash P \ \ \mathsf{OK} \\ h, {}^{r}\Gamma : {}^{s}\Gamma \ \ \mathsf{OK} \\ {}^{s}\Gamma \vdash {}^{s}T, {}^{s}T' \ \ \mathsf{OK} \\ {}^{s}\Gamma \vdash {}^{s}T <: \ {}^{s}T' \\ h, {}^{r}\Gamma \vdash v : \ {}^{s}T \end{array}\right\} \Longrightarrow \ h, {}^{r}\Gamma \vdash v : \ {}^{s}T'$$

The proof can be found in Sec. A.2 on page 139 and is an application of Lemma A.1.23 and Lemma A.1.21.

The lemma is used in the proof of Theorem 2.3.26 for the subsumption rule.

A.1.3.6 dyn Preserves Subtyping

For the proof of Lemma A.1.22 we use the following lemma that expresses that if two static types are subtypes, then their dynamizations with some substitutions for `lost` results in runtime subtypes.

*Lemma A.1.23 (*dyn *Preserves Subtyping)*

$$\left.\begin{array}{l} \vdash P \ \ \mathsf{OK} \\ h, {}^{r}\Gamma : {}^{s}\Gamma \ \ \mathsf{OK} \\ {}^{s}\Gamma \vdash {}^{s}T, {}^{s}T' \ \ \mathsf{OK} \\ {}^{s}\Gamma \vdash {}^{s}T <: \ {}^{s}T' \end{array}\right\} \Longrightarrow \begin{array}{l} \exists \overline{o\iota}, {}^{r}T. \ \ \mathrm{dyn}({}^{s}T, h, {}^{r}\Gamma, \overline{o\iota}) = {}^{r}T \wedge \\ \exists \overline{o\iota}', {}^{r}T'. \ \ \mathrm{dyn}\left({}^{s}T', h, {}^{r}\Gamma, \overline{o\iota}'\right) = {}^{r}T' \wedge \\ \qquad h \vdash {}^{r}T <: \ {}^{r}T' \end{array}$$

The proof can be found in Sec. A.2 on page 138 and is an analysis of static and runtime subtyping.

A.1.3.7 Static Type Assignment to Values and Substitutions

In the static type assignment judgment (Def. 2.2.15), type variables are replaced by the corresponding runtime types in ${}^{r}\Gamma$. The corresponding substitutions can also be performed on the static types and the static type assignment judgment continues to work.

Lemma A.1.24 (Static Type Assignment to Values and Substitutions)

$$\left.\begin{array}{l} \vdash P \ \ \mathsf{OK} \\ h \vdash {}^{r}\Gamma(\texttt{this}) \ : \ {}^{o}\iota \ C\texttt{<}\overline{{}^{r}T_k}\texttt{>} \\ \mathrm{ClassDom}(C) = \overline{X_k} \\ \mathrm{dyn}\left(\overline{{}^{s}T_k}, h, {}^{r}\Gamma, \emptyset\right) = \overline{{}^{r}T_k} \\ {}^{s}T\left[\overline{{}^{s}T_k}/\overline{X_k}\right] = {}^{s}T' \end{array}\right\} \Longrightarrow \ h, {}^{r}\Gamma \vdash v : \ {}^{s}T \Longleftrightarrow h, {}^{r}\Gamma \vdash v : \ {}^{s}T'$$

The proof is a simple analysis of Def. 2.2.15 and Def. 2.2.14. The runtime type of `this` is used to substitute type variables. If these type variables are substituted by static types, whose dynamization corresponds to the runtime types, we can also assign that type. □

This lemma is used in the proof of method calls in Sec. A.2. Def. 2.3.17 ensures that overriding methods from superclass and subclass have compatible signatures, i.e., that they only differ by substitutions of type arguments for type variables. This lemma gives us that, if the static type of the superclass can be assigned to a value, then we can also assign the type from the subclass, and vice versa.

A.1.4 Runtime Behavior

A.1.4.1 Runtime Meaning of Ownership Modifiers

The following lemma connects the meaning of the static ownership modifiers and the runtime owner. It establishes that the intended meaning of ownership modifiers holds.

For `self` and `peer` references, the owner of the referenced object is the owner of the current object. For `rep` references, the owner of the referenced object is the current object. From `lost` and `any` references, we do not gain any information about the runtime ownership.

Lemma A.1.25 (Runtime Meaning of Ownership Modifiers)

$$\begin{array}{l}\text{If } \mathrm{dyn}({}^{s}N, h, {}^{r}\Gamma, \overline{\circ\iota}) = {}^{r}T \ \wedge \ h \vdash \iota : {}^{r}T \text{ then} \\ \quad \mathrm{om}({}^{s}N) = \texttt{self} \implies \mathrm{owner}(h, \iota) = \mathrm{owner}(h, {}^{r}\Gamma(\texttt{this})) \\ \quad \mathrm{om}({}^{s}N) = \texttt{peer} \implies \mathrm{owner}(h, \iota) = \mathrm{owner}(h, {}^{r}\Gamma(\texttt{this})) \\ \quad \mathrm{om}({}^{s}N) = \texttt{rep} \implies \mathrm{owner}(h, \iota) = {}^{r}\Gamma(\texttt{this})\end{array}$$

The proof is a case analysis and the application of the definition of dyn (Def. 2.2.14) and can be found in Sec. A.2 on page 141.

The lemma is used in the proof of the owner-as-modifier discipline (Theorem 2.4.7).

A.1.4.2 Equivalence of sdyn and dyn

Under certain conditions we can show that the result of simple dynamization sdyn is equal to dynamization dyn.

Lemma A.1.26 (Equivalence of sdyn *and* dyn*)*

$$\left.\begin{array}{l}\vdash P \ \mathsf{OK} \\ h, {}^{r}T : {}^{s}\Gamma \ \mathsf{OK} \\ {}^{s}\Gamma \vdash u \ C\texttt{<}\overline{{}^{s}T}\texttt{>} \ \mathsf{strictly\ OK} \\ \mathrm{dyn}\big(u \ C\texttt{<}\overline{{}^{s}T}\texttt{>}, h, {}^{r}\Gamma, \emptyset\big) = {}^{\circ}\iota \ C\texttt{<}\overline{{}^{r}T}\texttt{>} \\ u \ C\texttt{<}\overline{{}^{s}T}\texttt{>} \rhd {}^{s}T = {}^{s}T' \\ \texttt{lost} \notin {}^{s}T' \\ \mathrm{sdyn}\big({}^{s}T, h, \iota, {}^{\circ}\iota \ C\texttt{<}\overline{{}^{r}T}\texttt{>}, \emptyset\big) = {}^{r}T\end{array}\right\} \implies \mathrm{dyn}({}^{s}T', h, {}^{r}\Gamma, \emptyset) = {}^{r}T$$

The proof can be found in Sec. A.2 on page 140 and runs by an induction on the shape of ${}^{s}T$.

The lemma is used in Lemma A.1.11 to show that the runtime type used to check the upper bounds is equal before and after viewpoint adaptation.

A.1.4.3 Evaluation Preserves Runtime Types

The evaluation of an expression preserves the types of the objects in the heap.

Lemma A.1.27 (Evaluation Preserves Runtime Types)

$$ {}^{r}\Gamma \vdash h, e \rightsquigarrow h', \_ \quad \Longrightarrow \quad (\forall \iota \in \mathrm{dom}(h)\,.\, h(\iota)\downarrow_1 = h'(\iota)\downarrow_1) $$

The proof is a simple inspection of the operational semantics and gives that the evaluation of an expression never modifies the runtime types of existing objects. □

The lemma is used when we need to adapt the heap that is used in a judgment. In particular, we have the following properties as simple consequences:

$$ \begin{aligned} &{}^{r}\Gamma \vdash h, e \rightsquigarrow h', \_ \Longrightarrow \\ &\quad (\forall \iota \in \mathrm{dom}(h)\,.\, \mathrm{owners}(h, \iota) = \mathrm{owners}(h', \iota)) \ \wedge \\ &\quad (\mathrm{dyn}({}^{s}T, h, {}^{r}\Gamma, \overline{o_\iota}) = {}^{r}T \Longrightarrow \mathrm{dyn}({}^{s}T, h', {}^{r}\Gamma, \overline{o_\iota}) = {}^{r}T) \ \wedge \\ &\quad (h,\ \iota \vdash {}^{r}T \ \ \mathsf{strictly\ OK} \Longrightarrow h',\ \iota \vdash {}^{r}T \ \ \mathsf{strictly\ OK}) \end{aligned} $$

That is, evaluation preserves the owners of objects, the result of dynamization, and the well-formedness of runtime types. These functions only depend on the available type information and are therefore not influenced by the evaluation of an expression.

A.1.4.4 dyn is Compositional

The dynamization of a non-variable type can be modeled by composing the dynamization of the type arguments.

Lemma A.1.28 (dyn is Compositional)

$$ \begin{aligned} &\mathrm{owner}(h, {}^{r}\Gamma(\texttt{this})) = {}^{o}\iota \ \wedge \\ &\frac{u\left[{}^{o}\iota \,/\, \texttt{self}\,,\ {}^{o}\iota \,/\, \texttt{peer}\,,\ {}^{r}\Gamma(\texttt{this}) \,/\, \texttt{rep}\,,\ \texttt{any}_a \,/\, \texttt{any}\,,\ {}^{o}\iota'' \,/\, \texttt{lost}\right] = {}^{o}\iota' \ \wedge}{\mathrm{dyn}({}^{s}T_k, h, {}^{r}\Gamma, \overline{o_\iota}_k) = {}^{r}T_k} \\ \Longleftrightarrow\ & \\ &u = \texttt{lost} \Longrightarrow \overline{o_\iota} = \left\{{}^{o}\iota''\right\} \cup \overline{\overline{o_\iota}_k} \ \wedge \\ &u \neq \texttt{lost} \Longrightarrow \overline{o_\iota} = \overline{\overline{o_\iota}_k} \ \wedge \\ &\mathrm{dyn}\big(u\ C\texttt{<}\overline{{}^{s}T_k}\texttt{>}, h, {}^{r}\Gamma, \overline{o_\iota}\big) = {}^{o}\iota'\ C\texttt{<}\overline{{}^{r}T_k}\texttt{>} \end{aligned} $$

The proof is a simple investigation of Def. 2.2.14 and gives that dyn applies the same substitution to the main modifier and to the type arguments. The lemma only needs some work to align the substitutions for `lost` modifiers. □

The lemma is used in the proofs of Lemma A.1.1 (see Sec. A.2 on page 123), Lemma A.1.2 (see Sec. A.2 on page 127), Lemma A.1.26 (see Sec. A.2 on page 140), and Lemma A.1.23 (see Sec. A.2 on page 138).

A.1.4.5 Dynamization and `lost`

If a static type can be dynamized using an empty substitution for `lost`, we can conclude that `lost` is not contained in the static type.

Lemma A.1.29 (Dynamization and `lost`)

$$ \begin{aligned} \mathrm{dyn}({}^{s}T, h, {}^{r}\Gamma, \emptyset) = {}^{r}T \quad &\Longrightarrow \quad \texttt{lost} \notin {}^{s}T \\ \mathrm{sdyn}({}^{s}T, h, \iota, {}^{r}T, \emptyset) = {}^{r}T' \quad &\Longrightarrow \quad \texttt{lost} \notin {}^{s}T \end{aligned} $$

This is a simple consequence of the definitions of the dynamization functions dyn (Def. 2.2.14) and sdyn (Def. 2.2.11). □

It is used in Lemma A.1.2 (see Sec. A.2 on page 127) to deduce that static types, for which the dynamization with an empty substitution is defined, cannot contain `lost`.

A.1.5 Technicalities

A.1.5.1 Encapsulated Programs are Well formed

A program that is well formed by the encapsulation rules is also well formed by the topological rules.

Lemma A.1.30 (Encapsulated Programs are Well formed)

$$\vdash P \ \mathsf{enc} \quad \Longrightarrow \quad \vdash P \ \mathsf{OK}$$

This follows directly from the definition of encapsulated program. □

This lemma is used in the proof of the encapsulation theorem (Sec. A.2 on page 119) to show that an encapsulated program is also a topologically well-formed program.

A.1.5.2 Encapsulated Expressions are Well typed

An encapsulated expression is also a well-typed expression.

Lemma A.1.31 (Encapsulated Expressions are Well typed)

$$^s\Gamma \vdash e \ \mathsf{enc} \quad \Longrightarrow \quad ^s\Gamma \vdash e : \_$$

This follows directly from the definition of encapsulated expression. □

This lemma is used in the proof of the encapsulation theorem (Sec. A.2 on page 119) to show that the topological type rules can be applied to an encapsulated expression.

A.1.5.3 Strict Purity implies Purity

The strict definition of purity (Def. 2.4.3) fulfills the assumption about pure expressions (Assumption 2.4.2).

Lemma A.1.32 (Strict Purity implies Purity)

$$^s\Gamma \vdash e \ \mathsf{strictly\ pure} \quad \Longrightarrow \quad ^s\Gamma \vdash e \ \mathsf{pure}$$

The strict definition forbids all field updates and non-pure method calls. Therefore all fields in the prestate remain unchanged, establishing our assumption of purity. □

This lemma was proved to show that strict purity fulfills our assumption of purity. As we only work with the assumption of purity in the other lemmas and proofs, we do not actually use this lemma in a proof.

A.1.5.4 Topological Generation Lemma

The following generation lemma allows us to draw conclusions on the possible derivation of the typing. We know that some expression e has a type ^sT in an environment $^s\Gamma$. Then there is a unique shape of the expression by which we can determine which type rule has been used to derive the type ^sT. This gives us information about all the conditions that must hold for this expression.

Lemma A.1.33 (Topological Generation Lemma)

If ${}^s\Gamma \vdash e : {}^sT$ then the following hold:

1. $e = \texttt{null} \Rightarrow \exists {}^sT_0.\ \texttt{self} \notin {}^sT_0 \wedge {}^s\Gamma \vdash {}^sT_0\ \mathsf{OK} \wedge {}^s\Gamma \vdash \texttt{null} : {}^sT_0 \wedge {}^s\Gamma \vdash {}^sT_0 <: {}^sT \wedge {}^s\Gamma \vdash {}^sT\ \mathsf{OK}$
2. $e = x \Rightarrow \exists {}^sT_0.\ {}^s\Gamma(x) = {}^sT_0 \wedge {}^s\Gamma \vdash {}^sT_0 <: {}^sT \wedge {}^s\Gamma \vdash {}^sT\ \mathsf{OK}$
3. $e = \texttt{new}\ {}^sT_0\texttt{()} \Rightarrow {}^s\Gamma \vdash {}^sT_0\ \mathsf{strictly\ OK} \wedge \mathrm{om}({}^sT_0, {}^s\Gamma) \in \{\texttt{peer}, \texttt{rep}\} \wedge {}^s\Gamma \vdash {}^sT_0 <: {}^sT \wedge {}^s\Gamma \vdash {}^sT\ \mathsf{OK}$
4. $e = e_0.f \Rightarrow \exists {}^sN_0, {}^sT_0.\ {}^s\Gamma \vdash e_0 : {}^sN_0 \wedge \mathrm{FType}({}^sN_0, f) = {}^sT_0 \wedge {}^s\Gamma \vdash {}^sT_0 <: {}^sT \wedge {}^s\Gamma \vdash {}^sT\ \mathsf{OK}$
5. $e = e_0.f\ \texttt{=}\ e_1 \Rightarrow \exists {}^sN_0, {}^sT_0.\ {}^s\Gamma \vdash e_0 : {}^sN_0 \wedge \mathrm{FType}({}^sN_0, f) = {}^sT_0 \wedge {}^s\Gamma \vdash e_1 :_s {}^sT_0 \wedge {}^s\Gamma \vdash {}^sT_0 <: {}^sT \wedge {}^s\Gamma \vdash {}^sT\ \mathsf{OK}$
6. $e = e_0\ \texttt{.}\ m\texttt{<}\overline{{}^sT_l}\texttt{>(}\overline{e_q}\texttt{)} \Rightarrow \exists {}^sN_0, \overline{X_l}, \overline{{}^sN_l}, {}^sT_0, \overline{{}^sT'_q}, \overline{pid}.$
$\quad {}^s\Gamma \vdash e_0 : {}^sN_0 \wedge \mathrm{MSig}\big({}^sN_0, m, \overline{{}^sT_l}\big) = ms \wedge$
$\quad ms = \texttt{\_ <}\overline{X_l\ \texttt{extends}\ {}^sN_l}\texttt{>}\ {}^sT_0\ m\texttt{(}\overline{{}^sT'_q\ pid}\texttt{)} \wedge$
$\quad {}^s\Gamma \vdash \overline{e_q} :_s \overline{{}^sT'_q} \wedge {}^s\Gamma \vdash \overline{{}^sT_l} <:_s \overline{{}^sN_l} \wedge$
$\quad {}^s\Gamma \vdash \overline{{}^sT_l}\ \mathsf{strictly\ OK} \wedge {}^s\Gamma \vdash {}^sT_0 <: {}^sT \wedge {}^s\Gamma \vdash {}^sT\ \mathsf{OK}$
7. $e = \texttt{(}{}^sT_0\texttt{)}\ e_0 \Rightarrow {}^s\Gamma \vdash e : \texttt{\_} \wedge {}^s\Gamma \vdash {}^sT_0\ \mathsf{OK} \wedge {}^s\Gamma \vdash {}^sT_0 <: {}^sT \wedge {}^s\Gamma \vdash {}^sT\ \mathsf{OK}$

The proof of Lemma A.1.33 runs by induction on the derivation tree of the topological type rules applied in ${}^s\Gamma \vdash e : {}^sT$. There are always two type rules that could apply to an expression: the rule for the particular kind of expression and the subsumption rule. From the particular rule we get all the conditions that are checked for this kind of expression; subsumption allows one to go to an arbitrary well-formed supertype of this type. □

A.1.5.5 Encapsulation Generation Lemma

Similarly, for the encapsulation rules, we can determine from the shape of the expression what conditions hold:

Lemma A.1.34 (Encapsulation Generation Lemma)

If ${}^s\Gamma \vdash e\ \mathsf{enc}$ then the following hold:

1. $e = \texttt{null} \Rightarrow {}^s\Gamma \vdash \texttt{null} : \texttt{\_}$
2. $e = x \Rightarrow {}^s\Gamma \vdash x : \texttt{\_}$
3. $e = \texttt{new}\ {}^sT_0\texttt{()} \Rightarrow {}^s\Gamma \vdash \texttt{new}\ {}^sT_0\texttt{()} : \texttt{\_}$
4. $e = e_0.f \Rightarrow {}^s\Gamma \vdash e_0.f : \texttt{\_} \wedge {}^s\Gamma \vdash e_0\ \mathsf{enc}$
5. $e = e_0.f\ \texttt{=}\ e_1 \Rightarrow \exists {}^sN_0.\ {}^s\Gamma \vdash e_0.f = e_1 : \texttt{\_} \wedge {}^s\Gamma \vdash e_0 : {}^sN_0 \wedge {}^s\Gamma \vdash e_0\ \mathsf{enc} \wedge {}^s\Gamma \vdash e_1\ \mathsf{enc} \wedge \mathrm{om}({}^sN_0) \in \{\texttt{self}, \texttt{peer}, \texttt{rep}\}$
6. $e = e_0\ \texttt{.}\ m\texttt{<}\overline{{}^sT_l}\texttt{>(}\overline{e_q}\texttt{)} \Rightarrow \exists {}^sN_0.\ {}^s\Gamma \vdash e_0\ \texttt{.}\ m\texttt{<}\overline{{}^sT}\texttt{>(}\overline{e}\texttt{)} : \texttt{\_} \wedge {}^s\Gamma \vdash e_0 : {}^sN_0 \wedge {}^s\Gamma \vdash e_0\ \mathsf{enc} \wedge {}^s\Gamma \vdash \overline{e}\ \mathsf{enc} \wedge (\mathrm{om}({}^sN_0) \in \{\texttt{self}, \texttt{peer}, \texttt{rep}\} \vee \mathrm{MSig}\big({}^sN_0, m, \overline{{}^sT}\big) = \texttt{pure <\_> \_}\ m\texttt{(\_)})$
7. $e = \texttt{(}{}^sT_0\texttt{)}\ e_0 \Rightarrow {}^s\Gamma \vdash \texttt{(}{}^sT_0\texttt{)}\ e : \texttt{\_} \wedge {}^s\Gamma \vdash e\ \mathsf{enc}$

The proof of Lemma A.1.34 again runs by induction on the derivation tree of the encapsulation rules applied in ${}^s\Gamma \vdash e\ \mathsf{enc}$. There is one unique rule that can be applied from which we get all the conditions that are checked. □

A.1.5.6 Operational Semantics Generation Lemma

Finally, from the shape of an expression we can determine the unique rule from the operational semantics that is used.

Lemma A.1.35 (Operational Semantics Generation Lemma)

If ${}^{r}\Gamma \vdash h, e \rightsquigarrow h', v$ then the following hold:

1. $e = \texttt{null} \quad \Rightarrow \quad h' = h \ \wedge \ v = \texttt{null}_a$
2. $e = x \quad \Rightarrow \quad h' = h \ \wedge \ {}^{r}\Gamma(x) = v$
3. $e = \texttt{new } {}^{s}T_0\texttt{()} \quad \Rightarrow \quad \exists {}^{r}T, C, \overline{fv}.\ \mathrm{dyn}({}^{s}T, h, {}^{r}\Gamma, \emptyset) = {}^{r}T \wedge$
   $\mathrm{ClassOf}({}^{r}T) = C \ \wedge \ \left(\forall f \in \mathrm{fields}(C)\,.\, \overline{fv}(f) = \texttt{null}_a\right) \ \wedge$
   $h + \left({}^{r}T, \overline{fv}\right) = (h', \iota) \ \wedge \ v = \iota$
4. $e = e_0\,.f \quad \Rightarrow \quad \exists \iota_0.\ {}^{r}\Gamma \vdash h, e_0 \rightsquigarrow h', \iota_0 \ \wedge \ h'(\iota_0\,.f) = v$
5. $e = e_0\,.f \texttt{ = } e_1 \quad \Rightarrow \quad \exists h_0, \iota_0, h_1.\ {}^{r}\Gamma \vdash h, e_0 \rightsquigarrow h_0, \iota_0 \wedge$
   ${}^{r}\Gamma \vdash h_0, e_1 \rightsquigarrow h_1, v \ \wedge \ h_1[\iota_0\,.f \texttt{ = } v] = h'$
6. $e = e_0\ .\ m\texttt{<}\overline{{}^{s}T_l}\texttt{>(}\overline{e_q}\texttt{)} \quad \Rightarrow \quad \exists h_0, \iota_0, h_1, \overline{v_q}, e, \overline{X_l}, \overline{pid}, \overline{{}^{r}T_l}.$
   ${}^{r}\Gamma \vdash h, e_0 \rightsquigarrow h_0, \iota_0 \wedge$
   ${}^{r}\Gamma \vdash h_0, \overline{e_q} \rightsquigarrow h_1, \overline{v_q} \ \wedge \ \mathrm{MBody}(h_0, \iota_0, m) = e \wedge$
   $\mathrm{MSig}(h_0, \iota_0, m) = \texttt{\_ <}\overline{X_l}\texttt{ extends \_> \_ }m\texttt{(\_ }\overline{pid}\texttt{)} \wedge$
   $\mathrm{dyn}\left(\overline{{}^{s}T_l}, h, {}^{r}\Gamma, \emptyset\right) = \overline{{}^{r}T_l} \wedge$
   ${}^{r}\Gamma' = \left\{\overline{X_l \mapsto {}^{r}T_l}\ ;\ \texttt{this} \mapsto \iota_0\ ,\ \overline{pid \mapsto v_q}\right\} \ \wedge$
   ${}^{r}\Gamma' \vdash h_1, e \rightsquigarrow h', v$
7. $e = \texttt{(}{}^{s}T_0\texttt{)}\ e_0 \quad \Rightarrow \quad {}^{r}\Gamma \vdash h, e \rightsquigarrow h', v \ \wedge \ h', {}^{r}\Gamma \vdash v : {}^{s}T$

The proof of Lemma A.1.35 again runs by induction on the derivation tree of the operational semantics applied in ${}^{r}\Gamma \vdash h, e \rightsquigarrow h', v$. There is one unique rule that can be applied from which we get all the conditions that are checked. □

A.1.5.7 Deterministic Semantics

The evaluation of an expression in a particular runtime environment and heap results in a unique resulting value and heap, up to the renaming of addresses.

Lemma A.1.36 (Deterministic Semantics)

$$\left.\begin{array}{l} {}^{r}\Gamma \vdash h, e \rightsquigarrow h', v \\ {}^{r}\Gamma \vdash h, e \rightsquigarrow h'', v' \end{array}\right\} \Longrightarrow v = v' \ \wedge \ h' = h'' \qquad \text{up to renaming of addresses}$$

The proof runs by rule induction on the shape of the expression e. The only non-determinism comes from rule OS_NEW, which does not uniquely determine the address of the new object. This information from OS_NEW can be used to build a mapping between the two executions of the expressions. □

A.1.6 Properties that do not Hold

In the following we discuss some properties that do not hold, but have some intuitive appeal, and illustrate via examples why they do not hold. These are included only as reference.

A.1.6.1 `lost` in Upper Bounds of Well-formed Types

The intuition that a type, that does not contain `lost`, also does not use `lost` in the viewpoint-adapted upper bounds is wrong.

$$\left.\begin{array}{l} {}^{s}\Gamma \vdash {}^{s}N \ \ \mathsf{OK} \\ \mathrm{ClassBnds}({}^{s}N) = \overline{{}^{s}N} \\ \texttt{lost} \notin {}^{s}N \end{array}\right\} \not\Longrightarrow \texttt{lost} \notin \overline{{}^{s}N}$$

Counterexample:

```
class C< X extends peer Object > {}

any C<peer Object> x;
```

Class C uses `peer Object` as upper bound. Variable `x` uses the type `any C<peer Object>`, which is well-formed in a suitable environment. However, the viewpoint-adapted upper bound is `lost Object` and such a type must not be created. Therefore, strict well-formedness of types has to separately check for occurrences of `lost` in the type itself and also in the viewpoint-adapted upper bounds.

A.1.6.2 Subtyping Preserves Well-formedness

The supertype of a well-formed static type is not always also well formed.

$$\left.\begin{array}{l} \vdash P \ \ \mathsf{OK} \\ {}^{s}\Gamma \ \ \mathsf{OK} \\ {}^{s}\Gamma \vdash {}^{s}T \ \ \mathsf{OK} \\ {}^{s}\Gamma \vdash {}^{s}T <: \ {}^{s}T' \end{array}\right\} \not\Longrightarrow {}^{s}\Gamma \vdash {}^{s}T' \ \ \mathsf{OK}$$

Counterexample:

```
class C< X extends peer Object > {}

peer C<peer Object> <: peer C<lost Object> <: any C<lost Object>
peer C<peer Object> <: any C<peer Object>
```

Class C again uses `peer Object` as upper bound. The type `peer C<peer Object>` is well formed in a suitable environment. The type `peer C<lost Object>` is a supertype, however, it is not well-formed, because `lost Object` is not a subtype of the viewpoint-adapted upper bound, which is still `peer Object`.

However, the two types `any C<peer Object>` and `any C<lost Object>` are supertypes that are well formed; the main modifier `any` results in a `lost` modifier in the corresponding upper bound and the type argument is a subtype thereof.

We did not want to couple the subtyping rules to the well-formedness rules, because to check well-formedness we also need subtyping. Instead, we strengthened the subsumption rule in Def. 2.3.13 to ensure that the supertype is well formed.

A.1.6.3 Usage of `self`

We use the `self` ownership modifier to distinguish accesses through the current object from other accesses. However, we do not enforce that `self` is not used in the program and just ensure that a reference with the `self` main modifier references the current object. Therefore, the following lemma does not hold:

$$\left.\begin{array}{l} \vdash P \ \ \mathsf{OK} \\ {}^{s}\Gamma \ \ \mathsf{OK} \\ {}^{s}\Gamma \vdash e : \texttt{self \_<\_>} \end{array}\right\} \not\Longrightarrow e = \texttt{this}$$

Not only $e = \texttt{this}$ can have `self` as main modifier, also declared method parameters x, field access through `this`, and also method calls on `this`, can have `self` as main modifier, if the declared field type or method return type uses `self` as main modifier.

A simple example:

```
class C {
  self C me;

  self C getMe() {
    return me;
  }

  void setMe(self C newme) {
    me = newme;
  }

  void demo() {
    this.setMe( this );
    me = this.getMe();
  }
}
```

However, as argued in Sec. 2.5.1, allowing the use of `self` in these positions does not add significant expressiveness and can be forbidden in programs.

A.2 Proofs

This section presents the proofs of the theorems and lemmas from Sec. 2.3.5, Sec. 2.4.6, and App. A.1 that were not proved directly with the definition of the lemma.

Note that in the definitions of judgments with a single rule, we use the rule notation, because it is supported by Ott. However, these judgments with unique rules are read as equivalences, i.e., if the conclusion holds, we can assume the antecedents.

We expand strict subtyping (Def. 2.3.10) and strict expression typing (Def. 2.3.13) in the following proofs, i.e., use non-strict subtyping and type rules and check for `lost` explicitly.

A.2.1 Main Results

A.2.1.1 Proof of Theorem 2.3.26 — Type Safety

We prove:

$$\left.\begin{array}{ll} 1. & \vdash P \ \ \mathsf{OK} \\ 2. & h, {}^{r}\Gamma : {}^{s}\Gamma \ \ \mathsf{OK} \\ 3. & {}^{s}\Gamma \vdash e : {}^{s}T \\ 4. & {}^{r}\Gamma \vdash h, e \ \rightsquigarrow \ h', v \end{array}\right\} \Longrightarrow \left\{\begin{array}{ll} I. & h', {}^{r}\Gamma : {}^{s}\Gamma \ \ \mathsf{OK} \\ II. & h', {}^{r}\Gamma \vdash v : \ {}^{s}T \end{array}\right.$$

Where it simplifies the proofs, we split $II.$ into the following two subgoals, according to RTSTE_DEF (Def. 2.2.15):

$$\begin{array}{ll} IIa. & \exists \overline{o}.\ \exists {}^{r}T.\ \mathrm{dyn}({}^{s}T, h', {}^{r}\Gamma, \overline{o}) \ = {}^{r}T \ \ \wedge \ \ h' \vdash v : \ {}^{r}T \\ IIb. & {}^{s}T = \texttt{self \_<\_>} \Longrightarrow {}^{r}\Gamma(\texttt{this}) = v \end{array}$$

We prove this by rule induction on the operational semantics.

Case 1: $e = \texttt{null}$

We have the assumptions of the theorem:

$$\begin{array}{ll} 1. \ \vdash P \ \ \mathsf{OK} & 2. \ \ h, {}^{r}\Gamma : {}^{s}\Gamma \ \ \mathsf{OK} \\ 3. \ {}^{s}\Gamma \vdash \texttt{null} : {}^{s}T & 4. \ \ {}^{r}\Gamma \vdash h, \texttt{null} \ \rightsquigarrow \ h', v \end{array}$$

From 3., the type rules, and the Topological Generation Lemma A.1.33 we get that there exists an ${}^sT_0$ such that:

$$\begin{array}{lll} {}^s\Gamma \vdash \texttt{null} : {}^sT_0 & & \texttt{self} \notin {}^sT_0 \\ {}^s\Gamma \vdash {}^sT_0 \ \mathsf{OK} & {}^s\Gamma \vdash {}^sT_0 <: {}^sT & {}^s\Gamma \vdash {}^sT \ \mathsf{OK} \end{array}$$

From 4., the operational semantics, and the Operational Semantics Generation Lemma A.1.35 we know:

$${}^r\Gamma \vdash h, \texttt{null} \rightsquigarrow h, \texttt{null}_a$$

Therefore, we have that:

$$h' = h \qquad v = \texttt{null}_a$$

- *Part I:* $h', {}^r\Gamma : {}^s\Gamma \ \ \mathit{OK}$

  Follows directly from 2. and $h = h'$.

- *Part IIa:* $\exists \overline{o_l}.\ \exists {}^rT.\ \mathrm{dyn}({}^sT, h', {}^r\Gamma, \overline{o_l}) = {}^rT \ \wedge \ h' \vdash \texttt{null}_a : {}^rT$

  We have 1., *I.*, and ${}^s\Gamma \vdash {}^sT \ \mathsf{OK}$ and can apply Lemma A.1.9 to arrive at $\exists \overline{o_l}.\ \exists {}^rT.\ \mathrm{dyn}({}^sT, h', {}^r\Gamma, \overline{o_l}) = {}^rT$.

  We know from the operational semantics that $v = \texttt{null}_a$ and can assign an arbitrary runtime type to $\texttt{null}_a$ by rule RTT_NULL from Def. 2.2.13.

- *Part IIb:* ${}^sT = \texttt{self \_<\_>} \implies {}^r\Gamma(\texttt{this}) = \texttt{null}_a$

  We know from the type rules that $\texttt{self} \notin {}^sT_0$. Therefore, also a supertype ${}^sT$ cannot have `self` as main modifier according to Lemma A.1.21.

Case 2: $e = x$

We have the assumptions of the theorem:

$$\begin{array}{lll} 1. \ \vdash P \ \mathsf{OK} & & 2. \ h, {}^r\Gamma : {}^s\Gamma \ \mathsf{OK} \\ 3. \ {}^s\Gamma \vdash x : {}^sT & & 4. \ {}^r\Gamma \vdash h, x \rightsquigarrow h', v \end{array}$$

From 3., the type rules, and the Topological Generation Lemma A.1.33 we get that there exists an ${}^sT_0$ such that:

$${}^s\Gamma(x) = {}^sT_0 \qquad {}^s\Gamma \vdash x : {}^sT_0 \qquad {}^s\Gamma \vdash {}^sT_0 <: {}^sT \qquad {}^s\Gamma \vdash {}^sT \ \mathsf{OK}$$

From 4., the operational semantics, and the Operational Semantics Generation Lemma A.1.35 we know:

$${}^r\Gamma(x) = v \qquad {}^r\Gamma \vdash h, x \rightsquigarrow h, v$$

Therefore, we have that:

$$h' = h$$

- *Part I:* $h', {}^r\Gamma : {}^s\Gamma \ \ \mathit{OK}$

  Follows directly from 2. and $h = h'$.

- *Part II:* $h', {}^{r}\Gamma \vdash v : {}^{s}T$

  We want to apply Lemma A.1.22 to arrive at II. From 1. we know that the program is well formed. From Part I. we know that the static and runtime environments conform. We know from the operational semantics that the value v is from the runtime environment. Therefore we know from the definition of well-formed runtime environment (Def. 2.3.25), that $h', {}^{r}\Gamma \vdash v : {}^{s}T_0$ holds, where ${}^{s}T_0$ is ${}^{s}\Gamma(x)$, the static type of the variable. We know that ${}^{s}T_0$ is well formed from 2. and that ${}^{s}T$ is well formed from 3. We also know that ${}^{s}\Gamma \vdash {}^{s}T_0 <: {}^{s}T$ and can therefore apply Lemma A.1.22 to arrive at $h', {}^{r}\Gamma \vdash v : {}^{s}T$.

Case 3: $e = \texttt{new}\ {}^{s}T_0\texttt{()}$

We have the assumptions of the theorem:

$$\begin{array}{ll} 1.\ \vdash P\ \ \mathsf{OK} & 2.\ \ h, {}^{r}\Gamma : {}^{s}\Gamma\ \ \mathsf{OK} \\ 3.\ \ {}^{s}\Gamma \vdash \texttt{new}\ {}^{s}T_0\texttt{()} : {}^{s}T & 4.\ \ {}^{r}\Gamma \vdash h, \texttt{new}\ {}^{s}T_0\texttt{()} \rightsquigarrow h', v \end{array}$$

From 3., the type rules, and the Topological Generation Lemma A.1.33 we get that there exists an ${}^{s}T_0$ such that:

$$\begin{array}{lll} {}^{s}\Gamma \vdash {}^{s}T_0\ \ \mathsf{strictly\ OK} & \mathrm{om}({}^{s}T_0, {}^{s}\Gamma) \in \{\texttt{peer}, \texttt{rep}\} & {}^{s}\Gamma \vdash \texttt{new}\ {}^{s}T_0\texttt{()} : {}^{s}T_0 \\ {}^{s}\Gamma \vdash {}^{s}T_0 <: {}^{s}T & {}^{s}\Gamma \vdash {}^{s}T\ \ \mathsf{OK} & \end{array}$$

From 4., the operational semantics, and the Operational Semantics Generation Lemma A.1.35 we know that there exist ${}^{r}T_0$, C, and $\overline{fv}$ such that:

$$\begin{array}{ll} \mathrm{dyn}({}^{s}T_0, h, {}^{r}\Gamma, \emptyset) = {}^{r}T_0 & \mathrm{ClassOf}({}^{r}T_0) = C \\ \forall f \in \mathrm{fields}(C)\,.\ \overline{fv}(f) = \texttt{null}_a & h + \left({}^{r}T_0, \overline{fv}\right) = (h', \iota) \\ v = \iota & \end{array}$$

- *Part I:* $h', {}^{r}\Gamma : {}^{s}\Gamma\ \ \mathsf{OK}$

  From the operational semantics we know that we only added a new object to the heap and leave all other runtime types unchanged. Therefore all the judgments concerning ${}^{r}\Gamma$ and ${}^{s}\Gamma$ from 2. still hold. What we do have to show is that the new heap is well formed, i.e., $h'\ \ \mathsf{OK}$.

  From the well-formed heap (Def. 2.3.24) and address (Def. 2.3.23) judgments we derive that we only need to consider the new address ι and that all other addresses in the heap are still well formed. This means that we have to show three things:

  - Ia. $h',\ \iota \vdash h'(\iota)\!\downarrow_1\ \ \mathsf{strictly\ OK}$
  - Ib. $\texttt{root}_a \in \mathrm{owners}(h', \iota)$
  - Ic. $\forall f \in \mathrm{fields}(C)\,.\ \exists\, {}^{s}T'.\ (\mathrm{FType}(h', \iota, f) = {}^{s}T' \ \wedge\ h', \iota \vdash h'(\iota\,.f) : {}^{s}T')$

  Note that class C is the class of the runtime type of $h'(\iota)$.

  The proofs are:

  Ia. We know $\mathrm{dyn}({}^{s}T_0, h, {}^{r}\Gamma, \emptyset) = {}^{r}T_0$ and that $h'(\iota)\!\downarrow_1 = {}^{r}T_0$ from the operational semantics (Def. 2.2.17) and the definition of + (Def. 2.2.6).
  We know from the type rules that ${}^{s}\Gamma \vdash {}^{s}T_0\ \ \mathsf{strictly\ OK}$. We can therefore apply Lemma A.1.10 with 1. and 2. to arrive at $h,\ \iota \vdash {}^{r}T_0\ \ \mathsf{strictly\ OK}$.

We can use h' instead of h in $h,\ \iota \vdash {}^{r}T_0$ strictly OK because of Lemma A.1.27 (simply put, adding the additional object does not change the well-formedness judgment).

Note that ι is not in the domain of h. Under the conditions in which we use it, the runtime well-formedness judgment is correctly applied.

Ib. We have $\mathrm{om}({}^{s}T_0, {}^{s}\Gamma) \in \{\texttt{peer}, \texttt{rep}\}$. If the type ${}^{s}T_0$ is a type variable, dyn replaces it by the runtime type from the environment or the heap. From 2. we know that this runtime type is well formed and therefore establishes *Ib*. If ${}^{s}T_0$ is a non-variable type, then from the definition of dyn we know that the $\mathrm{owner}(h', \iota) = {}^{o}\iota$ where ${}^{o}\iota$ is either the address of ${}^{r}\Gamma(\texttt{this})$ or the owner of ${}^{r}\Gamma(\texttt{this})$. Either ${}^{o}\iota$ is already $\texttt{root}_a$ and we have *Ib*. Otherwise there is some ι' and ${}^{o}\iota = \iota'$ for which we have $\texttt{root}_a \in \mathrm{owners}(h, \iota')$ from the well-formed heap h OK. From $\mathrm{owner}(h', \iota) = \iota'$ and $\texttt{root}_a \in \mathrm{owners}(h, \iota')$ follows $\texttt{root}_a \in \mathrm{owners}(h', \iota)$.

Ic. Recall that class C is the class of the runtime type ${}^{r}T_0$. Function fields(C) is defined exactly to yield all fields that are defined for the class of the created object. Therefore FType is always defined. The field type ${}^{s}T'$ is a declared field type and from 1. we therefore know that it is well-formed. Therefore, the dyn in the definition of Def. 2.2.16 is defined by Lemma A.1.9. Finally, all fields are initially $\texttt{null}_a$, so the second part holds trivially.

Note that the check $\texttt{lost} \notin {}^{s}T_0$ as part of ${}^{s}\Gamma \vdash {}^{s}T_0$ strictly OK could be made less strict by using arbitrary owners in the operational semantics. Also, the main modifier `any` could be allowed and an arbitrary owner could be chosen. But this would not give us any significant flexibility and we do not want to introduce this non-deterministic behavior.

- *Part IIa:* $\exists \overline{{}^{o}\iota}.\ \exists {}^{r}T.\ \mathrm{dyn}({}^{s}T, h', {}^{r}\Gamma, \overline{{}^{o}\iota}) = {}^{r}T \ \wedge\ h' \vdash \iota : {}^{r}T$

  This directly follows from Lemma A.1.23 with 1., *I.*, ${}^{s}\Gamma \vdash {}^{s}T_0$, ${}^{s}T$ OK, and ${}^{s}\Gamma \vdash {}^{s}T_0 <: {}^{s}T$. Note that by Lemma A.1.15 from ${}^{s}\Gamma \vdash {}^{s}T_0$ strictly OK from the type rules we get ${}^{s}\Gamma \vdash {}^{s}T_0$ OK. The lemma directly gives us the first conjunct. The second conjunct follows from $h'(\iota)\!\downarrow_1 = {}^{r}T_0 \ \wedge\ h \vdash {}^{r}T_0 <: {}^{r}T$ (Def. 2.2.13) and is then also directly from the lemma.

- *Part IIb:* ${}^{s}T = \texttt{self\ \_<\_>} \implies {}^{r}\Gamma(\texttt{this}) = \iota$

  The type rules forbid that the static type in a new expression contains `self`, by enforcing strict well-formedness of the static type ${}^{s}\Gamma \vdash {}^{s}T_0$ strictly OK. Therefore, also a supertype ${}^{s}T$ cannot have `self` as main modifier according to Lemma A.1.21.

Case 4: $e = e_0.f$

We have the assumptions of the theorem:

1. $\vdash P$ OK	2. $h, {}^{r}\Gamma : {}^{s}\Gamma$ OK
3. ${}^{s}\Gamma \vdash e_0.f : {}^{s}T$	4. ${}^{r}\Gamma \vdash h, e_0.f \rightsquigarrow h', v$

From 3., the type rules, and the Topological Generation Lemma A.1.33 we get that there exist ${}^{s}N_0$ and ${}^{s}T_1$ such that:

${}^{s}\Gamma \vdash e_0 : {}^{s}N_0$	$\mathrm{FType}({}^{s}N_0, f) = {}^{s}T_1$
${}^{s}\Gamma \vdash e_0.f : {}^{s}T_1$	${}^{s}\Gamma \vdash {}^{s}T_1 <: {}^{s}T$

Using Corollary A.1.6 and Lemma A.1.16 we deduce:

$$^{s}\Gamma \vdash {}^{s}T_1 \ \ \mathsf{OK}$$

From 4., the operational semantics, and the Operational Semantics Generation Lemma A.1.35 we know that there exists an ι_0 such that:

$$^{r}\Gamma \vdash h, e_0 \ \rightsquigarrow \ h', \iota_0 \qquad h'(\iota_0.f) = v$$

We apply the induction hypothesis to e_0:

$$\left.\begin{array}{ll} 1_0. & \vdash P \ \ \mathsf{OK} \\ 2_0. & h, {}^{r}\Gamma : {}^{s}\Gamma \ \ \mathsf{OK} \\ 3_0. & {}^{s}\Gamma \vdash e_0 : {}^{s}N_0 \\ 4_0. & {}^{r}\Gamma \vdash h, e_0 \ \rightsquigarrow \ h', \iota_0 \end{array}\right\} \Longrightarrow \left\{\begin{array}{ll} I_0. & h', {}^{r}\Gamma : {}^{s}\Gamma \ \ \mathsf{OK} \\ II_0. & h', {}^{r}\Gamma \vdash \iota_0 : \ {}^{s}N_0 \end{array}\right.$$

$1_0.$ is identical to 1. and $2_0.$ is identical to 2. $3_0.$ is from the type rules and $4_0.$ is from the operational semantics.

- *Part I:* $h', {}^{r}\Gamma : {}^{s}\Gamma \ \ \mathsf{OK}$

  Identical to I_0.

- *Part II:* $h', {}^{r}\Gamma \vdash v : \ {}^{s}T$

  We first want to apply Lemma 2.3.28 to derive $h', {}^{r}\Gamma \vdash v : \ {}^{s}T_1$. For this, we use the following instantiation of the lemma:

$$\left.\begin{array}{ll} 1_1. & h', {}^{r}\Gamma \vdash \iota_0 : \ {}^{s}N_0 \\ 2_1. & h', {}^{r}\Gamma' \vdash v : \ {}^{s}T_2 \\ 3_1. & \mathrm{dyn}(\emptyset, h', {}^{r}\Gamma, \emptyset) = \emptyset \\ 4_1. & {}^{r}\Gamma' = \{\emptyset \ ; \ \mathtt{this} \mapsto \iota_0, \_\} \\ 5_1. & \mathrm{ClassDom}(\mathrm{ClassOf}({}^{s}N_0)) = \overline{X} \\ 6_1. & \mathrm{free}({}^{s}T_2) \subseteq \overline{X}, \emptyset \\ 7_1. & {}^{s}\Gamma \vdash {}^{s}N_0 \ \ \mathsf{OK} \end{array}\right\} \Longrightarrow \left\{\begin{array}{lll} I_1. & ({}^{s}N_0 \rhd {}^{s}T_2)\,[\emptyset/\emptyset] & = \ {}^{s}T_1 \\ II_1. & h', {}^{r}\Gamma \vdash v : \ {}^{s}T_1 & \end{array}\right.$$

  Note that ${}^{s}T_1$ is the viewpoint-adapted field type from the type rules.

  We deduce the requirements as follows:

  - $1_1.$: we know from $II_0.$ that $h', {}^{r}\Gamma \vdash \iota_0 : \ {}^{s}N_0$.
  - From the well-formed heap judgment from Part I above, we can deduce that $\mathrm{FType}(h', \iota_0, f) = {}^{s}T_2$ and $h', \iota_0 \vdash v : \ {}^{s}T_2$, that is, that the declared field type ${}^{s}T_2$ can be assigned to the field value from the viewpoint of the target object. With the construction of ${}^{r}\Gamma'$ below, this gives us 2_1. As the field name is defined in a unique class and from the type rules, we know that ${}^{s}N_0 \rhd {}^{s}T_2 \ = \ {}^{s}T_1$.
  - In field types we do not have method type variables and therefore $3_1.$ is not needed and in $4_1.$ we construct the simple runtime environment ${}^{r}\Gamma' = \{\emptyset \ ; \ \mathtt{this} \mapsto \iota_0, \_\}$.
  - $5_1.$ and $6_1.$: we have 1. from which we can conclude that ${}^{s}\Gamma' \vdash {}^{s}T_2 \ \ \mathsf{OK}$, where ${}^{s}\Gamma'$ is the static environment in which the field type was checked for well formedness. From this follows that $\mathrm{free}({}^{s}T_2) \subseteq \overline{X}$ where $\mathrm{ClassDom}(\mathrm{ClassOf}({}^{s}N_0)) = \overline{X}$, because the static type rules use subsumption to find the type ${}^{s}N_0$ in which the field is declared.

– 7_1.: from 3_0. we know that expression e_0 is typable as ${}^sN_0$. Using Lemma A.1.16 with 1., 2., and 3_0. we can deduce that ${}^sN_0$ is well formed.

We now have all the requirements to apply Lemma 2.3.28 to arrive at the desired $h', {}^r\Gamma \vdash v : {}^sT_1$.

Finally, we have that ${}^s\Gamma \vdash {}^sT_1 <: {}^sT$ and can apply Lemma A.1.22 to arrive at the conclusion.

Case 5: $e = e_0.f = e_1$

We have the assumptions of the theorem:

$$\begin{array}{llll} 1. & \vdash P\ \mathsf{OK} & 2. & h, {}^r\Gamma : {}^s\Gamma\ \mathsf{OK} \\ 3. & {}^s\Gamma \vdash e_0.f = e_1 : {}^sT & 4. & {}^r\Gamma \vdash h, e_0.f = e_1 \rightsquigarrow h', v \end{array}$$

From 3., the type rules, and the Topological Generation Lemma A.1.33 we get that there exist ${}^sN_0$ and ${}^sT_1$ such that:

$$\begin{array}{lll} {}^s\Gamma \vdash e_0 : {}^sN_0 & \mathrm{FType}({}^sN_0, f) = {}^sT_1 & \mathtt{lost} \notin {}^sT_1 \\ {}^s\Gamma \vdash e_1 : {}^sT_1 & {}^s\Gamma \vdash e_0.f = e_1 : {}^sT_1 & {}^s\Gamma \vdash {}^sT_1 <: {}^sT \end{array}$$

Using Corollary A.1.6 and Lemma A.1.16 we deduce:

$${}^s\Gamma \vdash {}^sT_1\ \mathsf{OK}$$

From 4., the operational semantics, and the Operational Semantics Generation Lemma A.1.35 we know that there exist h_0, ι_0, and h_1 such that:

$${}^r\Gamma \vdash h, e_0 \rightsquigarrow h_0, \iota_0 \qquad {}^r\Gamma \vdash h_0, e_1 \rightsquigarrow h_1, v \qquad h_1[\iota_0.f = v] = h'$$

We apply the induction hypothesis to e_0:

$$\left.\begin{array}{ll} 1_0. & \vdash P\ \mathsf{OK} \\ 2_0. & h, {}^r\Gamma : {}^s\Gamma\ \mathsf{OK} \\ 3_0. & {}^s\Gamma \vdash e_0 : {}^sN_0 \\ 4_0. & {}^r\Gamma \vdash h, e_0 \rightsquigarrow h_0, \iota_0 \end{array}\right\} \Longrightarrow \left\{\begin{array}{ll} I_0. & h_0, {}^r\Gamma : {}^s\Gamma\ \mathsf{OK} \\ II_0. & h_0, {}^r\Gamma \vdash \iota_0 : {}^sN_0 \end{array}\right.$$

1_0. is identical to 1. and 2_0. is identical to 2. 3_0. is from the type rules and 4_0. is from the operational semantics.

We apply the induction hypothesis to e_1:

$$\left.\begin{array}{ll} 1_1. & \vdash P\ \mathsf{OK} \\ 2_1. & h_0, {}^r\Gamma : {}^s\Gamma\ \mathsf{OK} \\ 3_1. & {}^s\Gamma \vdash e_1 : {}^sT_1 \\ 4_1. & {}^r\Gamma \vdash h_0, e_1 \rightsquigarrow h_1, v \end{array}\right\} \Longrightarrow \left\{\begin{array}{ll} I_1. & h_1, {}^r\Gamma : {}^s\Gamma\ \mathsf{OK} \\ II_1. & h_1, {}^r\Gamma \vdash v : {}^sT_1 \end{array}\right.$$

1_1. is identical to 1. and 2_1. is identical to I_0. 3_1. is from the type rules and 4_1. is from the operational semantics.

- *Part I:* $h', {}^{r}\Gamma : {}^{s}\Gamma \ \ \mathsf{OK}$

  We have that $h_1[\iota_0.f = v] = h'$. We have $h_1, {}^{r}\Gamma : {}^{s}\Gamma \ \ \mathsf{OK}$ from I_1.

  From the definition of $h_1[\iota_0.f = v] = h'$ we know that the types in h' stay unchanged and therefore all the judgments concerning ${}^{r}\Gamma$ and ${}^{s}\Gamma$ still hold. What we do have to show is that the new heap is well formed, i.e., $h' \ \ \mathsf{OK}$.

  From the well-formed heap (Def. 2.3.24) and address (Def. 2.3.23) judgments, knowing that $\mathrm{dom}(h_1) = \mathrm{dom}(h')$ and that the runtime types in the heap stayed unchanged, the only thing that needs to be shown is: $\mathrm{FType}(h', \iota_0, f) = {}^{s}T_2$ and $h', \iota_0 \vdash h'(\iota_0.f) : {}^{s}T_2$, that is, that the declared type of the field can be assigned to the new value that is stored in the field.

  We want to apply Lemma 2.3.29 to arrive at $h', \iota_0 \vdash h'(\iota_0.f) : {}^{s}T_2$. We use the following instantiation of the lemma:

$$\left.\begin{array}{ll} 1_2. & h', {}^{r}\Gamma \vdash \iota_0 : {}^{s}N_0 \\ 2_2. & ({}^{s}N_0 \rhd {}^{s}T_2)\,[\emptyset/\emptyset] \ = \ {}^{s}T_1 \\ 3_2. & \texttt{lost} \notin {}^{s}T_1 \\ 4_2. & h', {}^{r}\Gamma \vdash v : {}^{s}T_1 \\ 5_2. & \mathrm{dyn}(\emptyset, h', {}^{r}\Gamma, \emptyset) = \emptyset \\ 6_2. & {}^{r}\Gamma' = \{\emptyset \ ; \ \texttt{this} \mapsto \iota, \_\} \\ 7_2. & \mathrm{ClassDom}(\mathrm{ClassOf}({}^{s}N_0)) = \overline{X} \\ 8_2. & \mathrm{free}({}^{s}T_2) \subseteq \overline{X}\,, \emptyset \\ 9_2. & {}^{s}\Gamma \vdash {}^{s}N_0 \ \ \mathsf{OK} \end{array}\right\} \Longrightarrow \ I_2. \quad h', {}^{r}\Gamma' \vdash v : {}^{s}T_2$$

  We deduce the requirements as follows:

  - $1_2.$ and $2_2.$: from the type rules we know $\mathrm{FType}({}^{s}N_0, f) = {}^{s}T_1$ and from $II_0.$ we have $h_0, {}^{r}\Gamma \vdash \iota_0 : {}^{s}N_0$. Therefore $\mathrm{FType}(h', \iota_0, f) = {}^{s}T_2$ is defined and ${}^{s}N_0 \rhd {}^{s}T_2 = {}^{s}T_1$ holds. In a field type we do not have method type variables and therefore we use empty substitutions.
  - $3_2.$: from the type rules we know $\texttt{lost} \notin {}^{s}T_1$.
  - $4_2.$: we know that $h'(\iota_0.f) = v$ and from $II_1.$ that $h_1, {}^{r}\Gamma \vdash v : {}^{s}T_1$. We can use h' instead of h_1 by Lemma A.1.27, because changing a field value does not change the validity of the type assignment.
  - $5_2.$ and $6_2.$: in a field type we do not have method type variables and therefore construct the simple runtime environment ${}^{r}\Gamma' = \{\emptyset \ ; \ \texttt{this} \mapsto \iota_0, \_\}$.
  - $7_2.$ and $8_2.$: we have 1. from which we can conclude that the field type was checked for well-formedness, i.e., ${}^{s}\Gamma' \vdash {}^{s}T_2 \ \ \mathsf{OK}$ was checked for a corresponding static environment ${}^{s}\Gamma'$. From this follows that $\mathrm{free}({}^{s}T_2) \subseteq \overline{X}$ where $\mathrm{ClassDom}(\mathrm{ClassOf}({}^{s}N_0)) = \overline{X}$.
  - $9_2.$: from $3_0.$ we know that expression e_0 is typable as ${}^{s}N_0$. Using Lemma A.1.16 with 1., 2., and $3_0.$ we can deduce that ${}^{s}N_0$ is well formed.

  We now have all the requirements to apply Lemma 2.3.29 to arrive at $h', {}^{r}\Gamma' \vdash v : {}^{s}T_2$, which is equivalent to $h', \iota_0 \vdash h'(\iota_0.f) : {}^{s}T_2$. Therefore the final heap h' is well formed.

- *Part II:* $h', {}^{r}\Gamma \vdash v : {}^{s}T$

  From $II_1.$ we have $h_1, {}^{r}\Gamma \vdash v : {}^{s}T_1$ and from Lemma A.1.27 and the definition of $h_1[\iota_0.f = v] = h'$ we get that $h', {}^{r}\Gamma \vdash v : {}^{s}T_1$.

Finally, we have that ${}^s\Gamma \vdash {}^sT_1 <: {}^sT$ and can apply Lemma A.1.22 to arrive at the conclusion.

Case 6: $e = e_0\ .\ m\texttt{<}\overline{{}^sT_l}\texttt{>(}\ e_1\ \texttt{)}$

For simplicity, we assume there is only one method argument. The extension to multiple arguments is standard, but tedious.

We have the assumptions of the theorem:

1. $\vdash P\ \ \mathsf{OK}$
2. $h, {}^r\Gamma : {}^s\Gamma\ \ \mathsf{OK}$
3. ${}^s\Gamma \vdash e_0\ .\ m\texttt{<}\overline{{}^sT_l}\texttt{>(}\ e_1\ \texttt{)} : {}^sT$
4. ${}^r\Gamma \vdash h, e_0\ .\ m\texttt{<}\overline{{}^sT_l}\texttt{>(}\ e_1\ \texttt{)} \rightsquigarrow h', v$

From 3., the type rules, and the Topological Generation Lemma A.1.33 we get that there exist ${}^sN_0$, $\overline{X_l}$, $\overline{{}^sN_{l0}}$, ${}^sT_{10}$, ${}^sT_{20}$, and pid such that:

$$
\begin{array}{ll}
{}^s\Gamma \vdash e_0 : {}^sN_0 & {}^s\Gamma \vdash \overline{{}^sT_l}\ \ \mathsf{strictly\ OK} \\
\mathrm{MSig}\left({}^sN_0, m, \overline{{}^sT_l}\right) = \texttt{\_ <}\overline{X_l\ \texttt{extends}\ {}^sN_{l0}}\texttt{>}\ {}^sT_{10}\ m\texttt{(}{}^sT_{20}\ pid\texttt{)} & \\
{}^s\Gamma \vdash e_1 : {}^sT_{20} & {}^s\Gamma \vdash \overline{{}^sT_l} <: \overline{{}^sN_{l0}} \\
\texttt{lost} \notin \overline{{}^sN_{l0}}\ ,\ {}^sT_{20} & {}^s\Gamma \vdash e_0\ .\ m\texttt{<}\overline{{}^sT_l}\texttt{>(}\ e_1\ \texttt{)} : {}^sT_{10} \\
{}^s\Gamma \vdash {}^sT_{10} <: {}^sT & {}^s\Gamma \vdash {}^sT\ \ \mathsf{OK}
\end{array}
$$

Using Corollary A.1.7 and Lemma A.1.16 we deduce:

$${}^s\Gamma \vdash \overline{{}^sN_{l0}}\ ,\ {}^sT_{20}\ ,\ {}^sT_{10}\ \ \mathsf{OK}$$

From 4., the operational semantics, and the Operational Semantics Generation Lemma A.1.35 we know that there exist h_0, ι_0, h_1, v_1, $\overline{{}^rT_l}$, and e_2 such that:

$$
\begin{array}{ll}
{}^r\Gamma \vdash h, e_0 \rightsquigarrow h_0, \iota_0 & {}^r\Gamma \vdash h_0, e_1 \rightsquigarrow h_1, v_1 \\
\mathrm{MSig}(h_0, \iota_0, m) = \texttt{\_ <}\overline{X_l\ \texttt{extends \_}}\texttt{>\ \_}\ m\texttt{(\_}\ pid\texttt{)} & \\
\mathrm{dyn}\left(\overline{{}^sT_l}, h, {}^r\Gamma, \emptyset\right) = \overline{{}^rT_l} & \\
{}^r\Gamma' = \left\{\overline{X_l \mapsto {}^rT_l}\ ;\ \texttt{this} \mapsto \iota_0\ ,\ pid \mapsto v_1\right\} & \\
\mathrm{MBody}(h_0, \iota_0, m) = e_2 & {}^r\Gamma' \vdash h_1, e_2 \rightsquigarrow h', v
\end{array}
$$

For a method call, we have to distinguish three different classes for the receiver type:

- Statically, we know that the receiver has type ${}^sN_0$ with class C_0 and we have the method signature $\mathrm{MSig}\left({}^sN_0, m, \overline{{}^sT_l}\right) = \texttt{\_ <}\overline{X_l\ \texttt{extends}\ {}^sN_{l0}}\texttt{>}\ {}^sT_{10}\ m\texttt{(}{}^sT_{20}\ pid\texttt{)}$ after viewpoint adaptation from the viewpoint ${}^sN_0$ and substitution of the method type arguments $\overline{{}^sT_l}$. The un-adapted method signature is
  $\mathrm{MSig}(C_0, m) = ms$ where $ms = \texttt{\_ <}\overline{X_l\ \texttt{extends}\ {}^sN_l}\texttt{>}\ {}^sT_1\ m\texttt{(}{}^sT_2\ pid\texttt{)}$.

- At runtime, the receiver object ι_0 has a runtime type with some class C_0'' and we have the signature
  $\mathrm{MSig}(h_0, \iota_0, m) = \texttt{\_ <}\overline{X_l\ \texttt{extends \_}}\texttt{>\ \_}\ m\texttt{(\_}\ pid\texttt{)}$ and the method body
  $\mathrm{MBody}(h_0, \iota_0, m) = e_2$.

  Note how the runtime method signature look-up (Def. 2.2.8) determines an arbitrary class that declares method m and returns that signature. The operational semantics only uses the identifiers of the method type variables and the parameters to build the runtime environment; the concrete types in the signature are irrelevant.

  In contrast, the runtime method body look-up returns the most concrete implementation of method m.

- The third class to consider is the class C_0' which contains the most concrete implementation of the method. Here we have the signature
  $\mathrm{MSig}(C_0', m) = ms'$ where $ms' = \texttt{\_ <}\overline{X_l \texttt{ extends } {}^sN_l'}\texttt{>} \ {}^sT_1' \ m\texttt{(}{}^sT_2' \ pid\texttt{)}$ and the method body
  $\mathrm{MBody}(C_0', m) = e_2$. This class is important, because we take the method body from this class and execute it. Note that the method body for class C_0'' is the same as the one for C_0'.

From the definitions of the look-up functions we know that the three classes are in the following subclass relationship:

$$C_0''\texttt{<\_>} \sqsubseteq C_0'\texttt{<\_>} \qquad\qquad C_0'\texttt{<\_>} \sqsubseteq C_0\texttt{<}\overline{{}^sT}\texttt{>}$$

for corresponding types $\overline{{}^sT}$.

From 1. we know that the program is well formed and therefore we know that the well-formedness of the methods was checked and, in particular, that the overriding rule Def. 2.3.17 was checked. Therefore the number and name of the method type variables and method parameters of method m are the same on all three levels and the types on the different levels only differ by substitution of class type arguments for class type variables. In particular, we have that $ms[\overline{{}^sT}/\overline{X}] = ms'$, where the $\overline{{}^sT}$ are the type arguments in the subclassing relationship between C_0' and C_0 derived above and $\overline{X}$ are the type variables of class C_0.

We apply the induction hypothesis to e_0:

$$\left.\begin{array}{ll} 1_0. & \vdash P \ \ \mathsf{OK} \\ 2_0. & h, {}^r\Gamma : {}^s\Gamma \ \ \mathsf{OK} \\ 3_0. & {}^s\Gamma \vdash e_0 : {}^sN_0 \\ 4_0. & {}^r\Gamma \vdash h, e_0 \ \rightsquigarrow \ h_0, \iota_0 \end{array}\right\} \Longrightarrow \left\{\begin{array}{ll} I_0. & h_0, {}^r\Gamma : {}^s\Gamma \ \ \mathsf{OK} \\ II_0. & h_0, {}^r\Gamma \vdash \iota_0 : \ {}^sN_0 \end{array}\right.$$

1_0. is identical to 1. and 2_0. is identical to 2. 3_0. is from the type rules and 4_0. is from the operational semantics.

We apply the induction hypothesis to e_1:

$$\left.\begin{array}{ll} 1_1. & \vdash P \ \ \mathsf{OK} \\ 2_1. & h_0, {}^r\Gamma : {}^s\Gamma \ \ \mathsf{OK} \\ 3_1. & {}^s\Gamma \vdash e_1 : {}^sT_{20} \\ 4_1. & {}^r\Gamma \vdash h_0, e_1 \ \rightsquigarrow \ h_1, v_1 \end{array}\right\} \Longrightarrow \left\{\begin{array}{ll} I_1. & h_1, {}^r\Gamma : {}^s\Gamma \ \ \mathsf{OK} \\ II_1. & h_1, {}^r\Gamma \vdash v_1 : \ {}^sT_{20} \end{array}\right.$$

1_1. is identical to 1. and 2_1. is identical to I_0. 3_1. is from the type rules and 4_1. is from the operational semantics.

We apply the induction hypothesis to e_2:

$$\left.\begin{array}{ll} 1_2. & \vdash P \ \ \mathsf{OK} \\ 2_2. & h_1, {}^r\Gamma' : {}^s\Gamma' \ \ \mathsf{OK} \\ 3_2. & {}^s\Gamma' \vdash e_2 : {}^sT_1' \\ 4_2. & {}^r\Gamma' \vdash h_1, e_2 \ \rightsquigarrow \ h', v \end{array}\right\} \Longrightarrow \left\{\begin{array}{ll} I_2. & h', {}^r\Gamma' : {}^s\Gamma' \ \ \mathsf{OK} \\ II_2. & h', {}^r\Gamma' \vdash v : \ {}^sT_1' \end{array}\right.$$

1_2. is identical to 1. and 4_2. is from the operational semantics.
Requirements 2_2. and 3_2. need a detailed development.

- **Requirement** 2_2.**:** From I_1. we know that $h_1 \ \ \mathsf{OK}$.

  In the operational semantics we construct the runtime environment ${}^r\Gamma'$ as follows:

  $$\mathrm{dyn}\Big(\overline{{}^sT_l}, h, {}^r\Gamma, \emptyset\Big) = \overline{{}^rT_l}$$
  $${}^r\Gamma' = \Big\{\overline{X_l \mapsto {}^rT_l} \ ; \ \texttt{this} \mapsto \iota_0 \, , \ pid \mapsto v_1\Big\}$$

From $\vdash P\ \ \mathsf{OK}$ we know that class C_0', the class from which we get the method body expression e_2, was type checked using an environment that contains the type variables from C_0' and m and maps the parameters to their declared types; that is, we have:
${}^s\Gamma' = \left\{\overline{X_l \mapsto {}^sN_l'}\,,\ \overline{X_k' \mapsto {}^sN_k'}\ ;\ \texttt{this} \mapsto \texttt{self}\ C_0'\texttt{<}\overline{X_k'}\texttt{>}\,,\ pid \mapsto {}^sT_2'\right\}$,
where the $\overline{X_k'}$ are the class type variables of C_0' and the $\overline{{}^sN_l'}$, $\overline{{}^sN_k'}$ are the corresponding upper bounds of the method and class type variables.

We have ${}^s\Gamma'\ \ \mathsf{OK}$ from $\vdash P\ \ \mathsf{OK}$, because from Def. 2.3.14 and Def. 2.3.16 we know that the corresponding conditions were checked.

What remains to be shown for $h_1, {}^r\Gamma' : {}^s\Gamma'\ \ \mathsf{OK}$, according to Def. 2.3.25, is:

1. ${}^r\Gamma'(\texttt{this})$ can be typed with ${}^s\Gamma'(\texttt{this})$,
2. the argument value can be typed with the declared parameter type,
3. the method type arguments are runtime subtypes of the dynamization of their declared upper bounds, and
4. the method type arguments are well-formed runtime types.

The proofs are:

1. ${}^r\Gamma'(\texttt{this})$ can be typed with ${}^s\Gamma'(\texttt{this})$, that is, $h_1, {}^r\Gamma' \vdash \iota_0 :\ \texttt{self}\ C_0'\texttt{<}\overline{X_k'}\texttt{>}$:
   We know that ι_0 has class C_0'' which is a subclass of C_0'. A simple investigation of Def. 2.2.15 shows that ${}^r\Gamma'$ correctly models the `self` modifier and the dynamization goes exactly to a supertype of the runtime type of ι_0 to determine all type variables of C_0'.
2. The argument value can be typed with the declared parameter type, that is, from the point of view of the receiver, we can assign the declared parameter type of class C_0' to the argument value: $h_1, {}^r\Gamma' \vdash v_1 :\ {}^sT_2'$.
   From II_1. we have $h_1, {}^r\Gamma \vdash v_1 :\ {}^sT_{20}$, that is, from the point of view of the caller, we can assign the viewpoint-adapted parameter type to the argument value. We want to apply Lemma 2.3.29 to arrive at $h_1, {}^r\Gamma' \vdash v_1 :\ {}^sT_2$, that is, that from the point of view of the receiver, we can assign the declared parameter type of class C_0 to the argument value. We use the following instantiation of the lemma:

$$
\begin{array}{ll}
 & 1_3.\quad h_1, {}^r\Gamma \vdash \iota_0 :\ {}^sN_0 \\
 & 2_3.\quad ({}^sN_0 \rhd {}^sT_2)\left[\overline{{}^sT_l}/\overline{X_l}\right] \ =\ {}^sT_{20} \\
 & 3_3.\quad \texttt{lost} \notin {}^sT_{20} \\
 & 4_3.\quad h_1, {}^r\Gamma \vdash v_1 :\ {}^sT_{20} \\
 & 5_3.\quad \mathrm{dyn}\left(\overline{{}^sT_l}, h_1, {}^r\Gamma, \emptyset\right) \ = \overline{{}^rT_l} \\
 & 6_3.\quad {}^r\Gamma' = \left\{\overline{X_l \mapsto {}^rT_l}\ ;\ \texttt{this} \mapsto \iota_0\,,\ pid \mapsto v_1\right\} \\
 & 7_3.\quad \mathrm{ClassDom}(\mathrm{ClassOf}({}^sN_0)) \ = \overline{X} \\
 & 8_3.\quad \mathrm{free}({}^sT_2) \subseteq \overline{X}\,,\ \overline{X_l} \\
 & 9_3.\quad {}^s\Gamma \vdash {}^sN_0\ \ \mathsf{OK} \\
\Longrightarrow & \\
 & I_3.\quad h_1, {}^r\Gamma' \vdash v_1 :\ {}^sT_2
\end{array}
$$

   We deduce the requirements as follows:
   - 1_3.: from II_0. we have $h_0, {}^r\Gamma \vdash \iota_0 :\ {}^sN_0$, by Lemma A.1.27 the judgment also holds for h_1.

- 2_3.: from the type rules we know that the method signature look-up was defined, therefore, this viewpoint adaptation is defined.
- 3_3.: from the type rules we know $\texttt{lost} \notin {}^sT_{20}$.
- 4_3.: identical to II_1.
- 5_3. and 6_3.: this corresponds to the construction of the runtime environment in the operational semantics.
- 7_3. and 8_3.: we have 1., from which we can conclude that the declared parameter type was checked for well formedness, i.e., ${}^s\Gamma'' \vdash {}^sT_2 \ \ \mathsf{OK}$ was checked for a corresponding ${}^s\Gamma''$.
- 9_3.: from 3_0. we know that expression e_0 is typable as ${}^sN_0$. Using Lemma A.1.16 with 1., 2., and 3_0. we can deduce that ${}^sN_0$ is well formed.

We now have all the requirements to apply Lemma 2.3.29 to arrive at $h_1, {}^r\Gamma' \vdash v_1 : {}^sT_2$.

Finally, Lemma A.1.24 ensures that the type assignment also holds for ${}^sT_2'$, the declared parameter type in the subclass C_0'.

3. The method type arguments are runtime subtypes of the dynamization of their declared upper bounds, that is, for $\mathrm{dyn}\left(\overline{{}^sN_l'}, h_1, {}^r\Gamma', \emptyset\right) = \overline{{}^rT_l'}$ we have $h_1 \vdash \overline{{}^rT_l} <: \overline{{}^rT_l'}$.

   This is shown by Lemma A.1.12. For each ${}^sN$ from $\overline{{}^sN_l'}$ and the corresponding ${}^sT$ from $\overline{{}^sT_l}$ we have:

$$
\begin{array}{rl}
 & 1_4.\ \ \vdash P \ \ \mathsf{OK} \\
 & 2_4.\ \ h_1, {}^r\Gamma : {}^s\Gamma \ \ \mathsf{OK} \\
 & 3_4.\ \ ({}^sN_0 \rhd {}^sN)\left[\overline{{}^sT}/\overline{X}\right] \ = \ {}^sT' \\
 & 4_4.\ \ \texttt{lost} \notin {}^sT' \\
 & 5_4.\ \ {}^s\Gamma \vdash {}^sT\,,\, {}^sT' \ \ \mathsf{OK} \\
 & 6_4.\ \ {}^s\Gamma \vdash {}^sT <: {}^sT' \\
 & 7_4.\ \ h_1, {}^r\Gamma \vdash \iota_0 : {}^sN_0 \\
 & 8_4.\ \ h_1, {}^r\Gamma \vdash {}^sN_0, {}^sN; \left(\overline{{}^sT}/\overline{X}, \iota_0\right) = {}^r\Gamma' \\
\Longrightarrow & \\
\exists \overline{o_l}, {}^rT.\ \ I_4. & \ \ \mathrm{dyn}({}^sT, h_1, {}^r\Gamma, \overline{o_l}) = {}^rT \wedge \\
\exists {}^rT'.\ \ II_4. & \ \ \mathrm{dyn}({}^sN, h_1, {}^r\Gamma', \emptyset) = {}^rT' \wedge \\
III_4. & \ \ h \vdash {}^rT <: {}^rT'
\end{array}
$$

   We deduce the requirements as follows:

   - 1_4.: identical to 1.
   - 2_4.: identical to I_1.
   - 3_4.: from the type rules we know that the method signature look-up was defined, therefore, this viewpoint adaptation is defined.
   - 4_4.: from the type rules we know that the viewpoint-adapted upper bounds do not contain `lost`.
   - 5_4.: from the type rules we know that the method type arguments are strictly well formed. Therefore, by Lemma A.1.15, they are also well formed. The viewpoint-adapted upper bounds are well-formed by Lemma A.1.7.
   - 6_4.: from the type rules we know that the method type arguments are subtypes of the viewpoint-adapted upper bounds.
   - 7_4.: from II_0. and Lemma A.1.27.
   - 8_4.: the well-formedness conditions on ${}^sN_0$ and ${}^sN$ are from the type rules. The

construction of the new runtime environment ${}^{r}\Gamma'$ corresponds to the operational semantics.

In conclusion, this lemma gives us that the type arguments $\overline{{}^{r}T_l}$, which were converted to runtime types in the callers context, are subtypes of the upper bounds $\overline{{}^{r}T'_l}$ from the receivers point of view.

4. The method type arguments are well-formed runtime types:
that is, $h_1,\ \iota_0 \vdash \overline{{}^{r}T_l}$ strictly OK holds.

We apply Lemma A.1.10 to derive this. We have the strict well-formedness of the method type arguments from the type rules and use the well-formedness of the program, 1., and the well-formed environments, I_1. Well-formedness holds for an arbitrary address, so also for the particular viewpoint of the receiver.

- **Requirement** 3_2**.:** The program is well formed and therefore the method m in class C'_0 was checked for well formedness in the environment ${}^{s}\Gamma'$ defined above. In particular, the method body expression e_2 is type checked against the declared return type of the method, ${}^{s}T'_1$, that is, ${}^{s}\Gamma' \vdash e_2 : {}^{s}T'_1$ is checked.

We now have all requirements to apply the induction hypothesis to e_2 and can derive I_2. and II_2.

Finally, we can show the two parts of the proof of this case:

- *Part I:* $h', {}^{r}\Gamma : {}^{s}\Gamma$ OK

  From I_2. we get the well formedness of the new heap h'. The well formedness of the environments ${}^{s}\Gamma$ and ${}^{r}\Gamma$ stays unchanged from I_1.

- *Part II:* $h', {}^{r}\Gamma \vdash v :\ {}^{s}T$

  We first want to apply Lemma 2.3.28 to derive $h', {}^{r}\Gamma \vdash v :\ {}^{s}T_{10}$, that is, that from the point of view of the caller, the viewpoint-adapted return type can be assigned to the value. For this, we use the following instantiation of the lemma:

$$
\begin{array}{rl}
 & 1_5.\quad h', {}^{r}\Gamma \vdash \iota_0 :\ {}^{s}N_0 \\
 & 2_5.\quad h', {}^{r}\Gamma' \vdash v :\ {}^{s}T_1 \\
 & 3_5.\quad \mathrm{dyn}\Big(\overline{{}^{s}T_l}, h', {}^{r}\Gamma, \emptyset\Big) \ = \overline{{}^{r}T_l} \\
 & 4_5.\quad {}^{r}\Gamma' = \Big\{\overline{X_l \mapsto {}^{r}T_l}\ ;\ \mathtt{this} \mapsto \iota_0\ ,\ pid \mapsto v_1\Big\} \\
 & 5_5.\quad \mathrm{ClassDom}(\mathrm{ClassOf}({}^{s}N_0)) \ = \overline{X} \\
 & 6_5.\quad \mathrm{free}({}^{s}T_1) \subseteq \overline{X}\ ,\ \overline{X_l} \\
 & 7_5.\quad {}^{s}\Gamma \vdash {}^{s}N_0 \ \ \mathsf{OK} \\
\Longrightarrow & \\
 & I_5.\quad ({}^{s}N_0 \rhd {}^{s}T_1)\left[\overline{{}^{s}T_l}/\overline{X_l}\right] \ = \ {}^{s}T_{10} \\
 & II_5.\quad h', {}^{r}\Gamma \vdash v :\ {}^{s}T_{10}
\end{array}
$$

  Note that ${}^{s}T_{10}$ is the viewpoint-adapted method return type from the type rules.

  We deduce the requirements as follows:

  – 1_5.: we know from II_0. that $h_0, {}^{r}\Gamma \vdash \iota_0 :\ {}^{s}N_0$. By Lemma A.1.27 this also holds for h'.

- 2_5.: from II_2. we have $h', {}^r\Gamma' \vdash v : {}^sT_1'$. ${}^sT_1'$ is the return type of the overriding method in class C_0', a subclass of C_0, the static class of the receiver. From the overriding rules we know that the return type in the subclass is the same as the one in superclasses, up to substitutions of class type arguments. We can apply Lemma A.1.24 to arrive at $h', {}^r\Gamma' \vdash v : {}^sT_1$, that is, from the point of view of the receiver, we can assigned the declared return type of the method to the return value.
- 3_5. and 4_5. are the construction of the new runtime environment ${}^r\Gamma'$ used in the operational semantics. We can use heap h' by Lemma A.1.27.
- 5_5. and 6_5.: we have 1. from which we can conclude that ${}^s\Gamma' \vdash {}^sT_1$ OK, where ${}^s\Gamma'$ is the static environment in which the declared method return type was checked for well formedness. From this follows that
  $\text{free}({}^sT_1) \subseteq \overline{X}, \overline{X_l}$,
  where $\text{ClassDom}(\text{ClassOf}({}^sN_0)) = \overline{X}$, because the static type rules use subsumption to find the type ${}^sN_0$ in which the method is declared.
- 7_5.: from 3_0. we know that expression e_0 is typable as ${}^sN_0$. Using Lemma A.1.16 with 1., 2., and 3_0. we can deduce that ${}^sN_0$ is well formed.

We now have all the requirements to apply Lemma 2.3.28 to arrive at the desired $h', {}^r\Gamma \vdash v : {}^sT_{10}$.

Finally, we have that ${}^s\Gamma \vdash {}^sT_{10} <: {}^sT$ and can apply Lemma A.1.22 to arrive at the conclusion.

Case 7: $e = ({}^sT_0)\ e_0$

We have the assumptions of the theorem:

$$\begin{array}{ll} 1.\ \vdash P\ \mathsf{OK} & 2.\ h, {}^r\Gamma : {}^s\Gamma\ \mathsf{OK} \\ 3.\ {}^s\Gamma \vdash ({}^sT_0)\ e_0 : {}^sT & 4.\ {}^r\Gamma \vdash h, ({}^sT_0)\ e_0 \rightsquigarrow h', v \end{array}$$

From 3., the type rules, and the Topological Generation Lemma A.1.33 we get that there exists an ${}^sT_1$ such that:

$$\begin{array}{lll} {}^s\Gamma \vdash e_0 : {}^sT_1 & {}^s\Gamma \vdash {}^sT_0\ \mathsf{OK} & {}^s\Gamma \vdash ({}^sT_0)\ e : {}^sT_0 \\ {}^s\Gamma \vdash {}^sT_0 <: {}^sT & {}^s\Gamma \vdash {}^sT\ \mathsf{OK} & \end{array}$$

From 4., the operational semantics, and the Operational Semantics Generation Lemma A.1.35 we know:

$${}^r\Gamma \vdash h, e_0 \rightsquigarrow h', v \qquad h', {}^r\Gamma \vdash v : {}^sT_0$$

We apply the induction hypothesis to e_0:

$$\left.\begin{array}{ll} 1_0. & \vdash P\ \mathsf{OK} \\ 2_0. & h, {}^r\Gamma : {}^s\Gamma\ \mathsf{OK} \\ 3_0. & {}^s\Gamma \vdash e_0 : {}^sT_1 \\ 4_0. & {}^r\Gamma \vdash h, e_0 \rightsquigarrow h', v \end{array}\right\} \Longrightarrow \left\{\begin{array}{ll} I_0. & h', {}^r\Gamma : {}^s\Gamma\ \mathsf{OK} \\ II_0. & h', {}^r\Gamma \vdash v : {}^sT_1 \end{array}\right.$$

1_0. is identical to 1. and 2_0. is identical to 2. 3_0. is from the type rules and 4_0. is from the operational semantics.

- *Part I:* $h', {}^r\Gamma : {}^s\Gamma\ \mathit{OK}$

  Identical to I_0.

- *Part II:* $h', {}^{r}\Gamma \vdash v : {}^{s}T$

  From the operational semantics we have $h', {}^{r}\Gamma \vdash v : {}^{s}T_0$. We have the subtype relationship ${}^{s}\Gamma \vdash {}^{s}T_0 <: {}^{s}T$ and can apply Lemma A.1.22 to arrive at the conclusion.

□

A.2.1.2 Proof of Theorem 2.4.7 — Encapsulation

We prove:

$$\left.\begin{array}{ll} 1. & \vdash P \ \ \mathsf{enc} \\ 2. & h, {}^{r}\Gamma : {}^{s}\Gamma \ \ \mathsf{OK} \\ 3. & {}^{s}\Gamma \vdash e \ \ \mathsf{enc} \\ 4. & {}^{r}\Gamma \vdash h, e \rightsquigarrow h', \_ \end{array}\right\} \Longrightarrow \begin{array}{l} \forall \iota \in \mathrm{dom}(h) \,.\ \forall f \in \mathrm{dom}(h(\iota)\downarrow_2) \,. \\ \quad I.\ h(\iota.f) = h'(\iota.f) \ \vee \\ \quad II.\ \mathrm{owner}(h, {}^{r}\Gamma(\texttt{this})) \in \mathrm{owners}(h, \iota) \end{array}$$

We prove this by rule induction on the operational semantics.

Case 1: $e = \texttt{null}$

We have the assumptions of the theorem:

$$\begin{array}{llll} 1. & \vdash P \ \ \mathsf{enc} & 2. & h, {}^{r}\Gamma : {}^{s}\Gamma \ \ \mathsf{OK} \\ 3. & {}^{s}\Gamma \vdash \texttt{null} \ \ \mathsf{enc} & 4. & {}^{r}\Gamma \vdash h, \texttt{null} \rightsquigarrow h', \_ \end{array}$$

From 3., the encapsulated expression rules, and the Encapsulation Generation Lemma A.1.34 we get:

$${}^{s}\Gamma \vdash \texttt{null} : \_$$

From 4., the operational semantics, and the Operational Semantics Generation Lemma A.1.35 we know:

$${}^{r}\Gamma \vdash h, \texttt{null} \rightsquigarrow h, \texttt{null}_a$$

Therefore, we have that:

$$h' = h \qquad\qquad v = \texttt{null}_a$$

The heap is unchanged and $I.$ holds.

Case 2: $e = x$

We have the assumptions of the theorem:

$$\begin{array}{llll} 1. & \vdash P \ \ \mathsf{enc} & 2. & h, {}^{r}\Gamma : {}^{s}\Gamma \ \ \mathsf{OK} \\ 3. & {}^{s}\Gamma \vdash x \ \ \mathsf{enc} & 4. & {}^{r}\Gamma \vdash h, x \rightsquigarrow h', \_ \end{array}$$

From 3., the encapsulated expression rules, and the Encapsulation Generation Lemma A.1.34 we get:

$${}^{s}\Gamma \vdash x : \_$$

From 4., the operational semantics, and the Operational Semantics Generation Lemma A.1.35 we know:

$${}^{r}\Gamma(x) = v \qquad\qquad {}^{r}\Gamma \vdash h, x \rightsquigarrow h, v$$

Therefore, we have that:

$$h' = h$$

The heap is unchanged and $I.$ holds.

Case 3: $e = \texttt{new}\ {}^{s}T_0\texttt{()}$

We have the assumptions of the theorem:

$$\begin{array}{ll} 1.\ \vdash P\ \ \mathsf{enc} & 2.\ \ h, {}^{r}\Gamma : {}^{s}\Gamma\ \ \mathsf{OK} \\ 3.\ \ {}^{s}\Gamma \vdash \texttt{new}\ {}^{s}T_0\texttt{()}\ \ \mathsf{enc} & 4.\ \ {}^{r}\Gamma \vdash h, \texttt{new}\ {}^{s}T_0\texttt{()}\ \rightsquigarrow\ h', \_ \end{array}$$

From 3., the encapsulated expression rules, and the Encapsulation Generation Lemma A.1.34 we get:

$${}^{s}\Gamma \vdash \texttt{new}\ {}^{s}T_0\texttt{()} : \_$$

From 4., the operational semantics, and the Operational Semantics Generation Lemma A.1.35 we know that there exist ${}^{r}T$, C, and $\overline{fv}$ such that:

$$\begin{array}{ll} \mathrm{dyn}({}^{s}T_0, h, {}^{r}\Gamma, \emptyset) = {}^{r}T & \mathrm{ClassOf}({}^{r}T) = C \\ \forall f \in \mathrm{fields}(C)\,.\ \overline{fv}(f) = \texttt{null}_a & h + \left({}^{r}T, \overline{fv}\right) = (h', \iota) \end{array}$$

From the definition of $h + \left({}^{r}T, \overline{fv}\right) = (h', \iota)$ we see that h remains unchanged except for the addition of the new object at ι. Therefore $I.$ holds.

Case 4: $e = e_0\,.f$

We have the assumptions of the theorem:

$$\begin{array}{ll} 1.\ \vdash P\ \ \mathsf{enc} & 2.\ \ h, {}^{r}\Gamma : {}^{s}\Gamma\ \ \mathsf{OK} \\ 3.\ \ {}^{s}\Gamma \vdash e_0\,.f\ \ \mathsf{enc} & 4.\ \ {}^{r}\Gamma \vdash h, e_0\,.f\ \rightsquigarrow\ h', \_ \end{array}$$

From 3., the encapsulated expression rules, and the Encapsulation Generation Lemma A.1.34 we get:

$${}^{s}\Gamma \vdash e_0\,.f : \_ \qquad {}^{s}\Gamma \vdash e_0\ \ \mathsf{enc}$$

From 4., the operational semantics, and the Operational Semantics Generation Lemma A.1.35 we know that there exists an ι_0 such that:

$${}^{r}\Gamma \vdash h, e_0\ \rightsquigarrow\ h', \iota_0 \qquad h'(\iota_0\,.f) = v$$

We apply the induction hypothesis to e_0:

$$\left.\begin{array}{l} 1_0.\ \vdash P\ \ \mathsf{enc} \\ 2_0.\ \ h, {}^{r}\Gamma : {}^{s}\Gamma\ \ \mathsf{OK} \\ 3_0.\ \ {}^{s}\Gamma \vdash e_0\ \ \mathsf{enc} \\ 4_0.\ \ {}^{r}\Gamma \vdash h, e_0\ \rightsquigarrow\ h', \_ \end{array}\right\} \Longrightarrow \begin{array}{l} \forall \iota \in \mathrm{dom}(h)\,.\ \ \forall f \in \mathrm{dom}(h(\iota)\downarrow_2)\,. \\ \quad I_0.\ h(\iota\,.f) = h'(\iota\,.f)\ \vee \\ \quad II_0.\ \mathrm{owner}(h, {}^{r}\Gamma(\texttt{this})) \in \mathrm{owners}(h, \iota) \end{array}$$

$1_0.$ is identical to 1. and $2_0.$ is identical to 2. $3_0.$ is from the encapsulated expression rules and $4_0.$ is from the operational semantics.

$I_0.$ and $II_0.$ are identical to our goals $I.$ and $II.$

Case 5: $e = e_0\,.f\ \texttt{=}\ e_1$

We have the assumptions of the theorem:

$$\begin{array}{ll} 1.\ \vdash P\ \ \mathsf{enc} & 2.\ \ h, {}^{r}\Gamma : {}^{s}\Gamma\ \ \mathsf{OK} \\ 3.\ \ {}^{s}\Gamma \vdash e_0\,.f\ \texttt{=}\ e_1\ \ \mathsf{enc} & 4.\ \ {}^{r}\Gamma \vdash h, e_0\,.f\ \texttt{=}\ e_1\ \rightsquigarrow\ h', \_ \end{array}$$

From 3., the encapsulated expression rules, and the Encapsulation Generation Lemma A.1.34 we get that there exists a ${}^sN_0$ such that:

$$\begin{array}{lll} {}^s\Gamma \vdash e_0\,\texttt{.}\,f\ \texttt{=}\ e_1 : \_ & {}^s\Gamma \vdash e_0 : {}^sN_0 & {}^s\Gamma \vdash e_0\ \ \texttt{enc} \\ {}^s\Gamma \vdash e_1\ \ \texttt{enc} & \mathrm{om}({}^sN_0) \in \{\texttt{self}\,,\,\texttt{peer}\,,\,\texttt{rep}\} & \end{array}$$

From 4., the operational semantics, and the Operational Semantics Generation Lemma A.1.35 we know that there exist h_0, ι_0, and h_1 such that:

$$ {}^r\Gamma \vdash h, e_0 \ \rightsquigarrow\ h_0, \iota_0 \qquad {}^r\Gamma \vdash h_0, e_1 \ \rightsquigarrow\ h_1, v \qquad h_1[\iota_0\,\texttt{.}\,f\ \texttt{=}\ v] \ = h'$$

We apply the induction hypothesis to e_0:

$$\left.\begin{array}{ll} 1_0. & \vdash P\ \ \texttt{enc} \\ 2_0. & h, {}^r\Gamma : {}^s\Gamma\ \ \mathsf{OK} \\ 3_0. & {}^s\Gamma \vdash e_0\ \ \texttt{enc} \\ 4_0. & {}^r\Gamma \vdash h, e_0\ \rightsquigarrow\ h_0, \_ \end{array}\right\} \Longrightarrow \begin{array}{l} \forall \iota \in \mathrm{dom}(h)\,.\ \ \forall f \in \mathrm{dom}(h(\iota)\downarrow_2)\,. \\ \qquad I_0.\ h(\iota\,.f) = h_0(\iota\,.f)\ \vee \\ \qquad II_0.\ \mathrm{owner}(h, {}^r\Gamma(\texttt{this})) \in \mathrm{owners}(h, \iota) \end{array}$$

1_0. is identical to 1. and 2_0. is identical to 2. 3_0. is from the encapsulated expression rules and 4_0. is from the operational semantics.

We apply the induction hypothesis to e_1:

$$\left.\begin{array}{ll} 1_1. & \vdash P\ \ \texttt{enc} \\ 2_1. & h_0, {}^r\Gamma : {}^s\Gamma\ \ \mathsf{OK} \\ 3_1. & {}^s\Gamma \vdash e_1\ \ \texttt{enc} \\ 4_1. & {}^r\Gamma \vdash h_0, e_1\ \rightsquigarrow\ h_1, \_ \end{array}\right\} \Longrightarrow \begin{array}{l} \forall \iota \in \mathrm{dom}(h_0)\,.\ \ \forall f \in \mathrm{dom}(h(\iota)\downarrow_2)\,. \\ \qquad I_1.\ h_0(\iota\,.f) = h_1(\iota\,.f)\ \vee \\ \qquad II_1.\ \mathrm{owner}(h_0, {}^r\Gamma(\texttt{this})) \in \mathrm{owners}(h_0, \iota) \end{array}$$

1_1. is identical to 1., 3_1. is from the encapsulated expression rules, and 4_1. is from the operational semantics. 2_1. corresponds to 2. and applying Theorem 2.3.26.

The only difference between the final heap h' and heap h_1 is the update of field f of object ι_0. This might change the value and therefore I. might not hold and we have to show that II. holds.

We have ${}^s\Gamma \vdash e_0 : {}^sN_0$ from the encapsulated expression rules and ${}^r\Gamma \vdash h, e_0 \ \rightsquigarrow\ h_0, \iota_0$ from the operational semantics. We can apply Theorem 2.3.26 to arrive at $h_0, {}^r\Gamma \vdash \iota_0 : \ {}^sN_0$.

We can now apply Lemma A.1.25 and use $\mathrm{om}({}^sN_0) \in \{\texttt{self}\,,\,\texttt{peer}\,,\,\texttt{rep}\}$ from the encapsulated expression rules to deduce that the owner of ι_0 is either ${}^r\Gamma(\texttt{this})$ or the owner of ${}^r\Gamma(\texttt{this})$, ensuring II.

Case 6: $e = e_0\ \texttt{.}\ m\texttt{<}\overline{{}^sT}\texttt{>(}\ e_1\ \texttt{)}$

For simplicity, we again assume there is only one method argument. The extension to multiple arguments is standard.

We have the assumptions of the theorem:

$$\begin{array}{lll} 1. & \vdash P\ \ \texttt{enc} & 2. \quad h, {}^r\Gamma : {}^s\Gamma\ \ \mathsf{OK} \\ 3. & {}^s\Gamma \vdash e_0\ \texttt{.}\ m\texttt{<}\overline{{}^sT}\texttt{>(}\ e_1\ \texttt{)}\ \ \texttt{enc} & \\ 4. & {}^r\Gamma \vdash h, e_0\ \texttt{.}\ m\texttt{<}\overline{{}^sT}\texttt{>(}\ e_1\ \texttt{)}\ \rightsquigarrow\ h', \_ & \end{array}$$

From 3., the encapsulated expression rules, and the Encapsulation Generation Lemma A.1.34 we get that there exists a ${}^sN_0$ such that:

$$\begin{array}{ll} {}^s\Gamma \vdash e_0\ \texttt{.}\ m\texttt{<}\overline{{}^sT}\texttt{>(}\ e_1\ \texttt{)} : \_ & {}^s\Gamma \vdash e_0 : {}^sN_0 \\ {}^s\Gamma \vdash e_0\ \ \texttt{enc} & {}^s\Gamma \vdash e_1\ \ \texttt{enc} \\ \mathrm{om}({}^sN_0) \in \{\texttt{self}\,,\,\texttt{peer}\,,\,\texttt{rep}\}\ \ \vee\ \ \mathrm{MSig}\Big({}^sN_0, m, \overline{{}^sT}\Big) = \texttt{pure <\_> \_ }m\texttt{(\_)} & \end{array}$$

From 4., the operational semantics, and the Operational Semantics Generation Lemma A.1.35 we know that there exist h_0, ι_0, h_1, v_1, $\overline{X_l}$, pid, $\overline{{}^rT_l}$, and e_2 such that:

$$
\begin{array}{l}
{}^r\Gamma \vdash h, e_0 \;\rightsquigarrow\; h_0, \iota_0 \qquad\qquad {}^r\Gamma \vdash h_0, e_1 \;\rightsquigarrow\; h_1, v_1 \\
\mathrm{MSig}(h_0, \iota_0, m) \;=\; \texttt{\_ <}\overline{X_l\ \texttt{extends \_}}\texttt{> \_ }m\texttt{(\_ }pid\texttt{)} \\
\mathrm{dyn}\left(\overline{{}^sT_l}, h, {}^r\Gamma, \emptyset\right) \;=\; \overline{{}^rT_l} \\
{}^r\Gamma' = \left\{\overline{X_l \mapsto {}^rT_l}\ ;\ \texttt{this} \mapsto \iota_0\ ,\ pid \mapsto v_1\right\} \\
\mathrm{MBody}(h_0, \iota_0, m) \;=\; e_2 \qquad {}^r\Gamma' \vdash h_1, e_2 \;\rightsquigarrow\; h', v
\end{array}
$$

We apply the induction hypothesis to e_0:

$$
\left.
\begin{array}{ll}
1_0. & \vdash P \;\; \mathsf{enc} \\
2_0. & h, {}^r\Gamma : {}^s\Gamma \;\; \mathsf{OK} \\
3_0. & {}^s\Gamma \vdash e_0 \;\; \mathsf{enc} \\
4_0. & {}^r\Gamma \vdash h, e_0 \;\rightsquigarrow\; h_0, \texttt{\_}
\end{array}
\right\} \Longrightarrow
\begin{array}{l}
\forall \iota \in \mathrm{dom}(h)\,.\;\; \forall f \in \mathrm{dom}(h(\iota)\downarrow_2)\,. \\
\qquad I_0.\ h(\iota\,.f) = h_0(\iota\,.f) \;\vee \\
\qquad II_0.\ \mathrm{owner}(h, {}^r\Gamma(\texttt{this})) \in \mathrm{owners}(h, \iota)
\end{array}
$$

1_0. is identical to 1. and 2_0. is identical to 2. 3_0. is from the encapsulated expression rules and 4_0. is from the operational semantics.

We apply the induction hypothesis to e_1:

$$
\left.
\begin{array}{ll}
1_1. & \vdash P \;\; \mathsf{enc} \\
2_1. & h_0, {}^r\Gamma : {}^s\Gamma \;\; \mathsf{OK} \\
3_1. & {}^s\Gamma \vdash e_1 \;\; \mathsf{enc} \\
4_1. & {}^r\Gamma \vdash h_0, e_1 \;\rightsquigarrow\; h_1, \texttt{\_}
\end{array}
\right\} \Longrightarrow
\begin{array}{l}
\forall \iota \in \mathrm{dom}(h_0)\,.\;\; \forall f \in \mathrm{dom}(h(\iota)\downarrow_2)\,. \\
\qquad I_1.\ h_0(\iota\,.f) = h_1(\iota\,.f) \;\vee \\
\qquad II_1.\ \mathrm{owner}(h_0, {}^r\Gamma(\texttt{this})) \in \mathrm{owners}(h_0, \iota)
\end{array}
$$

1_1. is identical to 1., 3_1. is from the encapsulated expression rules, and 4_1. is from the operational semantics. 2_1. corresponds to 2. and Theorem 2.3.26.

If the method is pure, it has to satisfy the purity judgment, which by Assumption 2.4.2 means that no fields in the pre-heap changed. Therefore, we are done with this case.

Otherwise, if the method is non-pure, we apply the induction hypothesis to e_2:

$$
\left.
\begin{array}{ll}
1_2. & \vdash P \;\; \mathsf{enc} \\
2_2. & h_1, {}^r\Gamma' : {}^s\Gamma' \;\; \mathsf{OK} \\
3_2. & {}^s\Gamma' \vdash e_2 \;\; \mathsf{enc} \\
4_2. & {}^r\Gamma' \vdash h_1, e_2 \;\rightsquigarrow\; h', \texttt{\_}
\end{array}
\right\} \Longrightarrow
\begin{array}{l}
\forall \iota \in \mathrm{dom}(h_1)\,.\;\; \forall f \in \mathrm{dom}(h(\iota)\downarrow_2)\,. \\
\qquad I_2.\ h_1(\iota\,.f) = h'(\iota\,.f) \;\vee \\
\qquad II_2.\ \mathrm{owner}\big(h_1, {}^r\Gamma'(\texttt{this})\big) \in \mathrm{owners}(h_1, \iota)
\end{array}
$$

For the development of 2_2. see the development of method calls (page 113) in the proof of soundness (Theorem 2.3.26). We use the same environments here. 1_2. is identical to 1. and 4_2. is from the operational semantics. From 1. we can conclude that method m was checked to be a well-encapsulated method. A non-pure method has to satisfy ${}^s\Gamma' \vdash e_2 \;\; \mathsf{enc}$, which is 3_2.

If I_2. holds, we are done because the heap remains unchanged.

If II_2. holds, we still need to show that
$\mathrm{owner}\big(h_1, {}^r\Gamma'(\texttt{this})\big) \in \mathrm{owners}(h_1, \iota) \Rightarrow \mathrm{owner}(h_1, {}^r\Gamma(\texttt{this})) \in \mathrm{owners}(h_1, \iota)$.

We have ${}^s\Gamma \vdash e_0 : {}^sN_0$ from the encapsulated expression rules and ${}^r\Gamma \vdash h, e_0 \;\rightsquigarrow\; h_0, \iota_0$ from the operational semantics. We apply Theorem 2.3.26 to arrive at
$h_0, {}^r\Gamma \vdash \iota_0 :\; {}^sN_0$. From the operational semantics we know that ${}^r\Gamma'(\texttt{this}) = \iota_0$.

We can now apply Lemma A.1.25 and $\mathrm{om}({}^sN_0) \in \{\texttt{self}\,, \texttt{peer}\,, \texttt{rep}\}$ from the encapsulated expression rules to deduce that the owner of ι_0 is either ${}^r\Gamma(\texttt{this})$ or the owner of ${}^r\Gamma(\texttt{this})$.

Therefore, $\mathrm{owner}(h_1, {}^r\Gamma(\texttt{this})) \in \mathrm{owners}(h_1, \iota)$ holds.

Case 7: $e = \texttt{(}{}^sT_0\texttt{)}\;\; e_0$

We have the assumptions of the theorem:

$$\begin{array}{ll} 1.\ \vdash P\ \ \mathsf{enc} & 2.\ \ h, {}^{r}\Gamma : {}^{s}\Gamma\ \ \mathsf{OK} \\ 3.\ \ {}^{s}\Gamma \vdash (\,{}^{s}T_0\,)\ e_0\ \ \mathsf{enc} & 4.\ \ {}^{r}\Gamma \vdash h, (\,{}^{s}T_0\,)\ e_0\ \rightsquigarrow\ h', \_ \end{array}$$

From 3., the encapsulated expression rules, and the Encapsulation Generation Lemma A.1.34 we get:

$${}^{s}\Gamma \vdash (\,{}^{s}T_0\,)\ e_0 : \_ \qquad {}^{s}\Gamma \vdash e_0\ \ \mathsf{enc}$$

From 4., the operational semantics, and the Operational Semantics Generation Lemma A.1.35 we know:

$${}^{r}\Gamma \vdash h, e_0\ \rightsquigarrow\ h', v \qquad h', {}^{r}\Gamma \vdash v :\ {}^{s}T_0$$

We apply the induction hypothesis to e_0:

$$\left.\begin{array}{l} 1_0.\ \vdash P\ \ \mathsf{enc} \\ 2_0.\ \ h, {}^{r}\Gamma : {}^{s}\Gamma\ \ \mathsf{OK} \\ 3_0.\ \ {}^{s}\Gamma \vdash e_0\ \ \mathsf{enc} \\ 4_0.\ \ {}^{r}\Gamma \vdash h, e_0\ \rightsquigarrow\ h', \_ \end{array}\right\} \Longrightarrow \begin{array}{l} \forall \iota \in \mathrm{dom}(h)\,.\ \ \forall f \in \mathrm{dom}(h(\iota)\downarrow_2)\,. \\ \qquad I_0.\ h(\iota\,.f) = h'(\iota\,.f)\ \vee \\ \qquad II_0.\ \mathrm{owner}(h, {}^{r}\Gamma(\texttt{this})) \in \mathrm{owners}(h, \iota) \end{array}$$

$1_0.$ is identical to 1. and $2_0.$ is identical to 2. $3_0.$ is from the encapsulated expression rules and $4_0.$ is from the operational semantics.

$I_0.$ and $II_0.$ are identical to our goals $I.$ and $II.$ □

A.2.2 Viewpoint Adaptation

A.2.2.1 Proof of Lemma A.1.1 — Adaptation from a Viewpoint Auxiliary Lemma

For the proofs of the adaptation lemmas we can assume that the type variables in ${}^{r}\Gamma$ and ${}^{r}\Gamma'$ are disjoint, that is, that the type variables that can appear in ${}^{s}N$ are different from the type variables that can appear in ${}^{s}T$. Consistent renaming of method type variables in method signature look-ups are applied to avoid capturing from happening. See Sec. A.2.7 for an example and discussion.

Also note that in the proofs of the viewpoint adaptation lemmas, we expand the definition of the viewpoint creation judgment (see Def. 2.3.27).

We prove:

$$\begin{array}{l} 1.\ \ h, {}^{r}\Gamma \vdash \iota :\ {}^{s}N \\ 2.\ \ \mathrm{dyn}\Big(\overline{{}^{s}T_l}, h, {}^{r}\Gamma, \emptyset\Big)\ = \overline{{}^{r}T_l} \\ 3.\ \ {}^{r}\Gamma' = \Big\{\overline{X_l \mapsto {}^{r}T_l}\ ;\ \texttt{this} \mapsto \iota, \_\Big\} \\ 4.\ \ \mathrm{ClassDom}(\mathrm{ClassOf}({}^{s}N))\ = \overline{X} \\ 5.\ \ \mathrm{free}({}^{s}T)\ \subseteq\ \overline{X}\,,\ \overline{X_l} \\ \Longrightarrow \\ \qquad \exists \overline{o_l}, {}^{r}T.\ \ I.\ \ \mathrm{dyn}\big({}^{s}T, h, {}^{r}\Gamma', \overline{o_l}\big)\ = {}^{r}T\ \wedge \\ \exists {}^{s}T', {}^{s}T''.\ \ II.\ \ {}^{s}N \rhd {}^{s}T\ =\ {}^{s}T'\ \wedge\ {}^{s}T'\left[\overline{{}^{s}T_l}/\overline{X_l}\right] = {}^{s}T''\ \wedge \\ \exists \overline{o_l}', {}^{r}T'.\ \ III.\ \ \mathrm{dyn}\Big({}^{s}T'', h, {}^{r}\Gamma, \overline{o_l}'\Big)\ = {}^{r}T'\ \wedge \\ \qquad\qquad IV.\ \ {}^{r}T = {}^{r}T' \end{array}$$

Note that in $II.$ we split viewpoint adaptation and substitution into two steps.

Goals $I.$, $II.$, and $III.$ are simple:

- *I.* is defined because we know from 5. that all type variables in ${}^sT$ are substituted by either $\overline{X}$ or $\overline{X_l}$, and from 1., 3., and 4. we know that these variables can be substituted. We can choose an arbitrary substitution for `lost` modifiers.
- *II.* is defined because of 4., as the only condition that needs to hold is that the domain lookup for the class of the left-hand side is defined.
- *III.* is defined because we know from 1. that ${}^sN$ can be dynamized and from 2. that the $\overline{{}^sT_l}$ can be dynamized. The substitution for `lost` modifiers depends on the previous substitution, in order to make both runtime types equal.

The proof of *IV.* runs by induction on the shape of ${}^sT$; from 5. we know that we have the following 4 cases:

Case 1: ${}^sT = X_i$ **and** $X_i \in \overline{X_l}$

${}^sN \rhd {}^sT = {}^sT'$ is therefore equal to ${}^sN \rhd X_i = X_i$, because of the assumption that the $\overline{X}$ and $\overline{X_l}$ are distinct sets of type variables and no substitution takes place.

$({}^sN \rhd {}^sT)\left[\overline{{}^sT_l}/\overline{X_l}\right] = {}^sT''$ is equal to $X_i\left[\overline{{}^sT_l}/\overline{X_l}\right] = {}^sT_i$ the i-th element of the sequence of types $\overline{{}^sT_l}$.

$\mathrm{dyn}({}^sT, h, {}^r\Gamma', \overline{\varrho}) = {}^rT$ corresponds to $\mathrm{dyn}(X_i, h, {}^r\Gamma', \overline{\varrho}) = {}^rT$. From the definition of dyn we know that ${}^r\Gamma'(X_i) = {}^rT$.

From 2. we have $\mathrm{dyn}({}^sT_i, h, {}^r\Gamma, \emptyset) = {}^rT$. So $\mathrm{dyn}\left({}^sT_i, h, {}^r\Gamma, \overline{\varrho}'\right) = {}^rT$ follows with $\overline{\varrho}' = \emptyset$.

Case 2: ${}^sT = X_j$ **and** $X_j \in \overline{X}$

${}^sN \rhd {}^sT = {}^sT'$ is therefore equal to ${}^sN \rhd X_j = {}^sT_j$, where ${}^sT_j$ is the j-th type argument of ${}^sN$.

$({}^sN \rhd {}^sT)\left[\overline{{}^sT_l}/\overline{X_l}\right] = {}^sT''$ is equal to ${}^sT_j\left[\overline{{}^sT_l}/\overline{X_l}\right] = {}^sT_j$, because of the assumption that the $\overline{X_l}$ do not appear in ${}^sN$.

From 1. we know for some ${}^o\iota''$, ${}^o\iota$, and $\overline{{}^rT}$ that $\mathrm{dyn}\left({}^sN, h, {}^r\Gamma, \overline{\varrho}''\right) = {}^o\iota\ C\texttt{<}\overline{{}^rT}\texttt{>}$ is defined.

From the definition of dyn and 1. we know that for some $\overline{\varrho}_3$ and ${}^rT_j$ we have the dynamization $\mathrm{dyn}({}^sT_j, h, {}^r\Gamma, \overline{\varrho}_3) = {}^rT_j$, where ${}^rT_j$ is the j-th type argument of the $\overline{{}^rT}$. As, if the dynamization of a non-variable type is defined, also the dynamization of the type arguments of that type is defined.

Because of 1. and the definition of dyn we know that $\mathrm{dyn}(X_l, h, {}^r\Gamma', \overline{\varrho}) = {}^rT_j$. We know that ${}^r\Gamma'(\texttt{this}) = \iota$ from 3. and can therefore use the knowledge of ${}^sN$ from 1.. Therefore, both dynamizations result in the same type.

Case 3: ${}^sT = u\ C\texttt{<>}$

We name the main modifier of ${}^sN$ as u_1, that is $\mathrm{om}({}^sN) = u_1$.

${}^sN \rhd {}^sT = u'\ C\texttt{<>}$ where $u_1 \rhd u = u'$. Because there are no type variables, we also have $({}^sN \rhd {}^sT)\left[\overline{{}^sT_l}/\overline{X_l}\right] = u'\ C\texttt{<>}$.

We make a further case analysis for the possible values of u'.

Case 3a: $u' = \texttt{peer}$

There are two combinations that result in $u' = \texttt{peer}$:

1. $(u_1 = \texttt{self} \ \wedge\ u = \texttt{peer}) \implies u_1 \rhd u = \texttt{peer}$ and
2. $(u_1 = \texttt{peer} \ \wedge\ u = \texttt{peer}) \implies u_1 \rhd u = \texttt{peer}$.

$\text{dyn}({}^sT, h, {}^r\Gamma', \overline{\iota_1}) = {}^rT$ is equal to $\text{dyn}(\texttt{peer}\ C\texttt{<>}, h, {}^r\Gamma', \overline{\iota_1}) = {}^rT$. There is no `lost` in ${}^sT$ so we can use $\overline{\iota_1} = \emptyset$. From the definition of dyn we know that `peer` is substituted by the owner of the current object, i.e., ${}^r\Gamma'(\texttt{this}) = \iota \ \wedge\ \text{owner}(h, \iota) = {}^o\iota_1$ and ${}^rT = {}^o\iota_1\ C\texttt{<>}$.

$\text{dyn}\left({}^sT'', h, {}^r\Gamma, \overline{\iota_1}'\right) = {}^rT'$ is equal to $\text{dyn}\left(\texttt{peer}\ C\texttt{<>}, h, {}^r\Gamma, \overline{\iota_1}'\right) = {}^rT'$. There is no `lost` in ${}^sT'$ so we can use $\overline{\iota_1}' = \emptyset$. From the definition of dyn we know that `peer` is substituted by the owner of the current object, i.e.,
${}^r\Gamma(\texttt{this}) = \iota_2 \ \wedge\ \text{owner}(h, \iota_2) = {}^o\iota_2$ and ${}^rT' = {}^o\iota_2\ C\texttt{<>}$.

From 1. and our knowledge of u_1 we know that we can assign `self _<_>` or `peer _<_>` to ι. Therefore, we know that the owner of ι is the owner of ι_2, that is, ${}^o\iota_1 = {}^o\iota_2$.

Case 3b: $u' = \texttt{rep}$

There are two combinations that result in $u' = \texttt{rep}$:

1. $(u_1 = \texttt{self} \ \wedge\ u = \texttt{rep}) \implies u_1 \rhd u = \texttt{rep}$ and

2. $(u_1 = \texttt{rep} \ \wedge\ u = \texttt{peer}) \implies u_1 \rhd u = \texttt{rep}$.

$\text{dyn}\left({}^sT'', h, {}^r\Gamma, \overline{\iota_1}'\right) = {}^rT'$ is equal to $\text{dyn}\left(\texttt{rep}\ C\texttt{<>}, h, {}^r\Gamma, \overline{\iota_1}'\right) = {}^rT'$. There is no `lost` in ${}^sT'$ so we can use $\overline{\iota_1}' = \emptyset$. From the definition of dyn we know that `rep` is substituted by the current object, i.e., ${}^r\Gamma(\texttt{this}) = \iota_2$ and ${}^rT' = \iota_2\ C\texttt{<>}$.

We make a further distinction of the two cases:

Case 3b1:. $\text{dyn}({}^sT, h, {}^r\Gamma', \overline{\iota_1}) = {}^rT$ is equal to $\text{dyn}(\texttt{rep}\ C\texttt{<>}, h, {}^r\Gamma', \overline{\iota_1}) = {}^rT$. There is no `lost` in ${}^sT$ so we can use $\overline{\iota_1} = \emptyset$. From the definition of dyn we know that `rep` is substituted by the current object, i.e., ${}^r\Gamma'(\texttt{this}) = \iota$ and ${}^rT = \iota\ C\texttt{<>}$.

From 1. we know that we can assign `self _<_>` to ι. Therefore, we know that the ${}^r\Gamma(\texttt{this}) = \iota$ and we have that $\iota = \iota_2$ and therefore that the types are equal.

Case 3b2:. $\text{dyn}({}^sT, h, {}^r\Gamma', \overline{\iota_1}) = {}^rT$ is equal to $\text{dyn}(\texttt{peer}\ C\texttt{<>}, h, {}^r\Gamma', \overline{\iota_1}) = {}^rT$. There is no `lost` in ${}^sT$ so we can use $\overline{\iota_1} = \emptyset$. From the definition of dyn we know that `peer` is substituted by the owner of the current object, i.e.,
${}^r\Gamma'(\texttt{this}) = \iota \ \wedge\ \text{owner}(h, \iota) = {}^o\iota_1$ and ${}^rT = {}^o\iota_1\ C\texttt{<>}$.

From 1. we know that we can assign `rep _<_>` to ι. Therefore, we know that
${}^r\Gamma(\texttt{this}) = \iota_2 \ \wedge\ \text{owner}(h, \iota) = \iota_2$ and that ${}^o\iota_1 = \iota_2$ and therefore that the types are equal.

Case 3c: $u' = \texttt{any}$

For an arbitrary u_1 we have $u = \texttt{any} \implies u_1 \rhd u = \texttt{any}$.

From the definition of dyn we see that the `any` ownership modifier is replaced by the $\texttt{any}_a$ address, regardless of environment and heap. Therefore, both dynamizations are the same.

Case 3d: $u' = \texttt{lost}$

We have that $\text{dyn}({}^sT, h, {}^r\Gamma', \overline{\iota_1}) = {}^rT$ for some $\overline{\iota_1}$ that is either empty or one element long (in the case where $u = \texttt{lost}$). We call ${}^rT = {}^o\iota\ C\texttt{<>}$.

Now for $\text{dyn}\left({}^sT'', h, {}^r\Gamma, \overline{\iota_1}'\right) = {}^rT'$ we can simply choose $\overline{\iota_1}' = {}^o\iota$ and know from the definition of dyn that `lost` will be substituted by ${}^o\iota$ and we arrive at the same types.

Case 4: ${}^sT = u\ C\texttt{<}\overline{{}^sT}\texttt{>}$

Let us first give names to the result of the application of dyn:
$\mathrm{dyn}({}^sT, h, {}^r\Gamma', \overline{\sigma_l}) = \iota''\ C\texttt{<}\overline{{}^rT}''\texttt{>}$
$\mathrm{dyn}\left({}^sT'', h, {}^r\Gamma, \overline{\sigma_l}'\right) = \iota'''\ C\texttt{<}\overline{{}^rT}'''\texttt{>}$

The same analysis as case 3. before results in $\iota'' = \iota'''$.

Now we apply the induction hypothesis to the $\overline{{}^sT}$ and use 1., 2., 3. and the fact that $\mathrm{free}({}^sT) \subseteq \mathrm{ClassDom}(\mathrm{ClassOf}({}^sN)), \overline{X}$ implies $\mathrm{free}(\overline{{}^sT}) \subseteq \mathrm{ClassDom}(\mathrm{ClassOf}(({}^sN)), \overline{X}$.

Therefore we have that $\iota'' = \iota'''$ and $\overline{{}^rT}'' = \overline{{}^rT}'''$.

From Lemma A.1.28 we then know that the runtime types are the same. □

A.2.2.2 Proof of Lemma 2.3.28 — Adaptation from a Viewpoint

We prove:

$$\left.\begin{array}{ll} 1. & h, {}^r\Gamma \vdash \iota : {}^sN \\ 2. & h, {}^r\Gamma' \vdash v : {}^sT \\ 3. & \mathrm{dyn}\left(\overline{{}^sT_l}, h, {}^r\Gamma, \emptyset\right) = \overline{{}^rT_l} \\ 4. & {}^r\Gamma' = \left\{\overline{X_l \mapsto {}^rT_l}\ ;\ \texttt{this} \mapsto \iota, \_\right\} \\ 5. & \mathrm{ClassDom}(\mathrm{ClassOf}({}^sN)) = \overline{X} \\ 6. & \mathrm{free}({}^sT) \subseteq \overline{X}, \overline{X_l} \\ 7. & {}^s\Gamma \vdash {}^sN\ \mathsf{OK} \end{array}\right\} \Longrightarrow \begin{array}{l} \exists {}^sT'.\ I.\ \ ({}^sN \rhd {}^sT)\left[\overline{{}^sT_l}/\overline{X_l}\right] = {}^sT' \wedge \\ \qquad II.\ \ h, {}^r\Gamma \vdash v : {}^sT' \end{array}$$

From 5. we know that $I.$ is defined for some ${}^sT'$.

Corresponding to Def. 2.2.15 we split $II.$ into the two parts:

$$\begin{array}{ll} IIa. & \exists \overline{\sigma_l}'.\ \exists {}^rT'.\ \mathrm{dyn}\left({}^sT', h, {}^r\Gamma, \overline{\sigma_l}'\right) = {}^rT' \qquad h \vdash v : {}^rT' \\ IIb. & {}^sT' = \texttt{self \_<\_>} \Longrightarrow v = {}^r\Gamma(\texttt{this}) \end{array}$$

We have 1., 3., 4., 5., and 6. and can therefore apply Lemma A.1.1 to derive:

$$\begin{array}{l} \exists \overline{\sigma_l}, {}^rT.\ \mathrm{dyn}({}^sT, h, {}^r\Gamma', \overline{\sigma_l}) = {}^rT \wedge \\ \exists {}^sT'.\ \ ({}^sN \rhd {}^sT)\left[\overline{{}^sT_l}/\overline{X_l}\right] = {}^sT' \wedge \\ \exists \overline{\sigma_l}', {}^rT'.\ \mathrm{dyn}\left({}^sT', h, {}^r\Gamma, \overline{\sigma_l}'\right) = {}^rT' \wedge \\ \qquad {}^rT = {}^rT' \end{array}$$

Corresponding to Def. 2.2.15 we split 2. into the two parts:

$$\begin{array}{l} \exists \overline{\sigma_l}, {}^rT.\ \mathrm{dyn}({}^sT, h, {}^r\Gamma', \overline{\sigma_l}) = {}^rT \qquad h \vdash v : {}^rT \\ {}^sT = \texttt{self \_<\_>} \Longrightarrow v = {}^r\Gamma'(\texttt{this}) \end{array}$$

We therefore have $h \vdash v : {}^rT$ and, for a corresponding choice of the $\overline{\sigma_l}$, that ${}^rT = {}^rT'$ and can therefore conclude IIa.

From Lemma A.1.3 using 7. and $I.$ we know that ${}^sT' = \texttt{self \_<\_>}$ can only hold, if also ${}^sT = \texttt{self \_<\_>}$ and that ${}^sN = \texttt{self \_<\_>}$. Note how the substitution of the $\overline{{}^sT_l}$ has no effect on the main modifier.

In the case where ${}^sT' = \texttt{self \_<\_>}$ we therefore have that ${}^r\Gamma(\texttt{this}) = \iota$ and $v = \iota$ and have IIb. □

A.2.2.3 Proof of Lemma A.1.2 — Adaptation to a Viewpoint Auxiliary Lemma

We prove:

$$
\begin{array}{rl}
& 1.\quad h, {}^{r}\Gamma \vdash \iota : {}^{s}N \\
& 2.\quad {}^{s}N \rhd {}^{s}T \;=\; {}^{s}T' \qquad {}^{s}T' \left[\overline{{}^{s}T_l}/\overline{X_l}\right] = {}^{s}T'' \\
& 3.\quad \texttt{lost} \notin {}^{s}T'' \\
& 4.\quad \mathrm{dyn}\left(\overline{{}^{s}T_l}, h, {}^{r}\Gamma, \emptyset\right) = \overline{{}^{r}T_l} \\
& 5.\quad {}^{r}\Gamma' = \left\{\overline{X_l \mapsto {}^{r}T_l}\ ;\ \texttt{this} \mapsto \iota, \_\right\} \\
& 6.\quad \mathrm{ClassDom}(\mathrm{ClassOf}({}^{s}N)) = \overline{X} \\
& 7.\quad \mathrm{free}({}^{s}T) \subseteq \overline{X}, \overline{X_l} \\
\Longrightarrow & \\
& \exists {}^{r}T.\ I.\quad \mathrm{dyn}({}^{s}T, h, {}^{r}\Gamma', \emptyset) = {}^{r}T \wedge \\
& \exists {}^{r}T'.\ II.\quad \mathrm{dyn}({}^{s}T'', h, {}^{r}\Gamma, \emptyset) = {}^{r}T' \wedge \\
& \qquad III.\quad {}^{r}T = {}^{r}T'
\end{array}
$$

Note that in 2. we split viewpoint adaptation and substitution into two steps.

The proofs of $I.$ and $II.$ are simple:

- $I.$ is defined because of 1., 5., and 7. we know that all type variables can be substituted and because of 3. and Lemma A.1.4 we know that `lost` is not contained in ${}^{s}T$ and we can therefore use an empty substitution.
- $II.$ is defined because of 1. and 7. we know that all type variables can be substituted and because of 4. and Lemma A.1.29 we know that `lost` is not contained in any of the $\overline{{}^{s}T_l}$ and from 3. we know that `lost` is not introduced into ${}^{s}T'$ and we can therefore use an empty substitution.

The proof of $III.$ runs by induction on the shape of ${}^{s}T$; from 7. we know that we have the following 4 cases:

Case 1: ${}^{s}T = X_i$ **and** $X_i \in \overline{X_l}$

${}^{s}N \rhd {}^{s}T \;=\; {}^{s}T'$ is therefore equal to ${}^{s}N \rhd X_i \;=\; X_i$, because of the assumption that $\overline{X}$ and $\overline{X_l}$ are distinct sets of type variables and no substitution takes place.

$({}^{s}N \rhd {}^{s}T) \left[\overline{{}^{s}T_l}/\overline{X_l}\right] \;=\; {}^{s}T''$ is equal to $X_i \left[\overline{{}^{s}T_l}/\overline{X_l}\right] = {}^{s}T_i$, the i-th element of the sequence of types $\overline{{}^{s}T_l}$.

$\mathrm{dyn}({}^{s}T, h, {}^{r}\Gamma', \emptyset) = {}^{r}T$ corresponds to $\mathrm{dyn}(X_i, h, {}^{r}\Gamma', \emptyset) = {}^{r}T$. From the definition of dyn we know that ${}^{r}\Gamma'(X_i) = {}^{r}T$. From 4. and 5. we know that ${}^{r}T$ is also $\mathrm{dyn}({}^{s}T_i, h, {}^{r}\Gamma, \emptyset) = {}^{r}T$.

Case 2: ${}^{s}T = X_j$ **and** $X_j \in \overline{X}$

${}^{s}N \rhd {}^{s}T \;=\; {}^{s}T'$ is therefore equal to ${}^{s}N \rhd X_j \;=\; {}^{s}T_j$, where ${}^{s}T_j$ is the j-th type argument of ${}^{s}N$.

$({}^{s}N \rhd {}^{s}T) \left[\overline{{}^{s}T_l}/\overline{X_l}\right] \;=\; {}^{s}T''$ is equal to ${}^{s}T_j \left[\overline{{}^{s}T_l}/\overline{X_l}\right] = {}^{s}T_j$, because of the assumption that the $\overline{X_l}$ do not appear in ${}^{s}N$.

From 1. we know for some ${}^{o}\iota'$, ${}^{o}\iota$, and $\overline{{}^{r}T}$ that $\mathrm{dyn}\left({}^{s}N, h, {}^{r}\Gamma, \overline{{}^{o}\iota'}\right) = {}^{o}\iota\ C\texttt{<}\overline{{}^{r}T}\texttt{>}$ is defined.

From the definition of dyn and 1. we know that $\mathrm{dyn}({}^{s}T_j, h, {}^{r}\Gamma, \emptyset) = {}^{r}T_j$, the j-th type argument in $\overline{{}^{r}T}$. As, if the dynamization of a non-variable type is defined, also the dynamization of the type arguments of that type is defined.

Because of 1. and dyn we know that $\mathrm{dyn}(X_l, h, {}^{r}\Gamma', \emptyset) = {}^{r}T_j$ We know that ${}^{r}\Gamma'(\texttt{this})$ is ι and can therefore use the knowledge of ${}^{s}N$. Therefore, both dynamizations result in the same type.

Case 3: ${}^sT = u\ C\texttt{<>}$ We name the main modifier of ${}^sN$ as u_1, that is $\mathrm{om}({}^sN) = u_1$.

${}^sN \rhd {}^sT\ =\ u'\ C\texttt{<>}$ where $u_1 \rhd u\ =\ u'$. Because there are no type variables, we also have $({}^sN \rhd {}^sT)\left[\overline{{}^sT_l}/\overline{X_l}\right]\ =\ u'\ C\texttt{<>}$.

We make a further case analysis for the possible values of u'.

Case 3a: $u' = \texttt{peer}$

There are two combinations that result in $u' = \texttt{peer}$:

1. $(u_1 = \texttt{self}\ \wedge\ u = \texttt{peer}) \implies u_1 \rhd u\ =\ \texttt{peer}$ and
2. $(u_1 = \texttt{peer}\ \wedge\ u = \texttt{peer}) \implies u_1 \rhd u\ =\ \texttt{peer}$.

$\mathrm{dyn}({}^sT, h, {}^r\Gamma', \emptyset)\ = {}^rT$ is equal to $\mathrm{dyn}(\texttt{peer}\ C\texttt{<>}, h, {}^r\Gamma', \emptyset)\ = {}^rT$. From the definition of dyn we know that `peer` is substituted by the owner of the current object, i.e., ${}^r\Gamma'(\texttt{this}) = \iota$, $\mathrm{owner}(h, \iota) = {}^o\iota_1$, and ${}^rT = {}^o\iota_1\ C\texttt{<>}$.

$\mathrm{dyn}({}^sT'', h, {}^r\Gamma, \emptyset)\ = {}^rT'$ is equal to $\mathrm{dyn}(\texttt{peer}\ C\texttt{<>}, h, {}^r\Gamma, \emptyset)\ = {}^rT'$. From the definition of dyn we know that `peer` is substituted by the owner of the current object, i.e., ${}^r\Gamma(\texttt{this}) = \iota_2$, $\mathrm{owner}(h, \iota_2) = {}^o\iota_2$, and ${}^rT' = {}^o\iota_2\ C\texttt{<>}$.

From 1. and our knowledge of u_1 we know that we can assign `self _<_>` or `peer _<_>` to ι. Therefore, we know that the owner of ι is the owner of ι_2, that is, ${}^o\iota_1 = {}^o\iota_2$.

Case 3b: $u' = \texttt{rep}$

There are two combinations that result in $u' = \texttt{rep}$:

1. $(u_1 = \texttt{self}\ \wedge\ u = \texttt{rep}) \implies u_1 \rhd u\ =\ \texttt{rep}$ and
2. $(u_1 = \texttt{rep}\ \wedge\ u = \texttt{peer}) \implies u_1 \rhd u\ =\ \texttt{rep}$.

$\mathrm{dyn}({}^sT'', h, {}^r\Gamma, \emptyset)\ = {}^rT'$ is equal to $\mathrm{dyn}(\texttt{rep}\ C\texttt{<>}, h, {}^r\Gamma, \emptyset)\ = {}^rT'$. From the definition of dyn we know that `rep` is substituted by the current object, i.e., ${}^r\Gamma(\texttt{this}) = \iota_2$ and ${}^rT' = \iota_2\ C\texttt{<>}$.

We make a further distinction of the two cases:

Case 3b1:. $\mathrm{dyn}({}^sT, h, {}^r\Gamma', \emptyset)\ = {}^rT$ is equal to $\mathrm{dyn}(\texttt{rep}\ C\texttt{<>}, h, {}^r\Gamma', \emptyset)\ = {}^rT$. From the definition of dyn we know that `rep` is substituted by the current object, i.e., ${}^r\Gamma'(\texttt{this}) = \iota$ and ${}^rT = \iota\ C\texttt{<>}$.

From 1. we know that we can assign `self _<_>` to ι. Therefore, we know that the ${}^r\Gamma(\texttt{this}) = \iota$ and we have that $\iota = \iota_2$ and therefore that the types are equal.

Case 3b2:. $\mathrm{dyn}({}^sT, h, {}^r\Gamma', \emptyset)\ = {}^rT$ is equal to $\mathrm{dyn}(\texttt{peer}\ C\texttt{<>}, h, {}^r\Gamma', \emptyset)\ = {}^rT$. From the definition of dyn we know that `peer` is substituted by the owner of the current object, i.e., ${}^r\Gamma'(\texttt{this}) = \iota\ \wedge\ \mathrm{owner}(h, \iota) = {}^o\iota_1$ and ${}^rT = {}^o\iota_1\ C\texttt{<>}$.

From 1. we know that we can assign `rep _<_>` to ι. Therefore, we know that ${}^r\Gamma(\texttt{this}) = \iota_2\ \wedge\ \mathrm{owner}(h, \iota) = \iota_2$.

The owner of an address is unique and therefore ${}^o\iota_1 = \iota_2$ and the types are equal.

Case 3c: $u' = \texttt{any}$

For an arbitrary u_1 we have $u = \texttt{any} \implies u_1 \rhd u\ =\ \texttt{any}$.

From the definition of dyn we see that the `any` ownership modifier is replaced by the $\texttt{any}_a$ address, regardless of environment and heap. Therefore, both dynamizations are the same.

Case 3d: $u' = \texttt{lost}$

This case is forbidden by 3.

Case 4: ${}^sT = u\ C\texttt{<}\overline{{}^sT}\texttt{>}$

Let us first give names to the result of the application of dyn:
$\mathrm{dyn}({}^sT, h, {}^r\Gamma', \emptyset) = \iota''\ C\texttt{<}\overline{{}^rT}''\texttt{>}$
$\mathrm{dyn}({}^sT'', h, {}^r\Gamma, \emptyset) = \iota'''\ C\texttt{<}\overline{{}^rT}'''\texttt{>}$

The same analysis as case 3. before results in $\iota'' = \iota'''$.

Now we apply the induction hypothesis to the $\overline{{}^sT}$ and use 1., 2., 3. and the fact that $\mathrm{free}({}^sT) \subseteq \mathrm{ClassDom}(\mathrm{ClassOf}({}^sN)), \overline{X}.$ implies $\mathrm{free}(\overline{{}^sT}) \subseteq \mathrm{ClassDom}(\mathrm{ClassOf}({}^sN)), \overline{X}$.

Therefore we have that $\iota'' = \iota'''$ and $\overline{{}^rT}'' = \overline{{}^rT}'''$.

From Lemma A.1.28 we then know that the runtime types are the same. □

A.2.2.4 Proof of Lemma 2.3.29 — Adaptation to a Viewpoint

We prove:

$$\left.\begin{array}{ll}
1. & h, {}^r\Gamma \vdash \iota : {}^sN \\
2. & ({}^sN \rhd {}^sT)\left[\overline{{}^sT_l}/\overline{X_l}\right] = {}^sT' \\
3. & \texttt{lost} \notin {}^sT' \\
4. & h, {}^r\Gamma \vdash v : {}^sT' \\
5. & \mathrm{dyn}\left(\overline{{}^sT_l}, h, {}^r\Gamma, \emptyset\right) = \overline{{}^rT_l} \\
6. & {}^r\Gamma' = \left\{\overline{X_l \mapsto {}^rT_l}\ ;\ \texttt{this} \mapsto \iota, _\right\} \\
7. & \mathrm{ClassDom}(\mathrm{ClassOf}({}^sN)) = \overline{X} \\
8. & \mathrm{free}({}^sT) \subseteq \overline{X}, \overline{X_l} \\
9. & {}^s\Gamma \vdash {}^sN\ \mathsf{OK}
\end{array}\right\} \Longrightarrow\ h, {}^r\Gamma' \vdash v : {}^sT$$

Corresponding to Def. 2.2.15, we split the goal into the two parts:

$$\begin{array}{ll}
I. & \exists\overline{o_l}.\ \exists{}^rT.\ \mathrm{dyn}({}^sT, h, {}^r\Gamma', \overline{o_l}) = {}^rT \qquad h \vdash v : {}^rT \\
II. & {}^sT = \texttt{self _<_>} \Longrightarrow v = {}^r\Gamma'(\texttt{this})
\end{array}$$

We have 1., 2., 3., and 5. to 8. and can therefore apply Lemma A.1.2 to derive the existence of ${}^rT$ and ${}^rT'$ such that:

$$\begin{array}{l}
\mathrm{dyn}({}^sT, h, {}^r\Gamma', \emptyset) = {}^rT \\
\mathrm{dyn}\left({}^sT'\left[\overline{{}^sT_l}/\overline{X_l}\right], h, {}^r\Gamma, \emptyset\right) = {}^rT' \\
{}^rT = {}^rT'
\end{array}$$

In $I.$ we therefore can use $\overline{o_l} = \emptyset$ to find ${}^rT$. From 4. we know that $h \vdash v : {}^rT'$ holds. Because the two runtime types are equal we therefore directly have $h \vdash v : {}^rT$ and have I.

From Lemma A.1.3, using 2. and 9., we know that ${}^sT' = \texttt{self \_<\_>}$ can only hold, if also ${}^sT = \texttt{self \_<\_>}$ and that ${}^sN = \texttt{self \_<\_>}$.

Therefore, in the case where ${}^sT = \texttt{self \_<\_>}$, we have that ${}^r\Gamma(\texttt{this}) = \iota$ and $v = \iota$ and have II. □

A.2.3 Well-formedness Properties

A.2.3.1 Proof of Lemma A.1.5 — Well-formedness and Viewpoint Adaptation

We prove:

$$
\begin{array}{ll}
1. & \text{ClassDom}(C) = \overline{X_k} \qquad \text{ClassBnds}(C) = \overline{{}^sN_k} \\
2. & {}^s\Gamma = \left\{ \overline{X_k \mapsto {}^sN_k}\,,\ \overline{X'_l \mapsto {}^sN'_l}\ ;\ \texttt{this} \mapsto \texttt{self}\ C\texttt{<}\overline{X_k}\texttt{>}, \_ \right\} \\
3. & {}^s\Gamma \vdash {}^sT\ \ \mathsf{OK} \\
4. & {}^s\Gamma' \vdash {}^sN\ \ \mathsf{OK} \qquad \text{ClassOf}({}^sN) = C \\
5. & \left({}^sN \rhd \overline{{}^sN'_l}\right)\left[\overline{{}^sT_l}/\overline{X'_l}\right] = \overline{{}^sN''_l} \\
6. & {}^s\Gamma' \vdash \overline{{}^sT_l}\ \ \mathsf{strictly\ OK} \\
7. & {}^s\Gamma' \vdash \overline{{}^sT_l} <: \overline{{}^sN''_l}
\end{array}
$$

$$
\Longrightarrow
$$

$$
\begin{array}{ll}
\exists {}^sT'. & I. \quad ({}^sN \rhd {}^sT)\left[\overline{{}^sT_l}/\overline{X'_l}\right] = {}^sT' \\
& II. \quad {}^s\Gamma' \vdash {}^sT'\ \ \mathsf{OK}
\end{array}
$$

Note that $\text{free}({}^sT) \subseteq \overline{X_k}, \overline{X'_l}$ is not needed as a separate requirement, it follows directly from 3.

Conclusion $I.$ is always defined, because of 3. and 4.

The proof of conclusion $II.$ runs by induction on the shape of ${}^sT$ and deriving the requirements for Def. 2.3.11. Cases 1, 2, and 3 are the base cases and case 4 is the induction step.

Case 1: ${}^sT = X_i$ and $X_i \in \overline{X_k}$

If ${}^sT$ is one of the class type variables, we know from the definition of $I.$ that ${}^sT' = {}^sT_i$, the i-th type argument of ${}^sN$. From 4. and the definition of well-formed type, we know that also the type arguments are well formed.

Case 2: ${}^sT = X_i$ and $X_i \in \overline{X'_l}$

If ${}^sT$ is one of the method type variables, we know from the definition of $I.$ that ${}^sT' = {}^sT_i$, the i-th method type argument. From 6. we know that the method type arguments are strictly well formed and therefore, by Lemma A.1.15, also well formed.

Case 3: ${}^sT = u\ C'\texttt{<>}$

If ${}^sT$ is a non-generic type, then the viewpoint adaptation might change the ownership modifier in ${}^sT'$, but the type ${}^sT'$ will still be well formed.

Case 4: ${}^sT = u\ C'\texttt{<}\overline{{}^sT'_k}\texttt{>}$

The shape of ${}^sT'$ will be ${}^sT' = u'\ C'\texttt{<}\overline{{}^sT''_k}\texttt{>}$, i.e. it is also a non-variable type with class C'.

From 3. and the definition of well-formed static type, we know that ${}^s\Gamma \vdash \overline{{}^sT'_k}\ \ \mathsf{OK}$, $\texttt{self} \notin \overline{{}^sT'_k}$, and ${}^s\Gamma \vdash \overline{{}^sT'_k} <: \overline{{}^sN''_k}$, where $\text{ClassBnds}\left(u\ C'\texttt{<}\overline{{}^sT'_k}\texttt{>}\right) = \overline{{}^sN''_k}$.

With ${}^s\Gamma \vdash \overline{{}^sT'_k}\ \ \mathsf{OK}$ we can apply the induction hypothesis to the $\overline{{}^sT'_k}$ to arrive at $\left({}^sN \rhd \overline{{}^sT'_k}\right)\left[\overline{{}^sT_l}/\overline{X'_l}\right] = \overline{{}^sT''_k}$ and ${}^s\Gamma' \vdash \overline{{}^sT''_k}\ \ \mathsf{OK}$.

We know from 4. that the type arguments of ${}^sN$ and from 6. that the $\overline{{}^sT_l}$ do not contain `self`. Together with Lemma A.1.3 this gives us that the $\overline{{}^sT''_k}$ do not contain `self`.

Finally, we know from 3., 4., and 7. that the upper bounds of ${}^sT'$ are respected, meeting the last requirement for ${}^s\Gamma' \vdash {}^sT'\ \ \mathsf{OK}$. □

A.2.3.2 Proof of Lemma A.1.10 — Strict Static Well-formedness implies Dynamization and Runtime Well-formedness

We prove:

$$\left.\begin{array}{ll}1. & \vdash P \ \ \mathsf{OK} \\ 2. & h, {}^{r}\Gamma : {}^{s}\Gamma \ \ \mathsf{OK} \\ 3. & {}^{s}\Gamma \vdash {}^{s}T \ \ \mathsf{strictly\ OK}\end{array}\right\} \Longrightarrow \begin{array}{ll}\exists {}^{r}T. & I. \quad \mathrm{dyn}({}^{s}T, h, {}^{r}\Gamma, \emptyset) = {}^{r}T \wedge \\ & II. \quad h, \_ \vdash {}^{r}T \ \ \mathsf{strictly\ OK}\end{array}$$

Part I:

from 3. using Lemma A.1.15 we deduce ${}^{s}\Gamma \vdash {}^{s}T \ \ \mathsf{OK}$. With this, 1., and 2. we can use Lemma A.1.9 to deduce that dyn is defined for some $\overline{q}$. From 3. we know that there are no `lost` ownership modifiers in ${}^{s}T$ and therefore we can use an empty substitution to define I.

Part II:

For the proof of part II. we use an induction on the shape of the static type ${}^{s}T$. Cases 1, 2, and 3 are the base cases and case 4 is the induction step.

Case 1: ${}^{s}T = X_i$ and X_i is a class type variable

From 2. we know that the heap is well formed and that the static and runtime environments correspond. Therefore, the X_i will be substituted by the corresponding type argument in the heap, i.e., the runtime type of ${}^{r}\Gamma(\texttt{this})$ will be used for the substitution.

As the heap is well formed we know that all runtime types in the heap are well formed, in particular, also all type arguments of the runtime types. If X_i is a type variable of the class of ${}^{r}\Gamma(\texttt{this})$ we can therefore directly conclude that ${}^{r}T$ is well formed.

However, the type variable X_i might have been declared in one of the superclasses of the type of ${}^{r}\Gamma(\texttt{this})$; let us call the class of ${}^{r}\Gamma(\texttt{this})$ class C and the superclass that declares X_i as C'. We also know that C' is the class of ${}^{s}\Gamma(\texttt{this})$ from 3., because otherwise the X_i would not be well formed. From 2. and Def. 2.3.25 we know that
$h, {}^{r}\Gamma \vdash {}^{r}\Gamma(\texttt{this}) : \texttt{self}\ C'\texttt{<}\overline{X}'\texttt{>}$
for the class C with domain $\overline{X}$ that declares type variable X_i.

Using Def. 2.2.15, Def. 2.2.14, Def. 2.2.13, Def. 2.2.12, and finally Def. 2.2.11 we derive that the runtime type arguments for C' are the dynamization of the type arguments $\overline{{}^{s}T}$ from the subclassing relation $C\texttt{<}\overline{X}\texttt{>} \sqsubseteq C'\texttt{<}\overline{{}^{s}T}\texttt{>}$. Using Lemma A.1.17, Lemma A.1.19, and Lemma A.1.20 we derive that the $\overline{{}^{s}T}$ are strictly well-formed types that do not contain `lost` and that are subtypes of the upper bounds of class C', in a static environment that maps the type variables of C to their upper bounds.

Finally, the runtime type of ${}^{r}\Gamma(\texttt{this})$ is used for the simple dynamization of the $\overline{{}^{s}T}$ and from the well-formedness of the type of ${}^{r}\Gamma(\texttt{this})$ and the above we can derive that the runtime supertype that is used for the substitution of X_i is well formed.

Case 2: ${}^{s}T = X_i$ and X_i is a method type variable

From 2. we know that the runtime environment is well formed and that the static and runtime environments correspond. The method type variable X_i will be substituted by the corresponding runtime type in the runtime environment. From 2. we know that this type is well formed.

Case 3: ${}^{s}T = u\ C\texttt{<>}$ a non-generic class

From 3. we know that the static type is well formed and therefore that the class name is valid. The ownership modifier u is replaced by one of the addresses determined from the heap

and runtime environment. By 2. these are well formed and therefore also the runtime type ${}^{r}T$ is well formed.

Case 4: ${}^{s}T = u\ C\texttt{<}\overline{{}^{s}T_k}\texttt{>}$ a generic class

From the definition of dyn we know that ${}^{r}T = {}^{o}\iota\ C\texttt{<}\overline{{}^{r}T_k}\texttt{>}$.

In this case, by Def. 2.3.22, we need to show:

1. $h,\ \iota \vdash \overline{{}^{r}T_k}$ strictly OK;
2. ${}^{o}\iota$ is in the domain of h, $\texttt{root}_a$, or $\texttt{any}_a$;
3. the type arguments are subtypes of their upper bounds.

We apply the induction hypothesis to the $\overline{{}^{s}T_k}$. Requirements 1. and 2. are unchanged and from 3. we know that also the type arguments $\overline{{}^{s}T_k}$ are strictly well formed. We therefore have that the dynamization of the type arguments is defined and that $h,\ \iota \vdash \overline{{}^{r}T_k}$ strictly OK.

From the definition of dyn we know that ${}^{o}\iota$ is either an address in the heap, $\texttt{root}_a$, or $\texttt{any}_a$. From Lemma A.1.13 and 3. we know that the upper bounds of C do not contain `rep`. Therefore, by Def. 2.3.22, we can choose an arbitrary viewpoint address for the well-formedness check. Also, recursively, the upper bounds of the classes of the type argument cannot contain `rep`, as proved in Lemma A.1.13. This allows us to also choose arbitrary viewpoint addresses for the type arguments.

What remains to be shown for $h,\ \iota \vdash {}^{r}T$ strictly OK is that the type arguments $\overline{{}^{r}T_k}$ are subtypes of the corresponding runtime upper bounds. This proof is relegated to Lemma A.1.11, which can be directly applied for this case. The first four requirements correspond to 1., 2., 3., and I.; the last requirement just gives a name to the result of ClassBnds. The application of ClassBnds is defined: we know that the upper bounds of class C do not contain `lost` from 3. and can therefore apply sdyn as required. □

A.2.3.3 Proof of Lemma A.1.11 — Correct Checking of Class Upper Bounds

We prove:

$$\left.\begin{array}{ll} 1. & \vdash P \ \text{OK} \\ 2. & h, {}^{r}\Gamma : {}^{s}\Gamma \ \text{OK} \\ 3. & {}^{s}\Gamma \vdash u\ C\texttt{<}\overline{{}^{s}T_k}\texttt{>} \ \text{strictly OK} \\ 4. & \mathrm{dyn}\Big(u\ C\texttt{<}\overline{{}^{s}T_k}\texttt{>}, h, {}^{r}\Gamma, \emptyset\Big) = {}^{o}\iota\ C\texttt{<}\overline{{}^{r}T_k}\texttt{>} \\ 5. & \mathrm{ClassBnds}\Big(h, \iota, {}^{o}\iota\ C\texttt{<}\overline{{}^{r}T_k}\texttt{>}, \emptyset\Big) = \overline{{}^{r}T'_k} \end{array}\right\} \Longrightarrow\ h \vdash \overline{{}^{r}T_k} <:\ \overline{{}^{r}T'_k}$$

From 3., using Def. 2.3.12, we know:

$$\begin{array}{ll} 6. & {}^{s}\Gamma \vdash \overline{{}^{s}T_k} \ \text{strictly OK} \\ 7. & \{\texttt{self}, \texttt{lost}\} \notin u\ C\texttt{<}\overline{{}^{s}T_k}\texttt{>} \\ 8. & \mathrm{ClassBnds}\Big(u\ C\texttt{<}\overline{{}^{s}T_k}\texttt{>}\Big) = \overline{{}^{s}N_k} \\ 9. & \texttt{lost} \notin \overline{{}^{s}N_k} \\ 10. & {}^{s}\Gamma \vdash \overline{{}^{s}T_k} <:\ \overline{{}^{s}N_k} \end{array}$$

From 5., using Def. 2.3.21, we know:

$$\begin{array}{ll} 11. & \mathrm{ClassBnds}(C) = \overline{{}^{s}N'_k} \\ 12. & \mathrm{sdyn}\Big(\overline{{}^{s}N'_k}, h, \iota, {}^{o}\iota\ C\texttt{<}\overline{{}^{r}T_k}\texttt{>}, \emptyset\Big) = \overline{{}^{r}T'_k} \end{array}$$

Note the relationship between the $\overline{{}^{s}N_k}$ and $\overline{{}^{s}N'_k}$ from Def. 2.3.6: $u\ C\texttt{<}\overline{{}^{s}T_k}\texttt{>} \rhd \overline{{}^{s}N'_k} = \overline{{}^{s}N_k}$. In particular, how the type arguments $\overline{{}^{s}T_k}$ are substituted in the declared upper bounds. Similarly, the sdyn substitutes the runtime type arguments $\overline{{}^{r}T_k}$ in the declared upper bounds. From Lemma A.1.14 we deduce that type variables declared by class C are used in these upper bounds and fulfill this requirement of sdyn. Finally note the relationship between the $\overline{{}^{s}T_k}$ and the $\overline{{}^{r}T_k}$ through the dynamization in 4.

The lemma obviously holds if the sequence of type arguments $\overline{{}^{s}T_k}$ is empty. Otherwise, we have to show for each ${}^{r}T \in \overline{{}^{r}T_k}$ and the corresponding ${}^{r}T' \in \overline{{}^{r}T'_k}$ that $h \vdash {}^{r}T <:\ {}^{r}T'$.

We proceed in two steps: first, using Lemma A.1.26, we show that ${}^{r}T'$ obtained from sdyn in 12. corresponds to a type obtained using dyn. Secondly, using this result and Lemma A.1.23, we show that the static subtype relation 10. also holds for the two runtime types, i.e., that $h \vdash {}^{r}T <:\ {}^{r}T'$ holds.

Part I:

Using 1., 2., 3., 4., 8., 9., and 12. in Lemma A.1.26 we deduce that

$$\left.\begin{array}{ll} 1_0. & \vdash P\ \ \mathsf{OK} \\ 2_0. & h, {}^{r}\Gamma : {}^{s}\Gamma\ \ \mathsf{OK} \\ 3_0. & {}^{s}\Gamma \vdash u\ C\texttt{<}\overline{{}^{s}T_k}\texttt{>}\ \ \mathsf{strictly\ OK} \\ 4_0. & \mathrm{dyn}\Big(u\ C\texttt{<}\overline{{}^{s}T_k}\texttt{>}, h, {}^{r}\Gamma, \emptyset\Big) = {}^{o}\iota\ C\texttt{<}\overline{{}^{r}T_k}\texttt{>} \\ 5_0. & u\ C\texttt{<}\overline{{}^{s}T_k}\texttt{>} \rhd \overline{{}^{s}N'_k} = \overline{{}^{s}N_k} \\ 6_0. & \texttt{lost} \notin \overline{{}^{s}N_k} \\ 7_0. & \mathrm{sdyn}\Big(\overline{{}^{s}N'_k}, h, \iota, {}^{o}\iota\ C\texttt{<}\overline{{}^{r}T_k}\texttt{>}, \emptyset\Big) = \overline{{}^{r}T'_k} \end{array}\right\} \Longrightarrow\ 13.\quad \mathrm{dyn}\Big(\overline{{}^{s}N_k}, h, {}^{r}\Gamma, \emptyset\Big) = \overline{{}^{r}T'_k}$$

That is, that the simple dynamization of the declared class upper bounds is equal to the dynamization of the viewpoint-adapted upper bounds.

Part II:

Using Corollary A.1.8 using 1., 3., and the knowledge that a well-formed class has well-formed upper bounds, we deduce that ${}^{s}\Gamma \vdash \overline{{}^{s}N_k}\ \ \mathsf{OK}$, that is, that the viewpoint-adapted upper bounds are well formed in the environment ${}^{s}\Gamma$.

Using this, 1., 2., 6., and 10. with Lemma A.1.23 we can conclude that $h \vdash {}^{r}T <:\ {}^{r}T'$ holds. Note that from 4. and 9. we know that the `lost` modifier is not contained in the two static types and therefore we can use empty substitutions in the dynamizations. □

A.2.3.4 Proof of Lemma A.1.12 — Correct Checking of Method Upper Bounds

This lemma is needed to show that the environment in a method call is well formed.

We prove:

$$\left.\begin{array}{ll} 1. & \vdash P\ \ \mathsf{OK} \\ 2. & h, {}^{r}\Gamma : {}^{s}\Gamma\ \ \mathsf{OK} \\ 3. & ({}^{s}N \rhd {}^{s}T_0)\left[\overline{{}^{s}T}/\overline{X}\right] = {}^{s}T' \\ 4. & \texttt{lost} \notin {}^{s}T' \\ 5. & {}^{s}\Gamma \vdash {}^{s}T, {}^{s}T'\ \ \mathsf{OK} \\ 6. & {}^{s}\Gamma \vdash {}^{s}T <:\ {}^{s}T' \\ 7. & h, {}^{r}\Gamma \vdash \iota :\ {}^{s}N \\ 8. & h, {}^{r}\Gamma \vdash {}^{s}N, {}^{s}T_0; \left(\overline{{}^{s}T}/\overline{X}, \iota\right) = {}^{r}\Gamma' \end{array}\right\} \Longrightarrow \begin{array}{rll} \exists \overline{{}^{o}\iota}, {}^{r}T. & I. & \mathrm{dyn}({}^{s}T, h, {}^{r}\Gamma, \overline{{}^{o}\iota}) = {}^{r}T \wedge \\ \exists {}^{r}T'. & II. & \mathrm{dyn}({}^{s}T_0, h, {}^{r}\Gamma', \emptyset) = {}^{r}T' \wedge \\ & III. & h \vdash {}^{r}T <:\ {}^{r}T' \end{array}$$

Using 1., 2., 5., and 6. with Lemma A.1.23 we deduce:

$$\left.\begin{array}{l} \vdash P \ \ \mathsf{OK} \\ h, {}^{r}\Gamma : {}^{s}\Gamma \ \ \mathsf{OK} \\ {}^{s}\Gamma \vdash {}^{s}T\,,\ {}^{s}T' \ \ \mathsf{OK} \\ {}^{s}\Gamma \vdash {}^{s}T <: \ {}^{s}T' \end{array}\right\} \Longrightarrow \begin{array}{l} \exists \overline{o_l}, {}^{r}T.\ \ \mathrm{dyn}({}^{s}T, h, {}^{r}\Gamma, \overline{o_l}) = {}^{r}T \wedge \\ \exists \overline{o_l}'', {}^{r}T''.\ \ \mathrm{dyn}\left({}^{s}T', h, {}^{r}\Gamma, \overline{o_l}''\right) = {}^{r}T'' \wedge \\ \qquad h \vdash {}^{r}T <: \ {}^{r}T'' \end{array}$$

Using 7., 3., 4., and 8. with Lemma A.1.2 we further deduce:

$$\left.\begin{array}{l} h, {}^{r}\Gamma \vdash \iota : \ {}^{s}N \\ ({}^{s}N \rhd {}^{s}T_0)\left[\overline{{}^{s}T}/\overline{X}\right] \ = \ {}^{s}T' \\ \texttt{lost} \notin {}^{s}T' \\ h, {}^{r}\Gamma \vdash {}^{s}N, {}^{s}T_0; \left(\overline{{}^{s}T}/\overline{X}, \iota\right) = {}^{r}\Gamma' \end{array}\right\} \Longrightarrow \begin{array}{l} \exists {}^{r}T'.\ \ \mathrm{dyn}({}^{s}T_0, h, {}^{r}\Gamma', \emptyset) = {}^{r}T' \wedge \\ \exists {}^{r}T''.\ \ \mathrm{dyn}({}^{s}T', h, {}^{r}\Gamma, \emptyset) = {}^{r}T'' \wedge \\ \qquad {}^{r}T' = {}^{r}T'' \end{array}$$

From these two results we can directly deduce $I.$, $II.$, and $III.$ □

A.2.3.5 Proof of Lemma A.1.14 — Free Variables in Upper Bounds and Superclass Instantiations

We prove:

$$\left.\begin{array}{ll} 1. & \vdash P \ \ \mathsf{OK} \\ 2. & C\texttt{<}\overline{X}\texttt{>} \sqsubseteq C'\texttt{<}\overline{{}^{s}T}\texttt{>} \end{array}\right\} \Longrightarrow \mathrm{free}\left(\overline{{}^{s}T}\right) \subseteq \overline{X}$$

This is done by an induction on the shape of the derivation of 2.

Case 1: last derivation applied is sc1

From 1. we know that the corresponding class declaration was checked for well formedness. This includes checking that the type `self` $C'\texttt{<}\overline{{}^{s}T}\texttt{>}$ is well formed in an environment that only maps the type variables of C, that is $\overline{X}$, to their upper bounds. This ensures that all free variables in $\overline{{}^{s}T}$ are contained in the $\overline{X}$.

Case 2: last derivation applied is sc2

In this case $\overline{{}^{s}T}$ is equals to $\overline{X}$ and the conclusion obviously holds.

Case 3: last derivation applied is sc3

In this case we apply the induction hypothesis to the two types separately. The final substitution replaces all free variables by types that are ensured to fulfill the property. □

We want to prove:

$$\left.\begin{array}{ll} 1. & \vdash P \ \ \mathsf{OK} \\ 2. & \mathrm{ClassDom}(C) = \overline{X} \\ 3. & \mathrm{ClassBnds}(C) = \overline{{}^{s}N} \end{array}\right\} \Longrightarrow \mathrm{free}\left(\overline{{}^{s}N}\right) \subseteq \overline{X}$$

From 1. we know that the corresponding class declaration was checked for well formedness. This includes checking that the types $\overline{{}^{s}N}$ are well formed in an environment that only maps the type variables of C, that is $\overline{X}$, to their upper bounds. This ensures that all free variables in $\overline{{}^{s}N}$ are contained in the $\overline{X}$. □

A.2.3.6 Proof of Lemma A.1.13 — Properties of Strictly Well-formed Static Types

The first part is

$$\left.\begin{array}{l} {}^{s}\Gamma \vdash u \ C\texttt{<}\overline{{}^{s}T}\texttt{>} \ \ \mathsf{strictly\ OK} \\ \mathrm{ClassBnds}(C) = \overline{{}^{s}N} \end{array}\right\} \Longrightarrow \texttt{rep} \notin \overline{{}^{s}N}$$

From the definition of strictly well-formed static type (Def. 2.3.12) we know that $u \neq \texttt{self}$ and that the viewpoint-adapted upper bounds of C do not contain `lost`.

If the un-adapted upper bounds contained `rep`, the adaptation with something other than `self` would contain `lost`. Therefore, we know that `rep` is not in the un-adapted upper bounds. □

The second part is

$$\left.\begin{array}{l} {}^{s}\Gamma \vdash \texttt{any}\ C\texttt{<}\overline{{}^{s}T}\texttt{>}\ \ \mathsf{strictly\ OK} \\ \mathrm{ClassBnds}(C)\ = \overline{{}^{s}N} \end{array}\right\} \Longrightarrow \texttt{peer} \notin \overline{{}^{s}N}$$

If we know that the main modifier is `any` we can conclude that the only ownership modifier that is used in the un-adapted upper bounds is `any`, because otherwise viewpoint adaptation would introduce `lost` and the type could not be strictly well formed. □

Note that the strictly well-formed type judgment recursively checks the type arguments and therefore we can also deduce the same properties about all the classes that are used in type arguments. Also note that the `self` and `lost` ownership modifiers in the upper bounds are directly forbidden by the strictly well-formed type judgment.

A.2.4 Ordering Relations

A.2.4.1 Proof of Lemma A.1.17 — Subclassing: Superclass Instantiation Uses Strictly Well-formed Types

We prove:

$$\left.\begin{array}{ll} 1. & \vdash P\ \ \mathsf{OK} \\ 2. & C\texttt{<}\overline{X}\texttt{>} \sqsubseteq C'\texttt{<}\overline{{}^{s}T}\texttt{>} \end{array}\right\} \Longrightarrow\ {}^{s}\Gamma \vdash \overline{{}^{s}T}\ \ \mathsf{strictly\ OK}$$

$$\begin{array}{l} \text{where } {}^{s}\Gamma = \left\{ \overline{X_k \mapsto {}^{s}N_k}\ ;\ \texttt{this} \mapsto \texttt{self}\ C\texttt{<}\overline{X_k}\texttt{>}, \_ \right\} \\ \quad \text{and } \mathrm{ClassDom}(C)\ = \overline{X_k} \text{ and } \overline{X} = \overline{X_k} \\ \quad \text{and } \mathrm{ClassBnds}(C)\ = \overline{{}^{s}N_k} \end{array}$$

We show for each ${}^{s}T$ in $\overline{{}^{s}T}$ that ${}^{s}\Gamma \vdash {}^{s}T\ \ \mathsf{strictly\ OK}$. We do this by an induction on the derivation of 2. (see Def. 2.2.1).

Case 1: sc1

From 1. we know that class C was checked to conform to Def. 2.3.14, which directly checks for strict well formedness of the type arguments of the superclass instantiation.

Case 2: sc2

This directly follows from 2. and the definition of ${}^{s}\Gamma$.

Case 3: sc3

We have $C\texttt{<}\overline{X}\texttt{>} \sqsubseteq C_1\texttt{<}\overline{{}^{s}T}_1\texttt{>}$ and $C_1\texttt{<}\overline{X}_1\texttt{>} \sqsubseteq C'\texttt{<}\overline{{}^{s}T}'\texttt{>}$ giving $C\texttt{<}\overline{X}\texttt{>} \sqsubseteq C'\texttt{<}\overline{{}^{s}T}\texttt{>}$, with the substitution $\overline{{}^{s}T} = \overline{{}^{s}T}'\left[\overline{{}^{s}T}_1/\overline{X}_1\right]$.

We apply the induction hypothesis to derive that
${}^{s}\Gamma \vdash \overline{{}^{s}T}_1\ \ \mathsf{strictly\ OK}$ and
${}^{s}\Gamma' \vdash \overline{{}^{s}T}'\ \ \mathsf{strictly\ OK}$
where ${}^{s}\Gamma'$ is the environment corresponding to the instantiation of the lemma for C_1.

From Lemma A.1.19 we can deduce that the $\overline{{}^{s}T}_1$ are subtypes of the upper bounds of class C_1. Therefore, the substitution of the $\overline{{}^{s}T}_1$ for the $\overline{X}_1$ in the $\overline{{}^{s}T}'$ maintains that the $\overline{{}^{s}T}$ are strictly well formed. □

A.2.4.2 Proof of Lemma A.1.19 — Subclassing: Superclass Instantiation Uses Subtypes of the Upper Bounds

We prove:

$$\left.\begin{array}{ll} 1. & \vdash P \ \ \mathsf{OK} \\ 2. & C\texttt{<}\overline{X}\texttt{>} \sqsubseteq C'\texttt{<}\overline{{}^{s}T}\texttt{>} \\ 3. & \text{ClassBnds}\Big(\texttt{self } C'\texttt{<}\overline{{}^{s}T}\texttt{>}\Big) = \overline{{}^{s}N} \end{array}\right\} \Longrightarrow \ {}^{s}\Gamma \vdash \overline{{}^{s}T} <: \ \overline{{}^{s}N}$$

where ${}^{s}\Gamma = \left\{\overline{X_k \mapsto {}^{s}N_k} \ ; \ \texttt{this} \mapsto \texttt{self } C\texttt{<}\overline{X_k}\texttt{>}, \_\right\}$
and $\text{ClassDom}(C) = \overline{X_k}$ and $\overline{X} = \overline{X_k}$
and $\text{ClassBnds}(C) = \overline{{}^{s}N_k}$

We show for each ${}^{s}T$ in $\overline{{}^{s}T}$ and the corresponding upper bound ${}^{s}N$ from $\overline{{}^{s}N}$ that ${}^{s}\Gamma \vdash {}^{s}T <: \ {}^{s}N$. We prove this by an induction on the derivation of 2. (see Def. 2.2.1).

Case 1: sc1
From 1. we know that class C was checked to conform to Def. 2.3.14, which directly checks for well formedness of the superclass instantiation ${}^{s}\Gamma \vdash \texttt{self } C'\texttt{<}\overline{{}^{s}T}\texttt{>} \ \ \mathsf{OK}$. This ensures that the $\overline{{}^{s}T}$ are subtypes of the upper bounds $\overline{{}^{s}N}$.

Case 2: sc2
This directly follows from 2. and the definition of ${}^{s}\Gamma$.

Case 3: sc3
We have $C\texttt{<}\overline{X}\texttt{>} \sqsubseteq C_1\texttt{<}\overline{{}^{s}T}_1\texttt{>}$ and $C_1\texttt{<}\overline{X}_1\texttt{>} \sqsubseteq C'\texttt{<}\overline{{}^{s}T}'\texttt{>}$ giving $C\texttt{<}\overline{X}\texttt{>} \sqsubseteq C'\texttt{<}\overline{{}^{s}T}\texttt{>}$, with the substitution $\overline{{}^{s}T} = \overline{{}^{s}T}'\left[\overline{{}^{s}T}_1/\overline{X}_1\right]$.

We apply the induction hypothesis to derive that ${}^{s}\Gamma \vdash \overline{{}^{s}T}_1 <: \ \overline{{}^{s}N}_1$ and ${}^{s}\Gamma' \vdash \overline{{}^{s}T}' <: \ \overline{{}^{s}N}'$ where ${}^{s}\Gamma'$ is the environment corresponding to the instantiation of the lemma for C_1, $\overline{{}^{s}N}_1$ are the upper bounds of type $\texttt{self } C_1\texttt{<}\overline{{}^{s}T}_1\texttt{>}$, and $\overline{{}^{s}N}'$ are the upper bounds of type $\texttt{self } C'\texttt{<}\overline{{}^{s}T}'\texttt{>}$.
The substitution of the $\overline{{}^{s}T}_1$ for the $\overline{X}_1$ in the $\overline{{}^{s}T}'$ results in $\overline{{}^{s}T}$ and that same substitution is performed in the upper bounds of class C'. As we are substituting subtypes of the upper bounds of each type variable, the resulting type is again a subtype of its upper bound. □

A.2.4.3 Proof of Lemma A.1.20 — Subclassing: Superclass Instantiation has Upper Bounds Without `lost`

We prove:

$$\left.\begin{array}{ll} 1. & \vdash P \ \ \mathsf{OK} \\ 2. & C\texttt{<}\overline{X}\texttt{>} \sqsubseteq C'\texttt{<}\overline{{}^{s}T}\texttt{>} \\ 3. & C \neq C' \\ 4. & \text{ClassBnds}\Big(\texttt{self } C'\texttt{<}\overline{{}^{s}T}\texttt{>}\Big) = \overline{{}^{s}N} \end{array}\right\} \Longrightarrow \ \texttt{lost} \notin \overline{{}^{s}N}$$

We prove this by an induction on the derivation of 2. (see Def. 2.2.1).

Case 1: sc1

From 1. we know that class C was checked to conform to Def. 2.3.14, which uses strict subtyping to check that the type arguments of the superclass are subtypes of the upper bounds. This ensures that the upper bounds of C' do not contain `lost`.

Case 2: sc2
Forbidden by 3.

Case 3: sc3
We have $C\texttt{<}\overline{X}\texttt{>} \sqsubseteq C_1\texttt{<}\overline{{}^sT}_1\texttt{>}$ and $C_1\texttt{<}\overline{X}_1\texttt{>} \sqsubseteq C'\texttt{<}\overline{{}^sT}'\texttt{>}$ giving $C\texttt{<}\overline{X}\texttt{>} \sqsubseteq C'\texttt{<}\overline{{}^sT}\texttt{>}$, with the substitution $\overline{{}^sT} = \overline{{}^sT}'\left[\overline{{}^sT}_1/\overline{X}_1\right]$.

We apply the induction hypothesis to derive that the upper bounds of `self` $C_1\texttt{<}\overline{{}^sT}_1\texttt{>}$ and `self` $C'\texttt{<}\overline{{}^sT}'\texttt{>}$ do not contain `lost`. The substitution of the $\overline{{}^sT}_1$ for the $\overline{X}_1$ in the $\overline{{}^sT}'$ results in $\overline{{}^sT}$ and we get that the upper bounds of `self` $C'\texttt{<}\overline{{}^sT}\texttt{>}$ do not contain `lost` directly. □

A.2.4.4 Proof of Lemma A.1.21 — Subtyping and `self`

The first part is

$$\left.\begin{array}{ll} 1. & \vdash P \ \ \mathsf{OK} \\ 2. & {}^s\Gamma \ \ \mathsf{OK} \\ 3. & {}^s\Gamma \vdash {}^sT <: \ {}^sT' \\ 4. & \texttt{self} \notin {}^sT \end{array}\right\} \Longrightarrow \texttt{self} \notin {}^sT'$$

We prove this by an induction on the derivation of 3. (see Def. 2.3.8).

Case 1: st1
From Corollary A.1.18 we know that `self` is not contained in the superclass instantiation. The viewpoint adaptation and substitution of the type arguments cannot introduce `self`, as ${}^sT$ does not contain `self`.

Case 2: st2
The ordering of ownership modifiers and type argument subtyping cannot introduce `self`.

Case 3: st3
If ${}^sT$ is a type variable, the supertype is either unchanged or the upper bound from the environment. The upper bound cannot contain `self` because of 2.

Case 4: st4
Transitivity does not change the types and can therefore not introduce `self` in the supertype. □

The second part is

$$\left.\begin{array}{ll} 1. & \vdash P \ \ \mathsf{OK} \\ 2. & {}^s\Gamma \ \ \mathsf{OK} \\ 3. & {}^s\Gamma \vdash {}^sT <: \ {}^sT' \\ 4. & \mathrm{om}({}^sT', {}^s\Gamma) = \texttt{self} \end{array}\right\} \Longrightarrow \mathrm{om}({}^sT, {}^s\Gamma) = \texttt{self}$$

`self` can only be the main modifier in a supertype if it was already the main modifier in the subtype. This directly follows from $u <:_u u'$. If ${}^sT$ is a type variable, then from 2. we know that the upper bound does not contain `self` and therefore 4. would not hold. If ${}^sT'$ is a type variable, then ${}^sT$ is the same type variable and neither can contain `self`. □

A.2.4.5 Proof of Lemma A.1.23 — dyn Preserves Subtyping

We prove:

$$\left.\begin{array}{ll} 1. & \vdash P \ \ \mathsf{OK} \\ 2. & h, {}^{r}\Gamma : {}^{s}\Gamma \ \ \mathsf{OK} \\ 3. & {}^{s}\Gamma \vdash {}^{s}T, {}^{s}T' \ \ \mathsf{OK} \\ 4. & {}^{s}\Gamma \vdash {}^{s}T <: {}^{s}T' \end{array}\right\} \Longrightarrow \begin{array}{rll} \exists \overline{o_\iota}, {}^{r}T. & I. & \mathrm{dyn}({}^{s}T, h, {}^{r}\Gamma, \overline{o_\iota}) = {}^{r}T \wedge \\ \exists \overline{o_\iota}', {}^{r}T'. & II. & \mathrm{dyn}\left({}^{s}T', h, {}^{r}\Gamma, \overline{o_\iota}'\right) = {}^{r}T' \wedge \\ & III. & h \vdash {}^{r}T <: {}^{r}T' \end{array}$$

From 1., 2., and 3. using Lemma A.1.9 we get $I.$ and $II.$ Note that the lengths of $\overline{o_\iota}$ and $\overline{o_\iota}'$ depend on the number of `lost` modifiers in the respective static types. Also, the choice of $\overline{o_\iota}'$ depends on the choice of $\overline{o_\iota}$, as consistent substitutions are chosen to ensure $III.$

If both ${}^{s}T$ and ${}^{s}T'$ are type variables, we know that they have to be the same (rule ST3 in Def. 2.3.8). Therefore, their dynamizations are also equal and $III.$ follows from the reflexivity of runtime subtyping.

If ${}^{s}T$ is a type variable and ${}^{s}T'$ is its upper bound, then from 2. we have that the runtime types are subtypes.

If ${}^{s}T$ is a type variable, but ${}^{s}T'$ is a non-variable type, we use the transitivity of both subtyping relations: ${}^{s}T$ is a subtype of its upper bound and that upper bound has to be a subtype of ${}^{s}T'$. On these two parts we can apply the current lemma and then use transitivity of runtime subtyping to arrive at $III.$

The interesting case is when both static types are non-variable types; we name their components as ${}^{s}T = u\ C\texttt{<}\overline{{}^{s}T}\texttt{>}$ and ${}^{s}T' = u'\ C'\texttt{<}\overline{{}^{s}T}'\texttt{>}$.

From 4. and the subtyping rules ST1 and ST2 (see Def. 2.3.8) we then have:

$$\begin{array}{ll} {}^{s}\Gamma \vdash u\ C\texttt{<}\overline{{}^{s}T}\texttt{>} <: u'\ C'\texttt{<}\overline{{}^{s}T}'\texttt{>} & C\texttt{<}\overline{X}\texttt{>} \sqsubseteq C'\texttt{<}\overline{{}^{s}T_1}\texttt{>} \\ u\ C\texttt{<}\overline{{}^{s}T}\texttt{>} \rhd \overline{{}^{s}T_1} = \overline{{}^{s}T_2} & u <:_u u' \\ \vdash \overline{{}^{s}T_2} <:_l \overline{{}^{s}T}' & \end{array}$$

From the definition of dyn (see Def. 2.2.14) and $I.$ we know for some ${}^{o}\iota$ and $\overline{{}^{r}T}$ where ${}^{r}T = {}^{o}\iota\ C\texttt{<}\overline{{}^{r}T}\texttt{>}$:

$$\mathrm{dyn}\left(u\ C\texttt{<}\overline{{}^{s}T}\texttt{>}, h, {}^{r}\Gamma, \overline{o_\iota}\right) = {}^{o}\iota\ C\texttt{<}\overline{{}^{r}T}\texttt{>}$$

From the definition of dyn and $II.$ we know for some ${}^{o}\iota'$ and $\overline{{}^{r}T}'$ where ${}^{r}T' = {}^{o}\iota'\ C'\texttt{<}\overline{{}^{r}T}'\texttt{>}$:

$$\mathrm{dyn}\left(u'\ C'\texttt{<}\overline{{}^{s}T}'\texttt{>}, h, {}^{r}\Gamma, \overline{o_\iota}'\right) = {}^{o}\iota'\ C'\texttt{<}\overline{{}^{r}T}'\texttt{>}$$

For $III.$ ($h \vdash {}^{o}\iota\ C\texttt{<}\overline{{}^{r}T}\texttt{>} <: {}^{o}\iota'\ C'\texttt{<}\overline{{}^{r}T}'\texttt{>}$), according to Def. 2.2.12, we have to show that there exist ι and $\overline{o_\iota}''$ such that:

$$\begin{array}{ll} C\texttt{<}\overline{X}\texttt{>} \sqsubseteq C'\texttt{<}\overline{{}^{s}T}_1\texttt{>} & {}^{o}\iota' \in \{{}^{o}\iota, \texttt{any}_a\} \\ \mathrm{sdyn}\left(\overline{{}^{s}T}_1, h, \iota, {}^{o}\iota\ C\texttt{<}\overline{{}^{r}T}\texttt{>}, \overline{o_\iota}''\right) = \overline{{}^{r}T}' & \end{array}$$

We already obtained the subclassing relationship from the subtype relation 4. Subclassing between two classes is unique, so the type arguments to C' are the same $\overline{{}^{s}T}_1$.

We know that $u <:_u u'$. Therefore, the address substituted for u' is either the same as the one used for u (in the cases where they are equal, `self` and `peer`, or `lost` when we can use a suitable address) or we have that $u' = \texttt{any}$ and we substitute it with ${}^{o}\iota' = \texttt{any}_a$.

From Lemma A.1.14 we know that the type variables that are used in $\overline{{}^{s}T}_1$ are a subset of the $\overline{X}$. This ensures that all type variables can be substituted by the type arguments $\overline{{}^{r}T}$.

So the last thing that needs to be shown is that we can find an address ι and substitutions $\overline{\circ\iota}''$ such that $\text{sdyn}\left(\overline{{}^{s}T}_1, h, \iota, {}^{\circ}\iota\ C\texttt{<}\overline{{}^{r}T}\texttt{>}, \overline{\circ\iota}''\right) = \overline{{}^{r}T}'$. Recall that $u\ C\texttt{<}\overline{{}^{s}T}\texttt{>} \triangleright \overline{{}^{s}T}_1 = \overline{{}^{s}T}_2$ and $\vdash \overline{{}^{s}T}_2 <:_l \overline{{}^{s}T}'$.

We know that $\text{dyn}\left(\overline{{}^{s}T}', h, {}^{r}\Gamma, \overline{\circ\iota}_3\right) = \overline{{}^{r}T}'$ from $II.$ and Lemma A.1.28, where $\overline{\circ\iota}_3$ is equal to $\overline{\circ\iota}'$, possibly without the first element (if $u' = \texttt{lost}$ the first element from $\overline{\circ\iota}'$ is used to substitute it and the remaining elements are used for substitutions in the type arguments).

Recall that $\vdash \overline{{}^{s}T}_2 <:_l \overline{{}^{s}T}'$ can either leave the types unchanged or change arbitrary ownership modifiers to `lost`. For ownership modifiers that are changed to `lost`, we can choose a substitution $\overline{\circ\iota}_3$ that uses the same address as required. In the following we can therefore assume that $\overline{{}^{s}T}_2 = \overline{{}^{s}T}'$

We continue with an analysis of $u\ C\texttt{<}\overline{{}^{s}T}\texttt{>} \triangleright \overline{{}^{s}T}_1 = \overline{{}^{s}T}_2$:

- `self` modifiers in $\overline{{}^{s}T}_1$ cannot happen because of Corollary A.1.18.
- `peer` modifiers in $\overline{{}^{s}T}_1$ are replaced by sdyn with ${}^{\circ}\iota$. In $\overline{{}^{s}T}_2$, these modifiers were either replaced by u (if u is `peer` or `rep`) or becomes `lost` (if u is `lost` or `any`). From $I.$ we know that u is replaced by ${}^{\circ}\iota$, so we get the same replacement.
- `rep` modifiers in $\overline{{}^{s}T}_1$, will be `rep` modifiers in $\overline{{}^{s}T}'$, if u is `self`. Then for sdyn we choose $\iota = {}^{r}\Gamma(\texttt{this})$ and dyn will also substitute the corresponding modifiers by ${}^{r}\Gamma(\texttt{this})$.
- `any` modifiers in $\overline{{}^{s}T}_1$ are not changed by viewpoint adaptation and are replaced by the $\texttt{any}_a$ address by both dynamizations.
- Type variables in $\overline{{}^{s}T}_1$ are replaced by the corresponding type arguments in $\overline{{}^{s}T}$. sdyn replaces these type variables with the arguments in $\overline{{}^{r}T}$. In dyn these type variables are replaced by the type arguments in ${}^{s}T$ and are therefore replaced with the same dynamizations.
- For any ownership modifier that is changed to `lost` by viewpoint adaptation we can select the correct substitution in $\overline{\circ\iota}''$ and $\overline{\circ\iota}_3$.

In conclusion, in all cases we can choose appropriate substitutions to ensure that $III.$ holds. □

A.2.4.6 Proof of Lemma A.1.22 — Static Type Assignment to Values Preserves Subtyping

We prove:

$$\left.\begin{array}{ll} 1. & \vdash P\ \ \mathsf{OK} \\ 2. & h, {}^{r}\Gamma : {}^{s}\Gamma\ \ \mathsf{OK} \\ 3. & {}^{s}\Gamma \vdash {}^{s}T, {}^{s}T'\ \ \mathsf{OK} \\ 4. & {}^{s}\Gamma \vdash {}^{s}T <:\ {}^{s}T' \\ 5. & h, {}^{r}\Gamma \vdash v :\ {}^{s}T \end{array}\right\} \Longrightarrow\ h, {}^{r}\Gamma \vdash v :\ {}^{s}T'$$

We split $h, {}^{r}\Gamma \vdash v :\ {}^{s}T'$ into the two parts $\exists \overline{\circ\iota}, {}^{r}T.\ \text{dyn}({}^{s}T', h, {}^{r}\Gamma, \overline{\circ\iota}) = {}^{r}T\ \wedge\ h \vdash v :\ {}^{r}T$ and ${}^{s}T' = \texttt{self \_<\_>} \Longrightarrow v = {}^{r}\Gamma(\texttt{this})$ according to Def. 2.2.15.

The first part follows from Lemma A.1.23, which we can instantiate using 1., 2., 3., and 4, combined with 5. which gives us the information about the type of v.

The second part follows from 5. and Lemma A.1.21. The supertype can only contain `self`, if also the subtype contains self. Therefore, from 4. we know that ${}^{r}\Gamma(\texttt{this}) = v$ holds for supertype and subtype if the supertype contains `self`. □

A.2.5 Runtime Behavior

A.2.5.1 Proof of Lemma A.1.26 — Equivalence of sdyn and dyn

We prove:

$$\left.\begin{array}{ll} 1. & \vdash P \ \ \mathsf{OK} \\ 2. & h, {}^{r}\Gamma : {}^{s}\Gamma \ \ \mathsf{OK} \\ 3. & {}^{s}\Gamma \vdash u \ C\texttt{<}\overline{{}^{s}T}\texttt{>} \ \ \mathsf{strictly\ OK} \\ 4. & \mathrm{dyn}\Big(u \ C\texttt{<}\overline{{}^{s}T}\texttt{>}, h, {}^{r}\Gamma, \emptyset\Big) \ = {}^{o}\iota \ C\texttt{<}\overline{{}^{r}T}\texttt{>} \\ 5. & u \ C\texttt{<}\overline{{}^{s}T}\texttt{>} \triangleright {}^{s}T \ = \ {}^{s}T' \\ 6. & \texttt{lost} \notin {}^{s}T' \\ 7. & \mathrm{sdyn}\Big({}^{s}T, h, \iota, {}^{o}\iota \ C\texttt{<}\overline{{}^{r}T}\texttt{>}, \emptyset\Big) \ = {}^{r}T \end{array}\right\} \Longrightarrow \ I. \quad \mathrm{dyn}({}^{s}T', h, {}^{r}\Gamma, \emptyset) \ = {}^{r}T$$

We prove this lemma by an induction on the shape of ${}^{s}T$. Cases 1, 2, and 3 are the base cases and case 4 is the induction step.

Case 1: ${}^{s}T = X_i$ a class type variable of class C

If ${}^{s}T$ is one of the class type variables of class C, we know from the definition of 7. that ${}^{r}T$ is ${}^{r}T_i$, the i-th type argument of $\overline{{}^{r}T}$.

The viewpoint adaptation 5. substitutes X_i by ${}^{s}T_i$, the i-th type argument of $\overline{{}^{s}T}$. From the definition of 4. we know that the dynamization $I.$ results in the same ${}^{r}T_i$.

Case 2: ${}^{s}T = X_i$ a method type variable

${}^{s}T$ cannot be a method type variable because 7. would not be defined.

Case 3: ${}^{s}T = u' \ C'\texttt{<>}$

If ${}^{s}T$ is a non-generic type, then we perform a case distinction for the result of 5. Let us call ${}^{s}T' = u'' \ C'\texttt{<>}$ where $u \triangleright u' \ = \ u''$.

We can distinguish 5 cases:

- $u = \texttt{self}$ and $u = \texttt{lost}$: forbidden by 3.
- $u = \texttt{peer}$ and $u' = \texttt{peer}$, resulting in $u'' = \texttt{peer}$: From 4. we know that `peer` is dynamized to ${}^{o}\iota$ and therefore $I.$ uses the same ${}^{o}\iota$.
- $u = \texttt{rep}$ and $u' = \texttt{peer}$, resulting in $u'' = \texttt{rep}$: From 4. we know that `rep` is dynamized to ${}^{o}\iota$ and therefore $I.$ uses the same ${}^{o}\iota$.
- $u' = \texttt{any}$, resulting in $u'' = \texttt{any}$: The `any` modifier in both 7. and $I.$ is substituted by $\texttt{any}_a$.
- All other cases result in $u'' = \texttt{lost}$, which is forbidden by 6.

Case 4: ${}^{s}T = u' \ C'\texttt{<}\overline{{}^{s}T}'\texttt{>}$

We apply the induction hypothesis to the $\overline{{}^{s}T}'$: Requirements 1. to 4. are unchanged, the viewpoint adaptation 5. is defined for the $\overline{{}^{s}T}'$ and we know that `lost` is not contained for the whole type and therefore also not for the type arguments; finally, 7. also applies to the type arguments as sdyn performs the same substitution on the type arguments.

We perform the same case analysis as in case 3 for the main modifier u' resulting in equality of the owner address.

Finally, we know that dyn is compositional from Lemma A.1.28 and can therefore combine the owner address and type arguments to deduce $I.$ □

A.2.5.2 Proof of Lemma A.1.25 — Runtime Meaning of Ownership Modifiers

We prove:

$$\begin{array}{l} \text{If } \ \mathrm{dyn}({}^{s}N, h, {}^{r}\Gamma, \overline{o_i}) = {}^{r}T \ \wedge \ h \vdash \iota : {}^{r}T \ \text{ then} \\ \quad \mathrm{om}({}^{s}N) = \texttt{self} \quad \Rightarrow \quad \mathrm{owner}(h, \iota) = \mathrm{owner}(h, {}^{r}\Gamma(\texttt{this})) \\ \quad \mathrm{om}({}^{s}N) = \texttt{peer} \quad \Rightarrow \quad \mathrm{owner}(h, \iota) = \mathrm{owner}(h, {}^{r}\Gamma(\texttt{this})) \\ \quad \mathrm{om}({}^{s}N) = \texttt{rep} \quad \Rightarrow \quad \mathrm{owner}(h, \iota) = {}^{r}\Gamma(\texttt{this}) \end{array}$$

The proof is a case analysis and the application of the definition of dyn (see Def. 2.2.14):

Case 1: $\mathrm{om}({}^{s}N) = \texttt{self}$
$\mathrm{dyn}({}^{s}N, h, {}^{r}\Gamma, \overline{o_i}) = {}^{r}T$ replaces `self` by $\mathrm{owner}(h, {}^{r}\Gamma(\texttt{this}))$, that is, the owner of ${}^{r}\Gamma(\texttt{this})$.

Case 2: $\mathrm{om}({}^{s}N) = \texttt{peer}$
$\mathrm{dyn}({}^{s}N, h, {}^{r}\Gamma, \overline{o_i}) = {}^{r}T$ replaces `peer` by $\mathrm{owner}(h, {}^{r}\Gamma(\texttt{this}))$, that is, the owner of ${}^{r}\Gamma(\texttt{this})$.

Case 3: $\mathrm{om}({}^{s}N) = \texttt{rep}$
$\mathrm{dyn}({}^{s}N, h, {}^{r}\Gamma, \overline{o_i}) = {}^{r}T$ replaces `rep` by ${}^{r}\Gamma(\texttt{this})$. □

Note that we do not need to require a well-formed heap, because all three cases are consequences of the definition of dyn.

A.2.6 Progress

We omit a proof of progress since this property is not affected by adding ownership to a Java-like language. The basic proof can easily be adapted from FGJ; extensions to include field updates and casts have also been done before, for example in ClassicJava [80] and MiddleWeightJava [20].

An investigation of the operational semantics shows that there are only three expressions that deal with ownership at runtime. We present an informal argument why they preserve progress:

Case 3: $e = \texttt{new}\ {}^{s}T\texttt{()}$

The operational semantics creates the runtime type from the static type using dyn without a substitution for `lost`. The type rules ensure that the static type is strictly well formed and forbid that the type includes `lost`. Therefore, the dynamization of the static type is defined.

A strictly well-formed static type also does not contain `lost` in its upper bounds. Therefore, in a method call we can be sure that the upper bounds of the class type variables of the receiver object can be dynamized without a substitution for `lost` in $h, {}^{r}\Gamma' : {}^{s}\Gamma'$ OK.

Case 6: $e = e_0\ \texttt{.}\ m\ \texttt{<}\overline{{}^{s}T}\texttt{>(}\ e_1\ \texttt{)}$

The method type arguments are dynamized with an empty substitution for `lost`. Again, the type rules ensure that the static type is strictly well formed and does not contain `lost`.

The type rules ensure that the viewpoint-adapted upper bounds of the method do not contain `lost` and therefore also the un-adapted upper bounds do not contain `lost`. This allows us to construct $h, {}^{r}\Gamma' : {}^{s}\Gamma'$ OK with an empty substitution for `lost`.

Case 7: $e = \texttt{(}{}^{s}T_0\texttt{)}\ e_0$

The type rules ensure well formedness of the cast type. The static type-to-value assignment judgment finds a substitution that substitutes all occurrences of `lost`.

Additional lemmas like the following could be proved easily:

Lemma A.2.1 (Evaluation Results in a Valid Address or $\texttt{null}_a$*)*

$$\left.\begin{array}{l} {}^{r}T \vdash h, e \rightsquigarrow h', v \\ v = \iota \end{array}\right\} \Longrightarrow \iota \in \mathrm{dom}(h')$$

A.2.7 Method Type Variables and Recursion

For the proofs of the adaptation lemmas we can assume that the type variables in ${}^{r}T$ and ${}^{r}T'$ are disjoint, that is, that the type variables that can appear in ${}^{s}N$ are different from the type variables that can appear in ${}^{s}T$. Consistent renaming of method type variables in method signature look-ups are applied to avoid capturing from happening.

Consider the following recursive method call:

```
class C<Xc> {
  <Xm> void m(Xc p1, Xm p2) {
    rep C<peer D<Xm>> o = new ...;
    o.m<rep D<Xm>>(p1, p2);
  }
}
```

In the method call `o.m` consider the type for parameter `p1`. If the method signature does not avoid capturing, we would perform

$$(\texttt{rep C<peer D<Xm>>} \triangleright \texttt{Xc})[\texttt{rep D<Xm>/Xm}] = \texttt{peer D<rep D<Xm>>}$$

which would substitute the `Xm` from the receiver type by the method type argument of the recursive call, which is not the intended interpretation. With capture-avoiding renaming, we derive `peer D<Xm>` as the type for parameter `p1`, correctly leaving the substitution of the `Xm` to the calling environment.

Putting GUT aside for a moment, consider the following Java 5 program:

```
class D<Z> {}
class E {}

class C<Xc> {
    <Xm> void m(Xc pxc, Xm pxm) {
        C< D<Xm> > o = new C< D<Xm> >();
        o.m( new D<Xm>(), new E() );     // A
        o.m( new D<E>(), new E() );      // B
    }
}
```

Looking at Featherweight Generic Java FGJ [99], we derive that line "A" should be valid and line "B" should be forbidden. The method lookup function *mtype* has to perform a capture-avoiding substitution to determine the signature of "m", that is

$$mtype(\texttt{m}, \texttt{C<D<Xm>>}) = \texttt{< Xm' >}\ (\texttt{D<Xm>}, \texttt{Xm'}) \rightarrow \texttt{void}$$

Then substituting the method type argument `E` results in a method that expects as subtype of `D<Xm>` as first argument and of `E` as second argument.

However, trying the example with different versions of javac gives a type checking error for line "A" and allows line "B".

Now consider this Java 5 program:

```
class D<Z> {
    Z f;
}
class E {}
class F {}

class C<Xc> {
    Xc dirty;

    <Xm> D<Xm> m(Xc pxc, Xm pxm, boolean rec) {
        System.out.println("this:␣" + this + "␣pxc:␣" + pxc + "␣pxm:␣" + pxm );

        dirty = pxc;
        C< D<Xm> > cdm = new C< D<Xm> >();
        if( rec ) {
            // A
            // does not type-check in javac
            // type checks in gcj and ecj, and runs without producing exceptions
            D<Xm> dm = new D<Xm>();
            dm.f = pxm;
            cdm.m( dm, new E(), false );

            // B
            // does not type-check in gcj and ecj
            // type checks in javac, and produces a cast exception
            D<E> de = new D<E>();
            de.f = new E();
            cdm.m( de, new E(), false );
        }
        return cdm.dirty;
    }
}

public class MTVDemo {
    public static void main( String[] args ) {
        C< D<F> > cdf = new C< D<F> >();

        D<F> df;
        df = cdf.m( new D<F>(), new F(), true );

        // if B is compiled with javac, this statement produces:
        // Exception in thread "main" java.lang.ClassCastException:
        // E cannot be cast to F
        F x = df.f;

        System.out.println("x:␣" + x);
    }
}
```

This program does not use casts and compiles in javac without a warning, but generates a `ClassCastException` when executed. The example exploits a heap pollution that was not caught by javac.

We observed this behavior with JDK versions 1.5.0_15, 1.6.0_11 and OpenJDK 1.7 build 57. On the other hand, gcj (GNU compiler for Java) version 4.3.2 and ecj (Eclipse compiler for

Java) version 3.3.1 allow line "A" and forbid line "B".

We could not find a discussion of how exactly substitutions are performed and how capturing is avoided in the Java Language Specification version 3 [86]. We filed a bug with javac (see `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6838943`).

Appendix B

Ott Formalization

B.1 Complete Grammar

We use the tool Ott [176] to formalize Generic Universe Types and used the generated LaTeX code throughout this document. The definition consists of 140 rules and 361 rule clauses and is disambiguated enough to also produce Isabelle [148] definitions.

We define the following Ott meta-variables:

k	index variable for class type variables and arguments
l	index variable for method type variables and arguments
q	index variable for method parameters and arguments
i, j	index variables as arbitrary elements
n	index variable as upper limit
f	field identifier
mid	method identifier
pid	parameter identifier
X	type variable identifier
Cid	derived class identifier
$RAId$	raw address identifier

We use the following naming convention: "identifier" is used for some undetermined set, e.g., the parameter identifiers pid are some unspecified set of values. "names" also contain some special values, e.g., the parameter names x also contains `this`. If there is no difference, the "Id" is left off in the short version, e.g., X is used for type variable identifiers and there are no special values.

Variables that are used in the conclusions, usually use primed names. Additional variables in the requirements might use numbered names.

Note that sometimes Ott required the addition of primes to keep differently indexed variables separate; for example, in TR_CALL there are $\overline{{}^{s}T_l}$ and $\overline{{}^{s}T'_q}$ and we need to use a prime to keep Ott happy. Also note that the index variable (l and q in the above example) appears next to the indexed element and again next to the overbar.

The option versions are only used to make un-definedness more explicit at the moment. However, in Isabelle we will need this distinction. The only place where we actually use un-definedness is in the rule for method overriding. We also used different symbols if a comparison involves an optional element, for example, from the method overriding rule, when we write $\mathrm{MSig}(C', m) =_o ms'_o$ the result of method signature look-up is an optional method signature and we assign the result to an optional type; when we write $\mathrm{MSig}(C, m) =_o ms$ the result of method signature look-up is still an optional method signature, but we also require that it actually is a concrete method signature.

terminals	::=	
	| `class`	keyword: class declaration
	| `extends`	keyword: super type declaration
	| `new`	keyword: object creation
	| `this`	keyword: current object
	| `null`	keyword: null value
	| `pure`	keyword: pure method
	| `impure`	keyword: non-pure method
	| `<`	syntax: start generics
	| `>`	syntax: end generics
	| `{`	syntax: start block
	| `}`	syntax: end block
	| `(`	syntax: start parameters
	| `)`	syntax: end parameters
	| `;`	syntax: separator
	| `.`	syntax: selector
	| `=`	syntax: assignment
	| `Object`	name of root class
	| $\in$	containment judgement
	| $\notin$	non-containment jdugement
	| $\vdash$	single element judgement
	| $\vdash$	multiple element judgement
	| :	separator
	| $:_s$	strict separator
	| $\mapsto$	maps-to
	| OK	well-formedness judgement
	| strictly OK	strict well-formedness judgement
	| pure	purity judgement
	| strictly pure	strict purity judgement

	enc	encapsulation judgement
	prg OK	programmer well-formedness judgement
	$\sqsubseteq$	subclassing
	$<:$	subtyping of single type
	$<:_u$	ordering of ownership modifiers
	$<:_l$	argument ordering
	$<:_s$	strict ordering
	$<:_i$	invariant ordering
	$=$	alias
	$\neq$	not alias
	$=$	multiple alias
	$=_o$	option alias
	$\neq_o$	option not alias
	$=_o$	optional, multiple alias
	$\subseteq$	subset relation
	$\cup$	append two lists
	$\vee$	logical or
	$\wedge$	logical and
	AND	top-level logical and
	$\Longrightarrow$	logical implication
	$\texttt{null}_a$	special null address
	$\texttt{any}_a$	special any address
	$\texttt{root}_a$	special root address
	$\texttt{lost}_a$	special lost address

formula	::=				formulas
	|	otherwise			none of the previous rules applied
	|	*judgement*			judgement
	|	$formula_1$	, .. ,	$formula_k$	sequence

	$(formula)$		bracketed
	$!formula$		negation
	$formula \ \vee \ formula'$		logical or
	$formula \ \wedge \ formula'$		logical and
	$formula \quad formula'$		top-level logical and
	$formula \implies formula'$		implies
	$\overline{^{s}fml}$		static formulas
	$\overline{^{r}fml}$		runtime formulas

C	::=			class name
	|	Cid		derived class identifier
	|	`Object`		name of base class
	|	`_`	M	some class name
	|	ClassOf(^{s}N)	M	class of a static non-variable type
	|	ClassOf(^{r}T)	M	class of a runtime type

u	::=			ownership modifier
	|	`self`		current object
	|	`peer`		same context
	|	`rep`		representation context
	|	`any`		any context
	|	`lost`		lost context
	|	`_`	M	some ownership modifier
	|	om(^{s}N)	M	ownership modifier of non-variable type
	|	om(^{s}T, $^{s}\Gamma$)	M	ownership modifier of static type

$\overline{u}$	::=			ownership modifiers
	|	u_1 , .. , u_n		ownership modifier list
	|	$\{\overline{u}\}$	M	notation

${}^{s}CT$	::=			class type
	|	$C\texttt{<}\overline{{}^{s}T}\texttt{>}$		generic class type
${}^{s}T$	::=			static type
	|	${}^{s}N$		non-variable type
	|	X		type variable
	|	_	M	some type
	|	${}^{s}T\left[\overline{{}^{s}T}/\overline{X}\right]$	M	substitution of $\overline{{}^{s}T}$ for $\overline{X}$ in ${}^{s}T$
$\overline{{}^{s}T}$	::=			static types
	|	${}^{s}T_1$, .. , ${}^{s}T_n$		static type list
	|	$\overline{{}^{s}T}_1$, $\overline{{}^{s}T}_2$		two static type lists
	|	$\overline{{}^{s}N}$		non-variable type list
	|	$\overline{X}$		variable type list
	|	$\emptyset$		no static types
	|	_	M	some static types
	|	$\overline{{}^{s}T}'\left[\overline{{}^{s}T}/\overline{X}\right]$	M	substitution of $\overline{{}^{s}T}$ for $\overline{X}$ in each type in $\overline{{}^{s}T}'$
${}^{s}T_o$	::=			static type option
	|	${}^{s}T$		lifted static type
	|	${}^{s}N_o$		non-variable type option
	|	${}^{s}\Gamma(x)$	M	look up parameter type
$\overline{{}^{s}T}_o$	::=			static types option
	|	$\overline{{}^{s}T}$		lifted static types
${}^{s}N$	::=			non-variable type
	|	$u\ C\texttt{<}\overline{{}^{s}T}\texttt{>}$		definition
	|	_	M	some non-variable type

$\overline{{}^{s}N}$	::=			non-variable types
	|	${}^{s}N$		one non-variable type
	|	$\overline{{}^{s}N}_1$, .. , $\overline{{}^{s}N}_n$		non-variable type list
	|	$\emptyset$		no non-variable types
	|	_	M	some non-variable types
${}^{s}N_o$	::=			non-variable type option
	|	${}^{s}N$		lifted non-variable type
	|	${}^{s}\Gamma(X)$	M	look up upper bound of type variable
$\overline{{}^{s}N}_o$	::=			non-variable types option
	|	$\overline{{}^{s}N}$		lifted non-variable types
$\overline{X}$	::=			type variables
	|	X		one type variable
	|	$\overline{X}_1$, .. , $\overline{X}_n$		type variable list
	|	$\emptyset$		no type variables
	|	_	M	some type variables
	|	$\mathrm{free}\left(\overline{{}^{s}T}\right)$	M	type variables in $\overline{{}^{s}T}$
$\overline{X}_o$	::=			type variables option
	|	$\overline{X}$		lifted type variables
P	::=			program
	|	$\overline{Cls}$, C, e		
Cls	::=			class declaration
	|	`class` $ClsHd$ `{` $\overline{fd}$ $\overline{md}$ `}`		class declaration
	|	`class Object {}`		declaration of base class

$\overline{Cls}$	::=			class declarations
	|	Cls_1 .. Cls_n		class declaration list
$ClsHd$	::=			class header
	|	Cid<$\overline{TP}$> `extends` C<$\overline{{}^sT}$>		generic class header declaration
$\overline{TP}$	::=			type parameters
	|	X `extends` ${}^sN$		type variable X has upper bound ${}^sN$
	|	$\overline{TP}_1$, .. , $\overline{TP}_k$		type parameter list
	|	_	M	some unspecified type parameters
$\overline{fd}$	::=			field declarations
	|	${}^sT$ f `;`		type ${}^sT$ and field name f
	|	$\overline{fd}_1$ .. $\overline{fd}_n$		field declaration list
	|	_	M	some field declarations
$\overline{f}$	::=			list of field identifiers
	|	f_1 .. f_n		field identifier list
	|	fields(C)	M	recursive fields look-up
e	::=			expression
	|	`null`		null expression
	|	x		variable read
	|	`new` ${}^sT$`()`		object construction
	|	e`.`f		field read
	|	e_0`.`f `=` e_1		field write
	|	e_0 `.` m`<`$\overline{{}^sT}$`>(`$\overline{e}$`)`		method call
	|	`(`${}^sT$`)` e		cast

e_o	::=			expression option
	|	e		lifted expression
$\overline{e}$	::=			expressions
	|	e_1 , .. , e_k		list of expressions
	|	$\emptyset$		empty list
md	::=			method declaration
	|	ms `{` e `}`		method signature and method body
$\overline{md}$	::=			method declarations
	|	md		method declaration
	|	$\overline{md}_1$ .. $\overline{md}_n$		method declaration list
	|	_	M	some method declarations
ms	::=			method signature
	|	p `<` $\overline{TP}$ `>` ${}^{s}T$ m`(`$\overline{mpd}$`)`		method signature definition
	|	$ms[\overline{{}^{s}T}/\overline{X}]$	M	substitution of $\overline{{}^{s}T}$ for $\overline{X}$ in ms
ms_o	::=			method signature option
	|	ms		lifted method signature
	|	*None*		no method signature defined
p	::=			method purity modifiers
	|	`pure`		side-effect free
	|	`impure`		side effects possible
	|	_	M	some purity modifier
m	::=			method name

	|	mid		method identifier
	|	MName(ms)	M	extract method name from signature
$\overline{mpd}$	::=			method parameter declarations
	|	${}^{s}T\ pid$		type and parameter name
	|	$\overline{mpd}_1\ ,\ ..\ ,\ \overline{mpd}_q$		list
	|	_	M	some method parameter declarations
x	::=			parameter name
	|	pid		parameter identifier
	|	`this`		name of current object
${}^{s}\Gamma$	::=			static environment
	|	$\{{}^{s}\gamma\ ;\ {}^{s}\delta\}$		composition
${}^{s}\gamma_e$	::=			static type environment entry
	|	$X \mapsto {}^{s}N$		type variable X has upper bound ${}^{s}N$
${}^{s}\gamma$	::=			static type environment
	|	${}^{s}\gamma_{e1}\ ,\ ..\ ,\ {}^{s}\gamma_{en}$		mapping list
	|	$\emptyset$		empty type environment
${}^{s}\delta_p$	::=			static variable parameter environment
	|	$pid \mapsto {}^{s}T$		variable pid has static type ${}^{s}T$
${}^{s}\delta_t$	::=			static variable environment for `this`
	|	`this` $\mapsto {}^{s}N$		variable `this` has static type ${}^{s}N$
${}^{s}\delta$	::=			static variable environment

	$^{s}\delta_{t}$	mapping for `this`
	$^{s}\delta_{t}, \_$	mapping for `this` and some others
	$^{s}\delta_{t}$, $^{s}\delta_{p_1}$, .., $^{s}\delta_{p_q}$	mappings list
$\overline{^{s}fml}$	::=	static formulas
	$^{s}T = {}^{s}T'$	type alias
	$^{s}T_{o} =_{o} {}^{s}T$	type option alias
	$^{s}T \neq {}^{s}T'$	type not alias
	$\overline{^{s}T} = \overline{^{s}T}'$	types alias
	$\overline{X} \subseteq \overline{X}'$	type variables $\overline{X}$ subset of $\overline{X}'$
	$^{s}T \in \overline{^{s}T}$	type ^{s}T contained in list $\overline{^{s}T}$
	$C = C'$	class name alias
	$C \neq C'$	class name not alias
	$^{s}\Gamma = {}^{s}\Gamma'$	static environment alias
	$ms = ms'$	method signature alias
	$ms_{o} =_{o} ms'_{o}$	method signature option alias
	$m = m'$	method name alias
	$m \neq m'$	method name not alias
	$e = e'$	expression alias
	$Cls \in P$	class definition in program
	`class` C`<`$\overline{TP}$`>`$\ldots \in P$	partial class definition in program
	$X \in {}^{s}\Gamma$	type variable in static environment
	$x \in {}^{s}\Gamma$	parameter in static environment
	$u = u'$	ownership modifier alias
	$u \neq u'$	ownership modifier not alias
	$\overline{u} \in \overline{^{s}T}$	ownership modifiers in types (ignores Xs)
	$\overline{u} \notin \overline{^{s}T}$	ownership modifiers not in types (ignores Xs)
	$u \in \overline{u}$	ownership modifier in set of ownership modifiers
	$p = p'$	purity alias

	$\forall f \in \overline{f}.\ formula$		for all f in $\overline{f}$ holds *formula*
	$\forall i \in \{1 \ldots k\}.\ formula$		for all i in $\{1 \ldots k\}$ holds *formula*
	$\exists\, \overline{^sT}.\ formula$		exists $\overline{^sT}$ such that *formula*
	$\forall\, \overline{^sT}.\ formula$		for all $\overline{^sT}$ holds *formula*
	$\forall^s CT.\ formula$		for all $^s CT$ holds *formula*
	$\forall C, C'.\ formula$		for all C and C' holds *formula*
ι	::=		address identifier
	RAId		raw address identifier
	$^r\Gamma(\texttt{this})$	M	currently active object look-up
	_	M	some address identifier
$\overline{\iota}$	::=		address identifiers
	$\iota_1\ ,\ ..\ ,\ \iota_n$		address identifier list
	$\emptyset$		empty list
	_	M	some address identifier list
	$\mathrm{dom}(h)$	M	domain of heap
v	::=		value
	ι		address identifier
	$\texttt{null}_a$		null value
	_	M	some value
v_o	::=		value option
	v		lifted value
	$h(\iota.f)$	M	field value look-up
	$^r\Gamma(x)$	M	argument value look-up
	$\overline{fv}(f)$	M	field value look-up

$\overline{v}$	::=			values
	|	v_1 , .. , v_n		value list
	|	$\emptyset$		empty list
$^{o}\iota$	::=			owner adress
	|	ι		address of the owner
	|	$\texttt{root}_a$		root owner
	|	$\texttt{any}_a$		special any address
	|	_	M	some owner address
$^{o}\iota_o$	::=			owner adress option
	|	$^{o}\iota$		lifted owner address
	|	owner(h, ι)	M	look up owner in heap
	|	$u\,[\sigma]$	M	substitute u using σ
$\overline{^{o}\iota}$	::=			owner addresses
	|	$^{o}\iota_1$, .. , $^{o}\iota_n$		owner address list
	|	$\overline{^{o}\iota}_1 \cup .. \cup \overline{^{o}\iota}_n$		union of owner address lists
	|	$\overline{\iota}$		list of address identifiers
	|	$\emptyset$		empty list
	|	$\overline{\_}$	M	some list of owner addresses
	|	$\{\overline{^{o}\iota}\}$		notation
$\overline{^{o}\iota}_o$	::=			owner addresses option
	|	$\overline{^{o}\iota}$		lifted owner addresses
	|	owners(h, ι)	M	look up transitive owners in heap
$\overline{\overline{^{o}\iota}}$	::=			list of owner address identifiers
	|	$\overline{^{o}\iota}_1$; .. ; $\overline{^{o}\iota}_n$		owner address identifiers list

σ_e	::=			runtime substitution
	|	$^o\iota \,/\, u$		substitute $^o\iota$ for u
	|	$\overline{^rT} \,/\, \overline{X}$		substitute $\overline{^rT}$ for $\overline{X}$
σ	::=			runtime substitutions
	|	σ_{e1} , .. , σ_{en}		list of substitutions
^rT	::=			runtime type
	|	$^o\iota$ C<$\overline{^rT}$>		definition
^rT_o	::=			runtime type option
	|	^rT		lifted runtime type
	|	$h(\iota)\downarrow_1$	M	look up type in heap
	|	$^r\Gamma(X)$	M	look up runtime type of type variable
$\overline{^rT}$	::=			runtime types
	|	$^rT_1$, .. , ^rT_n		runtime type list
	|	$\emptyset$		no runtime types
	|	_	M	some runtime types
$\overline{^rT}_o$	::=			runtime types option
	|	$\overline{^rT}$		lifted runtime types
	|	$\overline{^sT}\,[\sigma]$	M	apply substitutions σ to $\overline{^sT}$
$\overline{fv}$	::=			field values
	|	$f \mapsto v$		field f has value v
	|	$\overline{fv}_1$, .. , $\overline{fv}_n$		field value list
	|	_	M	some field values
	|	$h(\iota)\downarrow_2$	M	look up field values in heap

	|	$\overline{fv}[f \mapsto v]$	M	update existing field f to v
o	::=			object
	|	$\left({}^{r}T, \overline{fv}\right)$		runtime type ${}^{r}T$ and field values $\overline{fv}$
o_o	::=			object option
	|	o		lifted object
	|	$h(\iota)$	M	look up object in heap
he	::=			heap entry
	|	$(\iota \mapsto o)$		address ι maps to object o
h	::=			heap
	|	$\emptyset$		empty heap
	|	$h + he$		add he to h, overwriting existing mappings
${}^{r}\Gamma$	::=			runtime environment
	|	$\{{}^{r}\gamma \ ; \ {}^{r}\delta\}$		composition
${}^{r}\gamma$	::=			runtime type environment
	|	$X \mapsto {}^{r}T$		type variable X has runtime type ${}^{r}T$
	|	${}^{r}\gamma_1 \, , \, .. \, , \, {}^{r}\gamma_n$		list of mappings
	|	$\emptyset$		empty type environment
${}^{r}\delta_p$	::=			runtime variable environment parameter entry
	|	$pid \mapsto v$		variable pid has value v
${}^{r}\delta_t$	::=			runtime variable environment entry for `this`
	|	$\texttt{this} \mapsto \iota$		variable `this` has address ι

${}^{r}\delta$	::=		runtime variable environment
	|	${}^{r}\delta_t$	mapping for `this`
	|	${}^{r}\delta_t, \_$	mapping for `this` and some others
	|	${}^{r}\delta_t$, ${}^{r}\delta_{p_1}$, .. , ${}^{r}\delta_{p_q}$	mappings list
$\overline{{}^{r}fml}$	::=		runtime formulas
	|	$h = h'$	heap alias
	|	${}^{r}T = {}^{r}T'$	runtime type alias
	|	${}^{r}T_o =_o {}^{r}T'_o$	type option alias
	|	$\overline{{}^{r}T} = \overline{{}^{r}T}'$	runtime types alias
	|	$\overline{{}^{r}T}_o =_o \overline{{}^{r}T}'$	runtime types option alias
	|	${}^{r}T \in \overline{{}^{r}T}$	type ${}^{r}T$ contained in list $\overline{{}^{r}T}$
	|	$v = v'$	value alias
	|	$v \neq v'$	value not alias
	|	$v_o =_o v'_o$	value option alias
	|	$v_o \neq_o v'_o$	value option not alias
	|	$\iota \in \overline{\iota}$	address in addresses
	|	$\overline{\iota} \neq \overline{\iota}'$	addresses not aliased
	|	$\iota \notin \overline{\iota}$	address not in addresses
	|	${}^{o}\iota = {}^{o}\iota'$	owner address alias
	|	${}^{o}\iota_o =_o {}^{o}\iota_o{}'$	owner address option alias
	|	${}^{o}\iota \neq {}^{o}\iota'$	owner address not alias
	|	$\overline{{}^{o}\iota} = \overline{{}^{o}\iota}'$	owner addresses alias
	|	$\overline{\overline{{}^{o}\iota}} = \overline{\overline{{}^{o}\iota}}'$	owner addresses alias
	|	$\overline{{}^{o}\iota}_o =_o \overline{{}^{o}\iota}'_o$	owner addresses option alias
	|	${}^{o}\iota_o \in \overline{{}^{o}\iota}_o$	owner address in owner addresses
	|	$\overline{{}^{o}\iota}_o \subseteq \overline{{}^{o}\iota}'$	owners option $\overline{{}^{o}\iota}_o$ contained in $\overline{{}^{o}\iota}'$
	|	$o = o'$	object alias
	|	$o_o =_o o'$	object option alias

	$^{r}\Gamma = {}^{r}\Gamma'$	runtime environment alias
	$\overline{fv} = \overline{fv}'$	fields alias
	$X \in {}^{r}\Gamma$	type variable in runtime environment
	$x \in {}^{r}\Gamma$	parameter in runtime environment
	$f \in \mathrm{dom}\left(\overline{fv}\right)$	field identifier f contained in domain of $\overline{fv}$
	$\forall {}^{r}\overline{T}.\ formula$	for all $\overline{{}^{r}T}$ holds *formula*
	$\forall {}^{o}\iota \in \overline{{}^{o}\iota}.\ formula$	for all ${}^{o}\iota$ in $\overline{{}^{o}\iota}$ holds *formula*
	$\forall \iota \in \bar{\iota}, f \in \overline{fv}.\ formula$	for all ι in $\bar{\iota}$ and field f in $\overline{fv}$ holds *formula*
	$\forall \bar{\iota} \in \mathcal{P}(\overline{{}^{o}\iota}).\ formula$	for all $\bar{\iota}$ from $\overline{{}^{o}\iota}$ holds *formula*
	$\forall f \in \overline{fv}.\ formula$	for all f in $\overline{fv}$ holds *formula*

st_helpers ::=

	$\mathrm{FType}(C, f) =_o {}^{s}T_o$	look up field f in class C
	$\mathrm{FType}({}^{s}N, f) =_o {}^{s}T_o$	look up field f in type ${}^{s}N$
	$\mathrm{MSig}(C, m) =_o ms_o$	look up signature of method m in class C
	$\mathrm{MSig}\left({}^{s}N, m, \overline{{}^{s}T}\right) =_o ms_o$	m in ${}^{s}N$ with method type arguments $\overline{{}^{s}T}$ substituted
	$\mathrm{ClassDom}(C) =_o \overline{X}_o$	look up type variables of class C
	$\mathrm{ClassBnds}(C) =_o \overline{{}^{s}N}_o$	look up bounds of class C
	$\mathrm{ClassBnds}({}^{s}N) =_o \overline{{}^{s}N}_o$	look up bounds of type ${}^{s}N$

ucombdef ::=

	$u \rhd u' = u''$	combining two ownership modifiers
	$u \rhd {}^{s}T = {}^{s}T'$	ownership modifier - type combination
	$u \rhd \overline{{}^{s}T} = \overline{{}^{s}T}'$	ownership modifier - types combination

tcombdef ::=

	${}^{s}N \rhd {}^{s}T =_o {}^{s}T'_o$	type - type combination
	${}^{s}N \rhd \overline{{}^{s}T} =_o \overline{{}^{s}T}'_o$	type - types combination
	$\left({}^{s}N \rhd \overline{{}^{s}T}\right)\left[\overline{{}^{s}T}'/\overline{X}'\right] =_o \overline{{}^{s}T}''_o$	type - types combination and substitution

stsubxing	::=		
	|	$u <:_u u'$	ordering of ownership modifiers
	|	${}^s CT \sqsubseteq {}^s CT'$	subclassing
	|	${}^s\Gamma \vdash {}^s T <: {}^s T'$	static subtyping
	|	$\vdash {}^s T <:_l {}^s T'$	type argument subtyping
	|	${}^s\Gamma \vdash {}^s T <:_s {}^s T'$	strict static subtyping
	|	$\vdash \overline{{}^s T} <:_l \overline{{}^s T}'$	type argument subtypings
	|	${}^s\Gamma \vdash \overline{{}^s T} <: \overline{{}^s T}'$	static subtypings
	|	${}^s\Gamma \vdash \overline{{}^s T} <:_s \overline{{}^s T}'$	strict static subtypings
typerules	::=		
	|	${}^s\Gamma \vdash e : {}^s T$	expression typing
	|	${}^s\Gamma \vdash \overline{e} : \overline{{}^s T}$	expression typings
	|	${}^s\Gamma \vdash e :_s {}^s T$	strict expression typing
	|	${}^s\Gamma \vdash \overline{e} :_s \overline{{}^s T}$	strict expression typings
wfstatic	::=		
	|	${}^s\Gamma \vdash {}^s T$ OK	well-formed static type
	|	${}^s\Gamma \vdash \overline{{}^s T}$ OK	well-formed static types
	|	${}^s\Gamma \vdash {}^s T$ strictly OK	strictly well-formed static type
	|	${}^s\Gamma \vdash \overline{{}^s T}$ strictly OK	strictly well-formed static types
	|	*Cls* OK	well-formed class declaration
	|	${}^s\Gamma \vdash {}^s T\ f;$ OK	well-formed field declaration
	|	${}^s\Gamma \vdash \overline{fd}$ OK	well-formed field declarations
	|	${}^s\Gamma, C \vdash md$ OK	well-formed method declaration
	|	${}^s\Gamma, C \vdash \overline{md}$ OK	well-formed method declarations
	|	${}^s CT \vdash m$ OK	method overriding OK
	|	${}^s CT, C\texttt{<}\overline{{}^s T}, \overline{X}\texttt{>} \vdash m$ OK	method overriding OK auxiliary
	|	${}^s\Gamma$ OK	well-formed static environment

	|	$\vdash P$ OK	well-formed program
encapsulation	::=		
	|	${}^{s}\Gamma \vdash e$ enc	encapsulated expression
	|	${}^{s}\Gamma \vdash \overline{e}$ enc	encapsulated expressions
	|	${}^{s}\Gamma, C \vdash md$ enc	encapsulated method declaration
	|	${}^{s}\Gamma, C \vdash \overline{md}$ enc	encapsulated method declarations
	|	Cls enc	encapsulated class declaration
	|	$\vdash P$ enc	encapsulated program
purity	::=		
	|	${}^{s}\Gamma \vdash e$ pure	pure expression
	|	${}^{s}\Gamma \vdash e$ strictly pure	strictly pure expression
	|	${}^{s}\Gamma \vdash \overline{e}$ strictly pure	pure expressions
prgok	::=		
	|	${}^{s}\Gamma \vdash {}^{s}T$ prg OK	reasonable static type
	|	${}^{s}\Gamma, {}^{s}N'' \vdash {}^{s}T <: {}^{s}N, {}^{s}N'$	reasonable static type argument
	|	${}^{s}\Gamma, {}^{s}N'' \vdash \overline{{}^{s}T} <: \overline{{}^{s}N}, \overline{{}^{s}N}'$	reasonable static type arguments
	|	${}^{s}\Gamma \vdash \overline{{}^{s}T}$ prg OK	reasonable static types
	|	Cls prg OK	reasonable class declaration
	|	${}^{s}\Gamma \vdash {}^{s}T\ f;$ prg OK	reasonable field declaration
	|	${}^{s}\Gamma \vdash \overline{fd}$ prg OK	reasonable field declarations
	|	${}^{s}\Gamma, C \vdash md$ prg OK	reasonable method declaration
	|	${}^{s}\Gamma, C \vdash \overline{md}$ prg OK	reasonable method declarations
	|	$\vdash P$ prg OK	reasonable program
	|	${}^{s}\Gamma \vdash e$ prg OK	reasonable expression
	|	${}^{s}\Gamma \vdash \overline{e}$ prg OK	reasonable expressions

rt_helpers	::=		
	|	$h + o \;=\; (h', \iota)$	add object o to heap h resulting in heap h' and fresh address ι
	|	$h[\iota\,.\,f \texttt{ = } v] \;=_o\; h'$	field update in heap
	|	$\mathrm{FType}(h, \iota, f) \;=_o\; {}^{s}T_o$	look up type of field in heap
	|	$\mathrm{MSig}(h, \iota, m) \;=_o\; ms_o$	look up method signature of method m at ι
	|	$\mathrm{MBody}(C, m) \;=_o\; e_o$	look up most-concrete body of m in class C or a superclass
	|	$\mathrm{MBody}(h, \iota, m) \;=_o\; e_o$	look up most-concrete body of method m at ι
	|	$\mathrm{sdyn}\left(\overline{{}^{s}T}, h, \iota, {}^{r}T, \overline{o_\iota}\right) \;=_o\; \overline{{}^{r}T}$	simple dynamization of types $\overline{{}^{s}T}$
	|	$\mathrm{ClassBnds}(h, \iota, {}^{r}T, \overline{o_\iota}) \;=_o\; \overline{{}^{r}T}$	upper bounds of type ${}^{r}T$ from viewpoint ι
rtsubxing	::=		
	|	$h \vdash {}^{r}T <:\; {}^{r}T'$	type ${}^{r}T$ is a subtype of ${}^{r}T'$
	|	$h \vdash \overline{{}^{r}T} <:\; \overline{{}^{r}T}'$	runtime subtypings
	|	$h \vdash v_o :\; {}^{r}T$	runtime type ${}^{r}T$ assignable to value v_o
	|	$h, {}^{r}\Gamma \vdash v :\; {}^{s}T$	static type ${}^{s}T$ assignable to value v (relative to ${}^{r}\Gamma$)
	|	$h, {}^{r}\Gamma \vdash \overline{v} :\; \overline{{}^{s}T}$	static types $\overline{{}^{s}T}$ assignable to values $\overline{v}$ (relative to ${}^{r}\Gamma$)
	|	$h, \iota \vdash v_o :\; {}^{s}T$	static type ${}^{s}T$ assignable to value v_o (relative to ι)
	|	$\mathrm{dyn}({}^{s}T, h, {}^{r}\Gamma, \overline{o_\iota}) \;=_o\; {}^{r}T_o$	dynamization of static type (relative to ${}^{r}\Gamma$)
	|	$\mathrm{dyn}\left(\overline{{}^{s}T}, h, {}^{r}\Gamma, \overline{\overline{o_\iota}}\right) \;=_o\; \overline{{}^{r}T}_o$	dynamization of static types (relative to ${}^{r}\Gamma$)
	|	$h, {}^{r}\Gamma \vdash {}^{s}N, {}^{s}T; \left(\overline{{}^{s}T}/\overline{X}, \iota\right) \;=_o\; {}^{r}\Gamma'$	validate and create new viewpoint ${}^{r}\Gamma'$
semantics	::=		
	|	${}^{r}\Gamma \vdash h, e \;\rightsquigarrow\; h', v$	big-step operational semantics
	|	${}^{r}\Gamma \vdash h, \overline{e} \;\rightsquigarrow\; h', \overline{v}$	sequential big-step operational semantics
	|	$\vdash P \;\rightsquigarrow\; h, v$	big-step operational semantics of a program
wfruntime	::=		
	|	$h,\ \iota \vdash {}^{r}T_o$ strictly OK	strictly well-formed runtime type ${}^{r}T_o$
	|	$h,\ \iota \vdash \overline{{}^{r}T}$ strictly OK	strictly well-formed runtime types

|	$h,\ \bar{\iota} \vdash \overline{{}^{r}T}$ strictly OK	strictly well-formed runtime types
|	$h \vdash \iota$ OK	well-formed address
|	h OK	well-formed heap
|	$h, {}^{r}\Gamma : {}^{s}\Gamma$ OK	runtime and static environments correspond

B.2 Complete Definitions

$\boxed{\text{FType}(C, f) =_o {}^s T_o}$ look up field f in class C

$$\frac{\texttt{class } \mathit{Cid}\texttt{<\_> extends \_<\_> \{ \_ } {}^s T\ f\texttt{; \_ \_ \}} \in P}{\text{FType}(\mathit{Cid}, f) =_o {}^s T}\ \text{SFTC\_\_DEF}$$

$\boxed{\text{FType}({}^s N, f) =_o {}^s T_o}$ look up field f in type ${}^s N$

$$\frac{\begin{array}{l}\text{FType}(\text{ClassOf}({}^s N), f) =_o {}^s T_1 \\ {}^s N \rhd {}^s T_1 =_o {}^s T\end{array}}{\text{FType}({}^s N, f) =_o {}^s T}\ \text{SFTN\_\_DEF}$$

$\boxed{\text{MSig}(C, m) =_o ms_o}$ look up signature of method m in class C

$$\frac{\begin{array}{l}\texttt{class } \mathit{Cid}\texttt{<\_> extends \_<\_> \{ \_ \_ } ms\ \texttt{\{ } e\ \texttt{\} \_ \}} \in P \\ \text{MName}(ms) = m\end{array}}{\text{MSig}(\mathit{Cid}, m) =_o ms}\ \text{SMSC\_\_DEF}$$

$\boxed{\text{MSig}\left({}^s N, m, \overline{{}^s T}\right) =_o ms_o}$ m in ${}^s N$ with method type arguments $\overline{{}^s T}$ substituted

$$\frac{\begin{array}{l}\text{MSig}(\text{ClassOf}({}^s N), m) =_o p\ \texttt{<}\overline{X_l\ \texttt{extends}\ {}^s N_l}\texttt{>}\ {}^s T\ m\texttt{(}\overline{{}^s T'_q\ \mathit{pid}}\texttt{)} \\ \left({}^s N \rhd \overline{{}^s N_l}\right)\left[\overline{{}^s T_l}/\overline{X_l}\right] =_o \overline{{}^s N'_l} \qquad ({}^s N \rhd {}^s T)\left[\overline{{}^s T_l}/\overline{X_l}\right] =_o {}^s T' \\ \left({}^s N \rhd \overline{{}^s T'_q}\right)\left[\overline{{}^s T_l}/\overline{X_l}\right] =_o \overline{{}^s T''_q}\end{array}}{\text{MSig}\left({}^s N, m, \overline{{}^s T_l}\right) =_o p\ \texttt{<}\overline{X_l\ \texttt{extends}\ {}^s N'_l}\texttt{>}\ {}^s T'\ m\texttt{(}\overline{{}^s T''_q\ \mathit{pid}}\texttt{)}}\ \text{SMSN\_\_DEF}$$

$\boxed{\text{ClassDom}(C) =_o \overline{X}_o}$ look up type variables of class C

$$\frac{\texttt{class } \mathit{Cid}\texttt{<}\overline{X_k\ \texttt{extends \_}}\texttt{> extends \_<\_> \{ \_ \_ \}} \in P}{\text{ClassDom}(\mathit{Cid}) =_o \overline{X_k}}\ \text{SCD\_\_NVAR}$$

$$\frac{}{\text{ClassDom}(\texttt{Object}) =_o \emptyset}\ \text{SCD\_\_OBJECT}$$

$\boxed{\text{ClassBnds}(C) =_o \overline{{}^s N}_o}$ look up bounds of class C

$$\frac{\texttt{class } \mathit{Cid}\texttt{<}\overline{X_k\ \texttt{extends}\ {}^s N_k}\texttt{> extends \_<\_> \{ \_ \_ \}} \in P}{\text{ClassBnds}(\mathit{Cid}) =_o \overline{{}^s N_k}}\ \text{SCBC\_\_NVAR}$$

$$\frac{}{\text{ClassBnds}(\texttt{Object}) =_o \emptyset}\ \text{SCBC\_\_OBJECT}$$

$\boxed{\text{ClassBnds}({}^s N) =_o \overline{{}^s N}_o}$ look up bounds of type ${}^s N$

$$\frac{\begin{array}{l}\text{ClassBnds}(\text{ClassOf}({}^s N)) =_o \overline{{}^s N}_1 \\ {}^s N \rhd \overline{{}^s N}_1 =_o \overline{{}^s N}\end{array}}{\text{ClassBnds}({}^s N) =_o \overline{{}^s N}}\ \text{SCBN\_\_DEF}$$

$\boxed{u \rhd u' = u''}$ combining two ownership modifiers

$$\frac{}{\texttt{self} \rhd u = u}\ \text{UCU\_\_SELF}$$

$$\frac{}{\texttt{peer} \rhd \texttt{peer} = \texttt{peer}}\ \text{UCU\_\_PEER}$$

$$\frac{}{\texttt{rep} \rhd \texttt{peer} = \texttt{rep}}\ \text{UCU\_\_REP}$$

$$\frac{}{u \rhd \texttt{any} \;=\; \texttt{any}}\ \text{UCU\_\_ANY}$$

$$\frac{\text{otherwise}}{u \rhd u' \;=\; \texttt{lost}}\ \text{UCU\_\_LOST}$$

$\boxed{u \rhd {}^{s}T \;=\; {}^{s}T'}$ ownership modifier - type combination

$$\frac{}{u \rhd X \;=\; X}\ \text{UCT\_\_VAR}$$

$$\frac{u \rhd u' \;=\; u'' \qquad u \rhd \overline{{}^{s}T} \;=\; \overline{{}^{s}T}'}{u \rhd u'\ C\texttt{<}\overline{{}^{s}T}\texttt{>} \;=\; u''\ C\texttt{<}\overline{{}^{s}T}'\texttt{>}}\ \text{UCT\_\_NVAR}$$

$\boxed{u \rhd \overline{{}^{s}T} \;=\; \overline{{}^{s}T}'}$ ownership modifier - types combination

$$\frac{\overline{u \rhd {}^{s}T_k \;=\; {}^{s}T'_k}}{u \rhd \overline{{}^{s}T_k} \;=\; \overline{{}^{s}T'_k}}\ \text{UCTS\_\_DEF}$$

$\boxed{{}^{s}N \rhd {}^{s}T \;=_o\; {}^{s}T'_o}$ type - type combination

$$\frac{u \rhd {}^{s}T \;=\; {}^{s}T_1 \qquad {}^{s}T_1\left[\overline{{}^{s}T}/\overline{X}\right] = {}^{s}T' \qquad \text{ClassDom}(C) =_o \overline{X}}{u\ C\texttt{<}\overline{{}^{s}T}\texttt{>} \rhd {}^{s}T \;=_o\; {}^{s}T'}\ \text{TCT\_\_DEF}$$

$\boxed{{}^{s}N \rhd \overline{{}^{s}T} \;=_o\; \overline{{}^{s}T}'_o}$ type - types combination

$$\frac{\overline{{}^{s}N \rhd {}^{s}T_k \;=_o\; {}^{s}T'_k}}{{}^{s}N \rhd \overline{{}^{s}T_k} \;=_o\; \overline{{}^{s}T'_k}}\ \text{TCTS\_\_DEF}$$

$\boxed{\left({}^{s}N \rhd \overline{{}^{s}T}\right)\left[\overline{{}^{s}T}'/\overline{X}'\right] \;=_o\; \overline{{}^{s}T}''_o}$ type - types combination and substitution

$$\frac{{}^{s}N \rhd \overline{{}^{s}T} \;=_o\; \overline{{}^{s}T}_1 \qquad \overline{{}^{s}T}_1\left[\overline{{}^{s}T}'/\overline{X}'\right] = \overline{{}^{s}T}''}{\left({}^{s}N \rhd \overline{{}^{s}T}\right)\left[\overline{{}^{s}T}'/\overline{X}'\right] \;=_o\; \overline{{}^{s}T}''}\ \text{TCTSSUBSTS\_\_DEF}$$

$\boxed{u <:_u u'}$ ordering of ownership modifiers

$$\frac{}{\texttt{self} <:_u \texttt{peer}}\ \text{OMO\_\_TP}$$

$$\frac{}{\texttt{peer} <:_u \texttt{lost}}\ \text{OMO\_\_PL}$$

$$\frac{}{\texttt{rep} <:_u \texttt{lost}}\ \text{OMO\_\_RL}$$

$$\frac{}{u <:_u \texttt{any}}\ \text{OMO\_\_UA}$$

$$\frac{}{u <:_u u}\ \text{OMO\_\_REFL}$$

$\boxed{{}^{s}CT \sqsubseteq {}^{s}CT'}$ subclassing

$$\frac{\texttt{class}\ Cid\texttt{<}\overline{X_k\ \texttt{extends}\ \_}\texttt{>}\ \texttt{extends}\ C'\texttt{<}\overline{{}^{s}T}\texttt{>}\ \texttt{\{ \_ \_ \}} \in P}{Cid\texttt{<}\overline{X_k}\texttt{>} \sqsubseteq C'\texttt{<}\overline{{}^{s}T}\texttt{>}}\ \text{SC1}$$

$$\frac{\texttt{class}\ C\texttt{<}\overline{X_k\ \texttt{extends}\ \_}\texttt{>} \ldots \in P}{C\texttt{<}\overline{X_k}\texttt{>} \sqsubseteq C\texttt{<}\overline{X_k}\texttt{>}}\ \text{SC2}$$

$$\frac{\begin{array}{c} C\texttt{<}\overline{X}\texttt{>} \sqsubseteq C_1\texttt{<}\overline{{}^sT}_1\texttt{>} \\ C_1\texttt{<}\overline{X}_1\texttt{>} \sqsubseteq C'\texttt{<}\overline{{}^sT}'\texttt{>} \end{array}}{C\texttt{<}\overline{X}\texttt{>} \sqsubseteq C'\texttt{<}\overline{{}^sT}'\left[\overline{{}^sT}_1/\overline{X}_1\right]\texttt{>}}\ \text{SC3}$$

$\boxed{{}^s\Gamma \vdash {}^sT <: {}^sT'}$ static subtyping

$$\frac{\begin{array}{c} C\texttt{<}\overline{X}\texttt{>} \sqsubseteq C'\texttt{<}\overline{{}^sT}_1\texttt{>} \\ u\ C\texttt{<}\overline{{}^sT}\texttt{>} \rhd \overline{{}^sT}_1 =_o \overline{{}^sT}' \end{array}}{{}^s\Gamma \vdash u\ C\texttt{<}\overline{{}^sT}\texttt{>} <:\ u\ C'\texttt{<}\overline{{}^sT}'\texttt{>}}\ \text{ST1}$$

$$\frac{u <:_u u' \qquad \vdash \overline{{}^sT} <:_l \overline{{}^sT}'}{{}^s\Gamma \vdash u\ C\texttt{<}\overline{{}^sT}\texttt{>} <:\ u'\ C\texttt{<}\overline{{}^sT}'\texttt{>}}\ \text{ST2}$$

$$\frac{{}^sT = X \ \vee\ {}^s\Gamma(X) =_o {}^sT}{{}^s\Gamma \vdash X <:\ {}^sT}\ \text{ST3}$$

$$\frac{\begin{array}{c} {}^s\Gamma \vdash {}^sT <:\ {}^sT_1 \\ {}^s\Gamma \vdash {}^sT_1 <:\ {}^sT' \end{array}}{{}^s\Gamma \vdash {}^sT <:\ {}^sT'}\ \text{ST4}$$

$\boxed{\vdash {}^sT <:_l {}^sT'}$ type argument subtyping

$$\frac{\begin{array}{c} u' \in \{u\,,\ \texttt{lost}\} \\ \vdash \overline{{}^sT} <:_l \overline{{}^sT}' \end{array}}{\vdash u\ C\texttt{<}\overline{{}^sT}\texttt{>} <:_l u'\ C\texttt{<}\overline{{}^sT}'\texttt{>}}\ \text{AST1}$$

$$\frac{}{\vdash X <:_l X}\ \text{AST2}$$

$\boxed{{}^s\Gamma \vdash {}^sT <:_s {}^sT'}$ strict static subtyping

$$\frac{\begin{array}{c} {}^s\Gamma \vdash {}^sT <:\ {}^sT' \\ \texttt{lost} \notin {}^sT' \end{array}}{{}^s\Gamma \vdash {}^sT <:_s {}^sT'}\ \text{SSTDEF}$$

$\boxed{\vdash \overline{{}^sT} <:_l \overline{{}^sT}'}$ type argument subtypings

$$\frac{\overline{\vdash {}^sT_k <:_l {}^sT'_k}}{\vdash \overline{{}^sT_k} <:_l \overline{{}^sT'_k}}\ \text{ASTS\_\_DEF}$$

$\boxed{{}^s\Gamma \vdash \overline{{}^sT} <:\ \overline{{}^sT}'}$ static subtypings

$$\frac{\overline{{}^s\Gamma \vdash {}^sT_k <:\ {}^sT'_k}}{{}^s\Gamma \vdash \overline{{}^sT_k} <:\ \overline{{}^sT'_k}}\ \text{STS\_\_DEF}$$

$\boxed{{}^s\Gamma \vdash \overline{{}^sT} <:_s \overline{{}^sT}'}$ strict static subtypings

$$\frac{\overline{{}^s\Gamma \vdash {}^sT_k <:_s {}^sT'_k}}{{}^s\Gamma \vdash \overline{{}^sT_k} <:_s \overline{{}^sT'_k}}\ \text{SSTS\_\_DEF}$$

$\boxed{{}^s\Gamma \vdash e : {}^sT}$ expression typing

$$\frac{\begin{array}{c} {}^s\Gamma \vdash e : {}^sT_1 \\ {}^s\Gamma \vdash {}^sT_1 <:\ {}^sT \\ {}^s\Gamma \vdash {}^sT\ \ \mathsf{OK} \end{array}}{{}^s\Gamma \vdash e : {}^sT}\ \text{TR\_\_SUBSUM}$$

$$\frac{\begin{array}{c} \texttt{self} \notin {}^sT \\ {}^s\Gamma \vdash {}^sT\ \ \mathsf{OK} \end{array}}{{}^s\Gamma \vdash \texttt{null} : {}^sT}\ \text{TR\_\_NULL}$$

$$\frac{{}^s\Gamma(x) =_o {}^sT}{{}^s\Gamma \vdash x : {}^sT}\ \text{TR\_\_VAR}$$

$$\frac{{}^s\Gamma \vdash {}^sT\ \ \mathsf{strictly\ OK} \qquad \mathrm{om}({}^sT, {}^s\Gamma) \in \{\texttt{peer}, \texttt{rep}\}}{{}^s\Gamma \vdash \texttt{new}\ {}^sT\texttt{()} : {}^sT}\ \text{TR\_\_NEW}$$

$$\frac{{}^s\Gamma \vdash e_0 : {}^sN_0 \qquad \mathrm{FType}({}^sN_0, f) =_o {}^sT}{{}^s\Gamma \vdash e_0\texttt{.}f : {}^sT}\ \text{TR\_\_READ}$$

$$\frac{{}^s\Gamma \vdash e_0 : {}^sN_0 \qquad \mathrm{FType}({}^sN_0, f) =_o {}^sT \qquad {}^s\Gamma \vdash e_1 :_s {}^sT}{{}^s\Gamma \vdash e_0\texttt{.}f\ \texttt{=}\ e_1 : {}^sT}\ \text{TR\_\_WRITE}$$

$$\frac{{}^s\Gamma \vdash e_0 : {}^sN_0 \qquad \mathrm{MSig}\left({}^sN_0, m, \overline{{}^sT_l}\right) =_o \_\ \texttt{<}\overline{X_l\ \texttt{extends}\ {}^sN_l}\texttt{>}\ {}^sT\ m\texttt{(}\overline{{}^sT'_q\ pid}\texttt{)} \qquad {}^s\Gamma \vdash \overline{{}^sT_l} \mathrel{<:_s} \overline{{}^sN_l} \qquad {}^s\Gamma \vdash \overline{e_q} :_s \overline{{}^sT'_q}}{{}^s\Gamma \vdash e_0\ \texttt{.}\ m\texttt{<}\overline{{}^sT_l}\texttt{>(}\overline{e_q}\texttt{)} : {}^sT}\ \text{TR\_\_CALL}$$

$$\frac{{}^s\Gamma \vdash e : \_ \qquad {}^s\Gamma \vdash {}^sT\ \ \mathsf{OK}}{{}^s\Gamma \vdash \texttt{(}{}^sT\texttt{)}\ e : {}^sT}\ \text{TR\_\_CAST}$$

$\boxed{{}^s\Gamma \vdash \overline{e} : \overline{{}^sT}}$ expression typings

$$\frac{\overline{{}^s\Gamma \vdash e_k : {}^sT_k}}{{}^s\Gamma \vdash \overline{e_k} : \overline{{}^sT_k}}\ \text{TRM\_\_DEF}$$

$\boxed{{}^s\Gamma \vdash e :_s {}^sT}$ strict expression typing

$$\frac{{}^s\Gamma \vdash e : {}^sT \qquad \texttt{lost} \notin {}^sT}{{}^s\Gamma \vdash e :_s {}^sT}\ \text{STR\_\_DEF}$$

$\boxed{{}^s\Gamma \vdash \overline{e} :_s \overline{{}^sT}}$ strict expression typings

$$\frac{\overline{{}^s\Gamma \vdash e_k :_s {}^sT_k}}{{}^s\Gamma \vdash \overline{e_k} :_s \overline{{}^sT_k}}\ \text{STRM\_\_DEF}$$

$\boxed{{}^s\Gamma \vdash {}^sT\ \ \mathsf{OK}}$ well-formed static type

$$\frac{X \in {}^s\Gamma}{{}^s\Gamma \vdash X\ \ \mathsf{OK}}\ \text{WFT\_\_VAR}$$

$$\frac{{}^s\Gamma \vdash \overline{{}^sT}\ \ \mathsf{OK} \qquad \mathrm{ClassBnds}\left(u\ C\texttt{<}\overline{{}^sT}\texttt{>}\right) =_o \overline{{}^sN} \qquad \texttt{self} \notin \overline{{}^sT} \qquad {}^s\Gamma \vdash \overline{{}^sT} <: \overline{{}^sN}}{{}^s\Gamma \vdash u\ C\texttt{<}\overline{{}^sT}\texttt{>}\ \ \mathsf{OK}}\ \text{WFT\_\_NVAR}$$

$\boxed{{}^s\Gamma \vdash \overline{{}^sT}\ \ \mathsf{OK}}$ well-formed static types

$$\frac{\overline{{}^s\Gamma \vdash {}^sT_k\ \ \mathsf{OK}}}{{}^s\Gamma \vdash \overline{{}^sT_k}\ \ \mathsf{OK}}\ \text{WFTS\_\_DEF}$$

$\boxed{{}^s\Gamma \vdash {}^sT\ \ \mathsf{strictly\ OK}}$ strictly well-formed static type

$$\frac{X \in {}^s\Gamma}{{}^s\Gamma \vdash X\ \ \mathsf{strictly\ OK}}\ \text{SWFT\_\_VAR}$$

$$\frac{{}^s\Gamma \vdash \overline{{}^sT}\ \ \mathsf{strictly\ OK} \qquad \mathrm{ClassBnds}\left(u\ C\texttt{<}\overline{{}^sT}\texttt{>}\right) =_o \overline{{}^sN} \qquad \{\texttt{self}, \texttt{lost}\} \notin u\ C\texttt{<}\overline{{}^sT}\texttt{>} \qquad {}^s\Gamma \vdash \overline{{}^sT} <:_s \overline{{}^sN}}{{}^s\Gamma \vdash u\ C\texttt{<}\overline{{}^sT}\texttt{>}\ \ \mathsf{strictly\ OK}}\ \text{SWFT\_\_NVAR}$$

$\boxed{{}^{s}\Gamma \vdash \overline{{}^{s}T} \ \mathsf{strictly\ OK}}$ strictly well-formed static types

$$\frac{\overline{{}^{s}\Gamma \vdash {}^{s}T_k \ \mathsf{strictly\ OK}}}{{}^{s}\Gamma \vdash \overline{{}^{s}T_k} \ \mathsf{strictly\ OK}}\ \text{SWFTS\_\_DEF}$$

$\boxed{Cls \ \mathsf{OK}}$ well-formed class declaration

$$\frac{\begin{array}{c}{}^{s}\Gamma = \left\{\overline{X_k \mapsto {}^{s}N_k}\ ;\ \texttt{this} \mapsto \texttt{self}\ Cid\texttt{<}\overline{X_k}\texttt{>}, \_\right\} \\ {}^{s}\Gamma \vdash \overline{{}^{s}N_k} \ \mathsf{OK} \qquad \texttt{self} \notin \overline{{}^{s}N_k} \\ {}^{s}\Gamma \vdash \overline{{}^{s}T} \ \mathsf{strictly\ OK} \qquad \mathrm{ClassBnds}\big(\texttt{self}\ C\texttt{<}\overline{{}^{s}T}\texttt{>}\big) =_o \overline{{}^{s}N}' \qquad {}^{s}\Gamma \vdash \overline{{}^{s}T} <:_s \overline{{}^{s}N}' \\ {}^{s}\Gamma \vdash \overline{fd} \ \mathsf{OK} \qquad {}^{s}\Gamma, Cid \vdash \overline{md} \ \mathsf{OK}\end{array}}{\texttt{class}\ Cid\texttt{<}\overline{X_k\ \texttt{extends}\ {}^{s}N_k}\texttt{>}\ \texttt{extends}\ C\texttt{<}\overline{{}^{s}T}\texttt{>}\ \texttt{\{}\ \overline{fd}\ \overline{md}\ \texttt{\}} \ \mathsf{OK}}\ \text{WFC\_\_DEF}$$

$$\frac{}{\texttt{class Object \{\}} \ \mathsf{OK}}\ \text{WFC\_\_OBJECT}$$

$\boxed{{}^{s}\Gamma \vdash {}^{s}T\ f\texttt{;} \ \mathsf{OK}}$ well-formed field declaration

$$\frac{{}^{s}\Gamma \vdash {}^{s}T \ \mathsf{OK}}{{}^{s}\Gamma \vdash {}^{s}T\ f\texttt{;} \ \mathsf{OK}}\ \text{WFFD\_\_DEF}$$

$\boxed{{}^{s}\Gamma \vdash \overline{fd} \ \mathsf{OK}}$ well-formed field declarations

$$\frac{\overline{{}^{s}\Gamma \vdash {}^{s}T_i\ f_i\texttt{;} \ \mathsf{OK}}}{{}^{s}\Gamma \vdash \overline{{}^{s}T_i\ f_i\texttt{;}} \ \mathsf{OK}}\ \text{WFFDS\_\_DEF}$$

$\boxed{{}^{s}\Gamma, C \vdash md \ \mathsf{OK}}$ well-formed method declaration

$$\frac{\begin{array}{c}{}^{s}\Gamma = \left\{\overline{X'_k \mapsto {}^{s}N'_k}\ ;\ \texttt{this} \mapsto \texttt{self}\ C\texttt{<}\overline{X'_k}\texttt{>}, \_\right\} \\ {}^{s}\Gamma' = \left\{\overline{X'_k \mapsto {}^{s}N'_k}, \overline{X_l \mapsto {}^{s}N_l}\ ;\ \texttt{this} \mapsto \texttt{self}\ C\texttt{<}\overline{X'_k}\texttt{>}, \overline{pid \mapsto {}^{s}T_q}\right\} \\ {}^{s}\Gamma' \vdash \overline{{}^{s}N_l}, {}^{s}T, \overline{{}^{s}T_q} \ \mathsf{OK} \qquad \texttt{self} \notin \overline{{}^{s}N_l} \\ {}^{s}\Gamma' \vdash e : {}^{s}T \qquad C\texttt{<}\overline{X'_k}\texttt{>} \vdash m \ \mathsf{OK}\end{array}}{{}^{s}\Gamma, C \vdash \texttt{\_ <}\overline{X_l\ \texttt{extends}\ {}^{s}N_l}\texttt{>}\ {}^{s}T\ m\texttt{(}\overline{{}^{s}T_q\ pid}\texttt{)\ \{}\ e\ \texttt{\}} \ \mathsf{OK}}\ \text{WFMD\_\_DEF}$$

$\boxed{{}^{s}\Gamma, C \vdash \overline{md} \ \mathsf{OK}}$ well-formed method declarations

$$\frac{\overline{{}^{s}\Gamma, C \vdash md_k \ \mathsf{OK}}}{{}^{s}\Gamma, C \vdash \overline{md_k} \ \mathsf{OK}}\ \text{WFMDS\_\_DEF}$$

$\boxed{{}^{s}CT \vdash m \ \mathsf{OK}}$ method overriding OK

$$\frac{\forall C'\texttt{<}\overline{X}'\texttt{>}.\ \forall \overline{{}^{s}T}.\ \Big(C\texttt{<}\overline{X}\texttt{>} \sqsubseteq C'\texttt{<}\overline{{}^{s}T}\texttt{>} \implies C\texttt{<}\overline{X}\texttt{>}, C'\texttt{<}\overline{{}^{s}T},\ \overline{X}'\texttt{>} \vdash m \ \mathsf{OK}\Big)}{C\texttt{<}\overline{X}\texttt{>} \vdash m \ \mathsf{OK}}\ \text{OVR\_\_DEF}$$

$\boxed{{}^{s}CT, C\texttt{<}\overline{{}^{s}T},\ \overline{X}\texttt{>} \vdash m \ \mathsf{OK}}$ method overriding OK auxiliary

$$\frac{\begin{array}{c}\mathrm{MSig}(C, m) =_o ms \qquad \mathrm{MSig}(C', m) =_o ms'_o \\ ms'_o =_o None \ \vee \ \Big(ms'_o =_o ms' \ \wedge \ ms'[\overline{{}^{s}T}/\overline{X}'] = ms\Big)\end{array}}{C\texttt{<}\overline{X}\texttt{>}, C'\texttt{<}\overline{{}^{s}T},\ \overline{X}'\texttt{>} \vdash m \ \mathsf{OK}}\ \text{OVRA\_\_DEF}$$

$\boxed{{}^{s}\Gamma \ \mathsf{OK}}$ well-formed static environment

$$\frac{\begin{array}{c}{}^{s}\Gamma = \left\{\overline{X_k \mapsto {}^{s}N_k}, \overline{X'_l \mapsto {}^{s}N'_l}\ ;\ \texttt{this} \mapsto \texttt{self}\ C\texttt{<}\overline{X_k}\texttt{>}, \overline{pid \mapsto {}^{s}T_q}\right\} \\ \mathrm{ClassDom}(C) =_o \overline{X_k} \qquad \mathrm{ClassBnds}(C) =_o \overline{{}^{s}N_k} \\ {}^{s}\Gamma \vdash \overline{{}^{s}T_q}, \overline{{}^{s}N_k}, \overline{{}^{s}N'_l} \ \mathsf{OK} \qquad \texttt{self} \notin \overline{{}^{s}N_k}, \overline{{}^{s}N'_l}\end{array}}{{}^{s}\Gamma \ \mathsf{OK}}\ \text{SWFE\_\_DEF}$$

$\boxed{\vdash P \ \ \mathsf{OK}}$ well-formed program

$$\frac{\begin{array}{l}\overline{Cls_i \ \ \mathsf{OK}} \\ \{\emptyset \ ; \ \mathtt{this} \mapsto \mathtt{self}\ C\mathtt{<>}\} \vdash \mathtt{self}\ C\mathtt{<>} \ \ \mathsf{OK} \\ \{\emptyset \ ; \ \mathtt{this} \mapsto \mathtt{self}\ C\mathtt{<>}\} \vdash e : \_ \\ \forall C', C''.\ ((C'\mathtt{<\_>} \sqsubseteq C''\mathtt{<\_>} \ \wedge \ C''\mathtt{<\_>} \sqsubseteq C'\mathtt{<\_>}) \implies C' = C'')\end{array}}{\vdash \overline{Cls_i},\ C,\ e \ \ \mathsf{OK}}\ \text{WFP\_\_DEF}$$

$\boxed{{}^{s}\Gamma \vdash e \ \ \mathsf{enc}}$ encapsulated expression

$$\frac{{}^{s}\Gamma \vdash \mathtt{null} : \_}{{}^{s}\Gamma \vdash \mathtt{null} \ \ \mathsf{enc}}\ \text{E\_\_NULL}$$

$$\frac{{}^{s}\Gamma \vdash x : \_}{{}^{s}\Gamma \vdash x \ \ \mathsf{enc}}\ \text{E\_\_VAR}$$

$$\frac{{}^{s}\Gamma \vdash \mathtt{new}\ {}^{s}T\mathtt{()} : \_}{{}^{s}\Gamma \vdash \mathtt{new}\ {}^{s}T\mathtt{()} \ \ \mathsf{enc}}\ \text{E\_\_NEW}$$

$$\frac{{}^{s}\Gamma \vdash e_0\mathtt{.}f : \_ \quad {}^{s}\Gamma \vdash e_0 \ \ \mathsf{enc}}{{}^{s}\Gamma \vdash e_0\mathtt{.}f \ \ \mathsf{enc}}\ \text{E\_\_READ}$$

$$\frac{\begin{array}{l}{}^{s}\Gamma \vdash e_0\mathtt{.}f\ \mathtt{=}\ e_1 : \_ \\ {}^{s}\Gamma \vdash e_0 : {}^{s}N_0 \qquad {}^{s}\Gamma \vdash e_0 \ \ \mathsf{enc} \qquad {}^{s}\Gamma \vdash e_1 \ \ \mathsf{enc} \\ \mathrm{om}({}^{s}N_0) \in \{\mathtt{self}, \mathtt{peer}, \mathtt{rep}\}\end{array}}{{}^{s}\Gamma \vdash e_0\mathtt{.}f\ \mathtt{=}\ e_1 \ \ \mathsf{enc}}\ \text{E\_\_WRITE}$$

$$\frac{\begin{array}{l}{}^{s}\Gamma \vdash e_0\ \mathtt{.}\ m\mathtt{<}\overline{{}^{s}T}\mathtt{>(}\overline{e}\mathtt{)} : \_ \\ {}^{s}\Gamma \vdash e_0 : {}^{s}N_0 \qquad {}^{s}\Gamma \vdash e_0 \ \ \mathsf{enc} \qquad {}^{s}\Gamma \vdash \overline{e} \ \ \mathsf{enc} \\ \mathrm{om}({}^{s}N_0) \in \{\mathtt{self}, \mathtt{peer}, \mathtt{rep}\} \ \vee \ \mathrm{MSig}\left({}^{s}N_0, m, \overline{{}^{s}T}\right) =_o \mathtt{pure\ <\_>\ \_}\ m\mathtt{(\_)}\end{array}}{{}^{s}\Gamma \vdash e_0\ \mathtt{.}\ m\mathtt{<}\overline{{}^{s}T}\mathtt{>(}\overline{e}\mathtt{)} \ \ \mathsf{enc}}\ \text{E\_\_CALL}$$

$$\frac{{}^{s}\Gamma \vdash \mathtt{(}{}^{s}T\mathtt{)}\ e : \_ \quad {}^{s}\Gamma \vdash e \ \ \mathsf{enc}}{{}^{s}\Gamma \vdash \mathtt{(}{}^{s}T\mathtt{)}\ e \ \ \mathsf{enc}}\ \text{E\_\_CAST}$$

$\boxed{{}^{s}\Gamma \vdash \overline{e} \ \ \mathsf{enc}}$ encapsulated expressions

$$\frac{\overline{{}^{s}\Gamma \vdash e_k \ \ \mathsf{enc}}}{{}^{s}\Gamma \vdash \overline{e_k} \ \ \mathsf{enc}}\ \text{EM\_\_DEF}$$

$\boxed{{}^{s}\Gamma, C \vdash md \ \ \mathsf{enc}}$ encapsulated method declaration

$$\frac{\begin{array}{l}{}^{s}\Gamma, C \vdash p\ \mathtt{<}\overline{X_l\ \mathtt{extends}\ {}^{s}N_l}\mathtt{>}\ {}^{s}T\ m\mathtt{(}\overline{{}^{s}T_q\ pid}\mathtt{)\ \{}\ e\ \mathtt{\}} \ \ \mathsf{OK} \\ {}^{s}\Gamma = \left\{\overline{X'_k \mapsto {}^{s}N'_k}\ ;\ \mathtt{this} \mapsto \mathtt{self}\ C\mathtt{<}\overline{X'_k}\mathtt{>}, \_\right\} \\ {}^{s}\Gamma' = \left\{\overline{X'_k \mapsto {}^{s}N'_k}, \overline{X_l \mapsto {}^{s}N_l}\ ;\ \mathtt{this} \mapsto \mathtt{self}\ C\mathtt{<}\overline{X'_k}\mathtt{>}, \overline{pid \mapsto {}^{s}T_q}\right\} \\ \left(p = \mathtt{pure} \implies {}^{s}\Gamma' \vdash e \ \ \mathsf{pure}\right) \qquad \left(p = \mathtt{impure} \implies {}^{s}\Gamma' \vdash e \ \ \mathsf{enc}\right)\end{array}}{{}^{s}\Gamma, C \vdash p\ \mathtt{<}\overline{X_l\ \mathtt{extends}\ {}^{s}N_l}\mathtt{>}\ {}^{s}T\ m\mathtt{(}\overline{{}^{s}T_q\ pid}\mathtt{)\ \{}\ e\ \mathtt{\}} \ \ \mathsf{enc}}\ \text{EMD\_\_DEF}$$

$\boxed{{}^{s}\Gamma, C \vdash \overline{md} \ \ \mathsf{enc}}$ encapsulated method declarations

$$\frac{\overline{{}^{s}\Gamma, C \vdash md_i \ \ \mathsf{enc}}}{{}^{s}\Gamma, C \vdash \overline{md_i} \ \ \mathsf{enc}}\ \text{EMDS\_\_DEF}$$

$\boxed{Cls \ \ \mathsf{enc}}$ encapsulated class declaration

$$\frac{\begin{array}{l}\mathtt{class}\ Cid\mathtt{<}\overline{X_k\ \mathtt{extends}\ {}^{s}N_k}\mathtt{>}\ \mathtt{extends}\ C\mathtt{<}\overline{{}^{s}T}\mathtt{>\ \{}\ \overline{fd}\ \overline{md}\ \mathtt{\}} \ \ \mathsf{OK} \\ {}^{s}\Gamma = \left\{\overline{X_k \mapsto {}^{s}N_k}\ ;\ \mathtt{this} \mapsto \mathtt{self}\ Cid\mathtt{<}\overline{X_k}\mathtt{>}, \_\right\} \qquad {}^{s}\Gamma, Cid \vdash \overline{md} \ \ \mathsf{enc}\end{array}}{\mathtt{class}\ Cid\mathtt{<}\overline{X_k\ \mathtt{extends}\ {}^{s}N_k}\mathtt{>}\ \mathtt{extends}\ C\mathtt{<}\overline{{}^{s}T}\mathtt{>\ \{}\ \overline{fd}\ \overline{md}\ \mathtt{\}} \ \ \mathsf{enc}}\ \text{EC\_\_DEF}$$

$$\frac{}{\texttt{class Object \{\}}\ \ \mathsf{enc}}\ \text{EC\_\_OBJECT}$$

$\boxed{\vdash P\ \ \mathsf{enc}}$ encapsulated program

$$\frac{\begin{array}{l}\vdash \overline{Cls},\ C,\ e\ \ \mathsf{OK}\\ \overline{Cls_k}\ \ \mathsf{enc}\\ \{\emptyset\ ;\ \texttt{this} \mapsto \texttt{self}\ C\texttt{<>}\} \vdash e\ \ \mathsf{enc}\end{array}}{\vdash \overline{Cls},\ C,\ e\ \ \mathsf{enc}}\ \text{EP\_\_DEF}$$

$\boxed{{}^s\Gamma \vdash e\ \ \mathsf{pure}}$ pure expression $\boxed{{}^s\Gamma \vdash e\ \ \mathsf{strictly\ pure}}$ strictly pure expression

$$\frac{{}^s\Gamma \vdash \texttt{null} : \_}{{}^s\Gamma \vdash \texttt{null}\ \ \mathsf{strictly\ pure}}\ \text{SP\_\_NULL}$$

$$\frac{{}^s\Gamma \vdash x : \_}{{}^s\Gamma \vdash x\ \ \mathsf{strictly\ pure}}\ \text{SP\_\_VAR}$$

$$\frac{{}^s\Gamma \vdash \texttt{new}\ {}^sT\texttt{()} : \_}{{}^s\Gamma \vdash \texttt{new}\ {}^sT\texttt{()}\ \ \mathsf{strictly\ pure}}\ \text{SP\_\_NEW}$$

$$\frac{\begin{array}{l}{}^s\Gamma \vdash e_0\texttt{.}f : \_\\ {}^s\Gamma \vdash e_0\ \ \mathsf{strictly\ pure}\end{array}}{{}^s\Gamma \vdash e_0\texttt{.}f\ \ \mathsf{strictly\ pure}}\ \text{SP\_\_READ}$$

$$\frac{\begin{array}{l}{}^s\Gamma \vdash e_0\ \texttt{.}\ m\texttt{<}\overline{{}^sT}\texttt{>(}\overline{e}\texttt{)} : \_\\ {}^s\Gamma \vdash e_0 : {}^sN_0 \qquad {}^s\Gamma \vdash e_0\ \ \mathsf{strictly\ pure} \qquad {}^s\Gamma \vdash \overline{e}\ \ \mathsf{strictly\ pure}\\ \mathrm{MSig}\left({}^sN_0, m, \overline{{}^sT}\right) =_o \texttt{pure <\_> \_}\ m\texttt{(\_)}\end{array}}{{}^s\Gamma \vdash e_0\ \texttt{.}\ m\texttt{<}\overline{{}^sT}\texttt{>(}\overline{e}\texttt{)}\ \ \mathsf{strictly\ pure}}\ \text{SP\_\_CALL}$$

$$\frac{\begin{array}{l}{}^s\Gamma \vdash \texttt{(}{}^sT\texttt{)}\ e : \_\\ {}^s\Gamma \vdash e\ \ \mathsf{strictly\ pure}\end{array}}{{}^s\Gamma \vdash \texttt{(}{}^sT\texttt{)}\ e\ \ \mathsf{strictly\ pure}}\ \text{SP\_\_CAST}$$

$\boxed{{}^s\Gamma \vdash \overline{e}\ \ \mathsf{strictly\ pure}}$ pure expressions

$$\frac{\overline{{}^s\Gamma \vdash e_k\ \ \mathsf{strictly\ pure}}}{{}^s\Gamma \vdash \overline{e_k}\ \ \mathsf{strictly\ pure}}\ \text{PM\_\_DEF}$$

$\boxed{{}^s\Gamma \vdash {}^sT\ \ \mathsf{prg\ OK}}$ reasonable static type

$$\frac{X \in {}^s\Gamma}{{}^s\Gamma \vdash X\ \ \mathsf{prg\ OK}}\ \text{PWFT\_\_VAR}$$

$$\frac{\begin{array}{l}{}^s\Gamma \vdash \overline{{}^sT}\ \ \mathsf{prg\ OK} \qquad \texttt{self} \notin u\ C\texttt{<}\overline{{}^sT}\texttt{>}\\ \mathrm{ClassBnds}(C) =_o \overline{{}^sN} \qquad u\ C\texttt{<}\overline{{}^sT}\texttt{>} \rhd \overline{{}^sN} =_o \overline{{}^sN}'\\ {}^s\Gamma, u\ C\texttt{<}\overline{{}^sT}\texttt{>} \vdash \overline{{}^sT} <: \overline{{}^sN}, \overline{{}^sN}'\end{array}}{{}^s\Gamma \vdash u\ C\texttt{<}\overline{{}^sT}\texttt{>}\ \ \mathsf{prg\ OK}}\ \text{PWFT\_\_NVAR}$$

$\boxed{{}^s\Gamma, {}^sN'' \vdash {}^sT <: {}^sN, {}^sN'}$ reasonable static type argument

$$\frac{\begin{array}{l}{}^s\Gamma \vdash {}^sT <: {}^sN'\\ \mathrm{ClassDom}(C) =_o \overline{X_k} \qquad \mathrm{ClassBnds}(C) =_o \overline{{}^sN_k}\\ {}^s\Gamma' = \left\{\overline{X_k \mapsto {}^sN_k}\ ;\ \texttt{this} \mapsto \texttt{self}\ C\texttt{<}\overline{X_k}\texttt{>}, \_\right\}\\ \exists\, {}^sT_0.\ \left(u\ C\texttt{<}\overline{{}^sT}\texttt{>} \rhd {}^sT_0 =_o {}^sT\ \wedge\ {}^s\Gamma' \vdash {}^sT_0 <: {}^sN\right)\end{array}}{{}^s\Gamma, u\ C\texttt{<}\overline{{}^sT}\texttt{>} \vdash {}^sT <: {}^sN, {}^sN'}\ \text{PWFTA\_\_DEF}$$

$\boxed{{}^s\Gamma, {}^sN'' \vdash \overline{{}^sT} <: \overline{{}^sN}, \overline{{}^sN}'}$ reasonable static type arguments

$$\frac{\overline{{}^{s}\Gamma, {}^{s}N'' \vdash {}^{s}T_k <: {}^{s}N_k, {}^{s}N'_k}}{{}^{s}\Gamma, {}^{s}N'' \vdash \overline{{}^{s}T_k} <: \overline{{}^{s}N_k}, \overline{{}^{s}N'_k}}\ \text{PWFTAS\_\_DEF}$$

$\boxed{{}^{s}\Gamma \vdash \overline{{}^{s}T}\ \ \mathsf{prg\ OK}}$ reasonable static types

$$\frac{\overline{{}^{s}\Gamma \vdash {}^{s}T_k\ \ \mathsf{prg\ OK}}}{{}^{s}\Gamma \vdash \overline{{}^{s}T_k}\ \ \mathsf{prg\ OK}}\ \text{PTS\_\_DEF}$$

$\boxed{Cls\ \ \mathsf{prg\ OK}}$ reasonable class declaration

$$\frac{\begin{array}{l}\texttt{class}\ Cid\texttt{<}\overline{X_k\ \texttt{extends}\ {}^{s}N_k}\texttt{>}\ \texttt{extends}\ C\texttt{<}\overline{{}^{s}T}\texttt{>}\ \texttt{\{}\ \overline{fd}\ \overline{md}\ \texttt{\}}\ \ \mathsf{OK} \\ {}^{s}\Gamma = \left\{\overline{X_k \mapsto {}^{s}N_k}\ ;\ \texttt{this} \mapsto \texttt{self}\ Cid\texttt{<}\overline{X_k}\texttt{>}, \_\right\} \\ {}^{s}\Gamma \vdash \overline{{}^{s}N_k}\ \ \mathsf{prg\ OK} \qquad \texttt{lost} \notin \overline{{}^{s}N_k} \\ {}^{s}\Gamma \vdash \overline{fd}\ \ \mathsf{prg\ OK} \qquad {}^{s}\Gamma, Cid \vdash \overline{md}\ \ \mathsf{prg\ OK}\end{array}}{\texttt{class}\ Cid\texttt{<}\overline{X_k\ \texttt{extends}\ {}^{s}N_k}\texttt{>}\ \texttt{extends}\ C\texttt{<}\overline{{}^{s}T}\texttt{>}\ \texttt{\{}\ \overline{fd}\ \overline{md}\ \texttt{\}}\ \ \mathsf{prg\ OK}}\ \text{PC\_\_DEF}$$

$$\frac{}{\texttt{class Object \{\}}\ \ \mathsf{prg\ OK}}\ \text{PC\_\_OBJECT}$$

$\boxed{{}^{s}\Gamma \vdash {}^{s}T\ f\texttt{;}\ \ \mathsf{prg\ OK}}$ reasonable field declaration

$$\frac{{}^{s}\Gamma \vdash {}^{s}T\ \ \mathsf{prg\ OK} \qquad \texttt{lost} \notin {}^{s}T}{{}^{s}\Gamma \vdash {}^{s}T\ f\texttt{;}\ \ \mathsf{prg\ OK}}\ \text{PFD\_\_DEF}$$

$\boxed{{}^{s}\Gamma \vdash \overline{fd}\ \ \mathsf{prg\ OK}}$ reasonable field declarations

$$\frac{\overline{{}^{s}\Gamma \vdash {}^{s}T_i\ f_i\texttt{;}\ \ \mathsf{prg\ OK}}}{{}^{s}\Gamma \vdash \overline{{}^{s}T_i\ f_i\texttt{;}}\ \ \mathsf{prg\ OK}}\ \text{PFDS\_\_DEF}$$

$\boxed{{}^{s}\Gamma, C \vdash md\ \ \mathsf{prg\ OK}}$ reasonable method declaration

$$\frac{\begin{array}{l}{}^{s}\Gamma, C \vdash p\ \texttt{<}\overline{X_l\ \texttt{extends}\ {}^{s}N_l}\texttt{>}\ {}^{s}T\ m\texttt{(}\overline{{}^{s}T_q\ pid}\texttt{)}\ \texttt{\{}\ e\ \texttt{\}}\ \ \mathsf{OK} \\ {}^{s}\Gamma = \left\{\overline{X'_k \mapsto {}^{s}N'_k}\ ;\ \texttt{this} \mapsto \texttt{self}\ C\texttt{<}\overline{X'_k}\texttt{>}, \_\right\} \\ {}^{s}\Gamma' = \left\{\overline{X'_k \mapsto {}^{s}N'_k}, \overline{X_l \mapsto {}^{s}N_l}\ ;\ \texttt{this} \mapsto \texttt{self}\ C\texttt{<}\overline{X'_k}\texttt{>}, \overline{pid \mapsto {}^{s}T_q}\right\} \\ {}^{s}\Gamma' \vdash \overline{{}^{s}N_l}, {}^{s}T, \overline{{}^{s}T_q}\ \ \mathsf{prg\ OK} \qquad \texttt{lost} \notin \overline{{}^{s}N_l}, \overline{{}^{s}T_q} \qquad {}^{s}\Gamma' \vdash e\ \ \mathsf{prg\ OK} \\ p = \texttt{pure} \implies \left(\mathrm{free}\left(\overline{{}^{s}N_l}, \overline{{}^{s}T_q}\right) \subseteq \emptyset\ \wedge\ \texttt{any} \rhd \overline{{}^{s}N_l}, \overline{{}^{s}T_q} = \overline{{}^{s}N_l}, \overline{{}^{s}T_q}\right)\end{array}}{{}^{s}\Gamma, C \vdash p\ \texttt{<}\overline{X_l\ \texttt{extends}\ {}^{s}N_l}\texttt{>}\ {}^{s}T\ m\texttt{(}\overline{{}^{s}T_q\ pid}\texttt{)}\ \texttt{\{}\ e\ \texttt{\}}\ \ \mathsf{prg\ OK}}\ \text{PMD\_\_DEF}$$

$\boxed{{}^{s}\Gamma, C \vdash \overline{md}\ \ \mathsf{prg\ OK}}$ reasonable method declarations

$$\frac{\overline{{}^{s}\Gamma, C \vdash md_i\ \ \mathsf{prg\ OK}}}{{}^{s}\Gamma, C \vdash \overline{md_i}\ \ \mathsf{prg\ OK}}\ \text{PMDS\_\_DEF}$$

$\boxed{\vdash P\ \ \mathsf{prg\ OK}}$ reasonable program

$$\frac{\begin{array}{l}\vdash \overline{Cls_i},\ C,\ e\ \ \mathsf{OK} \\ \overline{Cls_i\ \ \mathsf{prg\ OK}} \\ \{\emptyset\ ;\ \texttt{this} \mapsto \texttt{self}\ C\texttt{<}\emptyset\texttt{>}\} \vdash e\ \ \mathsf{prg\ OK}\end{array}}{\vdash \overline{Cls_i},\ C,\ e\ \ \mathsf{prg\ OK}}\ \text{PP\_\_DEF}$$

$\boxed{{}^{s}\Gamma \vdash e\ \ \mathsf{prg\ OK}}$ reasonable expression

$$\frac{{}^{s}\Gamma \vdash \texttt{null} : \_}{{}^{s}\Gamma \vdash \texttt{null}\ \ \mathsf{prg\ OK}}\ \text{PE\_\_NULL}$$

$$\frac{{}^s\Gamma \vdash x : \_}{{}^s\Gamma \vdash x \ \ \mathsf{prg\ OK}}\ \text{PE\_\_VAR}$$

$$\frac{{}^s\Gamma \vdash \texttt{new}\ {}^sT\texttt{()} : \_}{{}^s\Gamma \vdash \texttt{new}\ {}^sT\texttt{()} \ \ \mathsf{prg\ OK}}\ \text{PE\_\_NEW}$$

$$\frac{{}^s\Gamma \vdash e_0\texttt{.}f : \_ \quad {}^s\Gamma \vdash e_0 \ \ \mathsf{prg\ OK}}{{}^s\Gamma \vdash e_0\texttt{.}f \ \ \mathsf{prg\ OK}}\ \text{PE\_\_READ}$$

$$\frac{{}^s\Gamma \vdash e_0\texttt{.}f\ \texttt{=}\ e_1 : \_ \quad {}^s\Gamma \vdash e_0 \ \ \mathsf{prg\ OK} \quad {}^s\Gamma \vdash e_1 \ \ \mathsf{prg\ OK}}{{}^s\Gamma \vdash e_0\texttt{.}f\ \texttt{=}\ e_1 \ \ \mathsf{prg\ OK}}\ \text{PE\_\_WRITE}$$

$$\frac{{}^s\Gamma \vdash e_0\ \texttt{.}\ m\texttt{<}\overline{{}^sT}\texttt{>(}\overline{e}\texttt{)} : \_ \quad {}^s\Gamma \vdash e_0 \ \ \mathsf{prg\ OK} \quad {}^s\Gamma \vdash \overline{e} \ \ \mathsf{prg\ OK}}{{}^s\Gamma \vdash e_0\ \texttt{.}\ m\texttt{<}\overline{{}^sT}\texttt{>(}\overline{e}\texttt{)} \ \ \mathsf{prg\ OK}}\ \text{PE\_\_CALL}$$

$$\frac{{}^s\Gamma \vdash \texttt{(}{}^sT\texttt{)}\ e : \_ \quad {}^s\Gamma \vdash e \ \ \mathsf{prg\ OK} \quad {}^s\Gamma \vdash {}^sT \ \ \mathsf{prg\ OK}}{{}^s\Gamma \vdash \texttt{(}{}^sT\texttt{)}\ e \ \ \mathsf{prg\ OK}}\ \text{PE\_\_CAST}$$

$\boxed{{}^s\Gamma \vdash \overline{e} \ \ \mathsf{prg\ OK}}$ reasonable expressions

$$\frac{\overline{{}^s\Gamma \vdash e_i \ \ \mathsf{prg\ OK}}}{{}^s\Gamma \vdash \overline{e_i} \ \ \mathsf{prg\ OK}}\ \text{PEM\_\_DEF}$$

$\boxed{h + o \ = \ (h', \iota)}$ add object o to heap h resulting in heap h' and fresh address ι

$$\frac{\iota \notin \mathrm{dom}(h) \qquad h' = h \ + \ (\iota \mapsto o)}{h + o \ = \ (h', \iota)}\ \text{HNEW\_\_DEF}$$

$\boxed{h[\iota\texttt{.}f\ \texttt{=}\ v] \ =_o \ h'}$ field update in heap

$$\frac{\begin{array}{l} v = \texttt{null}_a \ \vee \ (v = \iota' \ \wedge \ \iota' \in \mathrm{dom}(h)) \\ h(\iota) =_o \left({}^rT, \overline{fv}\right) \qquad f \in \mathrm{dom}\left(\overline{fv}\right) \qquad \overline{fv}' = \overline{fv}[f \mapsto v] \\ h' = h \ + \ \left(\iota \mapsto \left({}^rT, \overline{fv}'\right)\right) \end{array}}{h[\iota\texttt{.}f\ \texttt{=}\ v] \ =_o \ h'}\ \text{HUP\_\_DEF}$$

$\boxed{\mathrm{FType}(h, \iota, f) \ =_o \ {}^sT_o}$ look up type of field in heap

$$\frac{h \vdash \iota : \ \_\ C\texttt{<\_>} \qquad \mathrm{FType}(C, f) \ =_o \ {}^sT}{\mathrm{FType}(h, \iota, f) \ =_o \ {}^sT}\ \text{RFT\_\_DEF}$$

$\boxed{\mathrm{MSig}(h, \iota, m) \ =_o \ ms_o}$ look up method signature of method m at ι

$$\frac{h \vdash \iota : \ \_\ C\texttt{<\_>} \qquad \mathrm{MSig}(C, m) \ =_o \ ms}{\mathrm{MSig}(h, \iota, m) \ =_o \ ms}\ \text{RMS\_\_DEF}$$

$\boxed{\mathrm{MBody}(C, m) \ =_o \ e_o}$ look up most-concrete body of m in class C or a superclass

$$\frac{\texttt{class}\ Cid\texttt{<\_>}\ \texttt{extends}\ \texttt{\_<\_>}\ \texttt{\{}\ \texttt{\_}\ \texttt{\_}\ ms\ \texttt{\{}\ e\ \texttt{\}}\ \texttt{\_}\ \texttt{\}} \in P \quad \mathrm{MName}(ms) = m}{\mathrm{MBody}(Cid, m) \ =_o \ e}\ \text{SMBC\_\_FOUND}$$

$$\frac{\texttt{class}\ Cid\texttt{<\_>}\ \texttt{extends}\ C_1\texttt{<\_>}\ \texttt{\{}\ \texttt{\_}\ \overline{ms_n\ \texttt{\{}\ e_n\ \texttt{\}}}\ \texttt{\}} \in P \quad \overline{\mathrm{MName}(ms_n) \neq m} \qquad \mathrm{MBody}(C_1, m) \ =_o \ e}{\mathrm{MBody}(Cid, m) \ =_o \ e}\ \text{SMBC\_\_INH}$$

$\boxed{\text{MBody}(h, \iota, m) =_o e_o}$ look up most-concrete body of method m at ι

$$\frac{h(\iota)\downarrow_1 =_o \_\ C\texttt{<\_>} \qquad \text{MBody}(C, m) =_o e}{\text{MBody}(h, \iota, m) =_o e}\ \text{RMB\_\_DEF}$$

$\boxed{\text{sdyn}\big(\overline{{}^sT}, h, \iota, {}^rT, \overline{{}^o\iota}\big) =_o \overline{{}^rT}}$ simple dynamization of types $\overline{{}^sT}$

$$\frac{\begin{array}{c}{}^o\iota' \in \text{dom}(h) \cup \{\texttt{root}_a\} \qquad {}^o\iota \neq \texttt{any}_a \implies {}^o\iota' = {}^o\iota \\ \text{ClassDom}(C) =_o \overline{X} \qquad \texttt{rep} \in \overline{{}^sT} \implies \text{owner}(h, \iota) =_o {}^o\iota' \\ \overline{{}^sT}\left[{}^o\iota' / \texttt{peer}, \iota / \texttt{rep}, \texttt{any}_a / \texttt{any}, \overline{{}^rT} / \overline{X}, \overline{{}^o\iota_i} / \texttt{lost}\right] =_o \overline{{}^rT}'\end{array}}{\text{sdyn}\big(\overline{{}^sT}, h, \iota, {}^o\iota\ C\texttt{<}\overline{{}^rT}\texttt{>}, \overline{{}^o\iota_i}\big) =_o \overline{{}^rT}'}\ \text{SDYN}$$

$\boxed{\text{ClassBnds}(h, \iota, {}^rT, \overline{{}^o\iota}) =_o \overline{{}^rT}}$ upper bounds of type ${}^rT$ from viewpoint ι

$$\frac{\text{ClassBnds}(\text{ClassOf}({}^rT)) =_o \overline{{}^sN} \qquad \text{sdyn}\big(\overline{{}^sN}, h, \iota, {}^rT, \overline{{}^o\iota}\big) =_o \overline{{}^rT}}{\text{ClassBnds}(h, \iota, {}^rT, \overline{{}^o\iota}) =_o \overline{{}^rT}}\ \text{RCB\_\_DEF}$$

$\boxed{h \vdash {}^rT <:\ {}^rT'}$ type ${}^rT$ is a subtype of ${}^rT'$

$$\frac{\begin{array}{c}C\texttt{<}\overline{X}\texttt{>} \sqsubseteq C'\texttt{<}\overline{{}^sT}\texttt{>} \qquad {}^o\iota' \in \{{}^o\iota, \texttt{any}_a\} \\ \text{sdyn}\big(\overline{{}^sT}, h, \_, {}^o\iota\ C\texttt{<}\overline{{}^rT}\texttt{>}, \overline{{}^o\iota}\big) =_o \overline{{}^rT}'\end{array}}{h \vdash {}^o\iota\ C\texttt{<}\overline{{}^rT}\texttt{>} <:\ {}^o\iota'\ C'\texttt{<}\overline{{}^rT}'\texttt{>}}\ \text{RT\_\_DEF}$$

$\boxed{h \vdash \overline{{}^rT} <:\ \overline{{}^rT}'}$ runtime subtypings

$$\frac{\overline{h \vdash {}^rT_i <:\ {}^rT_i'}}{h \vdash \overline{{}^rT_i} <:\ \overline{{}^rT_i'}}\ \text{RTS\_\_DEF}$$

$\boxed{h \vdash v_o :\ {}^rT}$ runtime type ${}^rT$ assignable to value v_o

$$\frac{h(\iota)\downarrow_1 =_o {}^rT_1 \qquad h \vdash {}^rT_1 <:\ {}^rT}{h \vdash \iota :\ {}^rT}\ \text{RTT\_\_ADDR}$$

$$\frac{}{h \vdash \texttt{null}_a :\ {}^rT}\ \text{RTT\_\_NULL}$$

$\boxed{h, {}^r\Gamma \vdash v :\ {}^sT}$ static type ${}^sT$ assignable to value v (relative to ${}^r\Gamma$)

$$\frac{\begin{array}{c}\text{dyn}({}^sT, h, {}^r\Gamma, \overline{{}^o\iota}) =_o {}^rT \qquad h \vdash v :\ {}^rT \\ {}^sT = \texttt{self \_<\_>} \implies v = {}^r\Gamma(\texttt{this})\end{array}}{h, {}^r\Gamma \vdash v :\ {}^sT}\ \text{RTSTE\_\_DEF}$$

$\boxed{h, {}^r\Gamma \vdash \overline{v} :\ \overline{{}^sT}}$ static types $\overline{{}^sT}$ assignable to values $\overline{v}$ (relative to ${}^r\Gamma$)

$$\frac{\overline{h, {}^r\Gamma \vdash v_i :\ {}^sT_i}}{h, {}^r\Gamma \vdash \overline{v_i} :\ \overline{{}^sT_i}}\ \text{RTSTSE\_\_DEF}$$

$\boxed{h, \iota \vdash v_o :\ {}^sT}$ static type ${}^sT$ assignable to value v_o (relative to ι)

$$\frac{\begin{array}{c}{}^r\Gamma = \{\emptyset\ ;\ \texttt{this} \mapsto \iota\} \\ h, {}^r\Gamma \vdash v :\ {}^sT\end{array}}{h, \iota \vdash v :\ {}^sT}\ \text{RTSTA\_\_DEF}$$

$\boxed{\text{dyn}({}^sT, h, {}^r\Gamma, \overline{{}^o\iota}) =_o {}^rT_o}$ dynamization of static type (relative to ${}^r\Gamma$)

$$\frac{{}^{r}\Gamma = \left\{\overline{X_l \mapsto {}^{r}T_l}\ ;\ \texttt{this} \mapsto \iota, \_\right\} \qquad h \vdash \iota :\ {}^{o}\iota\ C\texttt{<}\overline{{}^{r}T}\texttt{>}}{{}^{o}\iota \in \mathrm{dom}(h) \cup \{\texttt{root}_a\} \qquad \mathrm{ClassDom}(C) =_o \overline{X} \qquad {}^{s}T\left[{}^{o}\iota\,/\,\texttt{self}\ ,\ {}^{o}\iota\,/\,\texttt{peer}\ ,\ \iota\,/\,\texttt{rep}\ ,\ \texttt{any}_a\,/\,\texttt{any}\ ,\ \overline{{}^{r}T}\,/\,\overline{X}\ ,\ \overline{{}^{r}T_l\,/\,X_l}\ ,\ \overline{{}^{o}\iota_i\,/\,\texttt{lost}}\right] =_o {}^{r}T'}{\mathrm{dyn}({}^{s}T, h, {}^{r}\Gamma, \overline{{}^{o}\iota_i}) =_o {}^{r}T'}\ \textsc{dyne}$$

$\boxed{\mathrm{dyn}\left(\overline{{}^{s}T}, h, {}^{r}\Gamma, \overline{\overline{{}^{o}\iota}}\right) =_o \overline{{}^{r}T_o}}$ dynamization of static types (relative to ${}^{r}\Gamma$)

$$\frac{\mathrm{dyn}({}^{s}T_1, h, {}^{r}\Gamma, \overline{{}^{o}\iota_1}) =_o {}^{r}T_1 \quad , \quad .. \quad , \quad \mathrm{dyn}({}^{s}T_n, h, {}^{r}\Gamma, \overline{{}^{o}\iota_n}) =_o {}^{r}T_n}{\mathrm{dyn}({}^{s}T_1\ ,\ ..\ ,\ {}^{s}T_n, h, {}^{r}\Gamma, \overline{{}^{o}\iota_1}\ \texttt{;}\ ..\ \texttt{;}\ \overline{{}^{o}\iota_n}) =_o {}^{r}T_1\ ,\ ..\ ,\ {}^{r}T_n}\ \textsc{dynse\_\_def}$$

$\boxed{h, {}^{r}\Gamma \vdash {}^{s}N, {}^{s}T; \left(\overline{{}^{s}T}/\overline{X}, \iota\right) =_o {}^{r}\Gamma'}$ validate and create new viewpoint ${}^{r}\Gamma'$

$$\frac{{}^{s}\Gamma \vdash {}^{s}N\ \ \mathsf{OK} \qquad \mathrm{ClassDom}(\mathrm{ClassOf}({}^{s}N)) =_o \overline{X} \qquad \mathrm{free}({}^{s}T) \subseteq \overline{X}\ ,\ \overline{X_l} \qquad \mathrm{dyn}\left(\overline{{}^{s}T_l}, h, {}^{r}\Gamma, \emptyset\right) =_o \overline{{}^{r}T_l} \qquad {}^{r}\Gamma' = \left\{\overline{X_l \mapsto {}^{r}T_l}\ ;\ \texttt{this} \mapsto \iota, \_\right\}}{h, {}^{r}\Gamma \vdash {}^{s}N, {}^{s}T; \left(\overline{{}^{s}T_l}/\overline{X_l}, \iota\right) =_o {}^{r}\Gamma'}\ \textsc{nvp\_\_def}$$

$\boxed{{}^{r}\Gamma \vdash h, e \rightsquigarrow h', v}$ big-step operational semantics

$$\frac{}{{}^{r}\Gamma \vdash h, \texttt{null} \rightsquigarrow h, \texttt{null}_a}\ \textsc{os\_\_null}$$

$$\frac{{}^{r}\Gamma(x) =_o v}{{}^{r}\Gamma \vdash h, x \rightsquigarrow h, v}\ \textsc{os\_\_var}$$

$$\frac{\mathrm{dyn}({}^{s}T, h, {}^{r}\Gamma, \emptyset) =_o {}^{r}T \qquad \mathrm{ClassOf}({}^{r}T) = C \qquad \left(\forall f \in \mathrm{fields}(C)\,.\,\overline{fv}(f) =_o \texttt{null}_a\right) \qquad h + \left({}^{r}T, \overline{fv}\right) = (h', \iota)}{{}^{r}\Gamma \vdash h, \texttt{new}\ {}^{s}T\texttt{()} \rightsquigarrow h', \iota}\ \textsc{os\_\_new}$$

$$\frac{{}^{r}\Gamma \vdash h, e_0 \rightsquigarrow h', \iota_0 \qquad h'(\iota_0\,.\,f) =_o v}{{}^{r}\Gamma \vdash h, e_0\,.\,f \rightsquigarrow h', v}\ \textsc{os\_\_read}$$

$$\frac{{}^{r}\Gamma \vdash h, e_0 \rightsquigarrow h_0, \iota_0 \qquad {}^{r}\Gamma \vdash h_0, e_1 \rightsquigarrow h_1, v \qquad h_1[\iota_0\,.f\ \texttt{=}\ v] =_o h'}{{}^{r}\Gamma \vdash h, e_0\,.f\ \texttt{=}\ e_1 \rightsquigarrow h', v}\ \textsc{os\_\_write}$$

$$\frac{{}^{r}\Gamma \vdash h, e_0 \rightsquigarrow h_0, \iota_0 \qquad {}^{r}\Gamma \vdash h_0, \overline{e_q} \rightsquigarrow h_1, \overline{v_q} \qquad \mathrm{MBody}(h_0, \iota_0, m) =_o e \qquad \mathrm{MSig}(h_0, \iota_0, m) =_o \_\ \texttt{<}\overline{X_l\ \texttt{extends}\ \_}\texttt{>}\ \_\ m\texttt{(}\overline{\_\ pid}\texttt{)} \qquad \mathrm{dyn}\left(\overline{{}^{s}T_l}, h, {}^{r}\Gamma, \emptyset\right) =_o \overline{{}^{r}T_l} \qquad {}^{r}\Gamma' = \left\{\overline{X_l \mapsto {}^{r}T_l}\ ;\ \texttt{this} \mapsto \iota_0\ ,\ \overline{pid \mapsto v_q}\right\} \qquad {}^{r}\Gamma' \vdash h_1, e \rightsquigarrow h', v}{{}^{r}\Gamma \vdash h, e_0\ .\ m\texttt{<}\overline{{}^{s}T_l}\texttt{>(}\overline{e_q}\texttt{)} \rightsquigarrow h', v}\ \textsc{os\_\_call}$$

$$\frac{{}^{r}\Gamma \vdash h, e \rightsquigarrow h', v \qquad h', {}^{r}\Gamma \vdash v :\ {}^{s}T}{{}^{r}\Gamma \vdash h, \texttt{(}{}^{s}T\texttt{)}\ e \rightsquigarrow h', v}\ \textsc{os\_\_cast}$$

$\boxed{{}^{r}\Gamma \vdash h, \overline{e} \rightsquigarrow h', \overline{v}}$ sequential big-step operational semantics

$$\frac{{}^{r}\Gamma \vdash h, e \rightsquigarrow h_0, v \qquad {}^{r}\Gamma \vdash h_0, \overline{e_i} \rightsquigarrow h', \overline{v_i}}{{}^{r}\Gamma \vdash h, e\ ,\ \overline{e_i} \rightsquigarrow h', v\ ,\ \overline{v_i}}\ \textsc{oss\_\_def}$$

$$\frac{}{{}^{r}\Gamma \vdash h, \emptyset \rightsquigarrow h, \emptyset}\ \textsc{oss\_\_empty}$$

$\boxed{\vdash P \rightsquigarrow h, v}$ big-step operational semantics of a program

$$\frac{\forall f \in \mathrm{fields}(C)\,.\,\overline{fv}(f) =_o \texttt{null}_a \qquad \emptyset + \left(\texttt{root}_a\ C\texttt{<>}, \overline{fv}\right) = (h_0, \iota_0) \qquad {}^{r}\Gamma_0 = \{\emptyset\ ;\ \texttt{this} \mapsto \iota_0\} \qquad {}^{r}\Gamma_0 \vdash h_0, e \rightsquigarrow h, v}{\vdash \overline{Cls},\ C,\ e \rightsquigarrow h, v}\ \textsc{osp\_\_def}$$

$\boxed{h,\ \iota \vdash {}^{r}T_o\ \ \mathsf{strictly\ OK}}$ strictly well-formed runtime type ${}^{r}T_o$

$$\frac{{}^{o}\iota \in \mathrm{dom}(h) \cup \{\mathtt{any}_a\,,\ \mathtt{root}_a\} \qquad \mathrm{ClassBnds}\big(h, \iota, {}^{o}\iota\ C\texttt{<}\overline{{}^{r}T_k}\texttt{>}, \emptyset\big) =_o \overline{{}^{r}T'_k} \qquad h,\ \_ \vdash \overline{{}^{r}T_k}\ \ \mathsf{strictly\ OK} \qquad h \vdash \overline{{}^{r}T_k} <:\ \overline{{}^{r}T'_k}}{h,\ \iota \vdash {}^{o}\iota\ C\texttt{<}\overline{{}^{r}T_k}\texttt{>}\ \ \mathsf{strictly\ OK}}\ \textsc{swfrt\_\_def}$$

$\boxed{h,\ \iota \vdash \overline{{}^{r}T}\ \ \mathsf{strictly\ OK}}$ strictly well-formed runtime types

$$\frac{\overline{h,\ \iota \vdash {}^{r}T_i\ \ \mathsf{strictly\ OK}}}{h,\ \iota \vdash \overline{{}^{r}T_i}\ \ \mathsf{strictly\ OK}}\ \textsc{swfrtso\_\_def}$$

$\boxed{h,\ \bar{\iota} \vdash \overline{{}^{r}T}\ \ \mathsf{strictly\ OK}}$ strictly well-formed runtime types

$$\frac{\overline{h,\ \iota_i \vdash {}^{r}T_i\ \ \mathsf{strictly\ OK}}}{h,\ \overline{\iota_i} \vdash \overline{{}^{r}T_i}\ \ \mathsf{strictly\ OK}}\ \textsc{swfrts\_\_def}$$

$\boxed{h \vdash \iota\ \ \mathsf{OK}}$ well-formed address

$$\frac{h(\iota)\downarrow_1 =_o \_\ C\texttt{<\_>} \qquad h,\ \iota \vdash h(\iota)\downarrow_1\ \ \mathsf{strictly\ OK} \qquad \mathtt{root}_a \in \mathrm{owners}(h, \iota) \qquad \forall f \in \mathrm{fields}(C)\,.\ \exists\, {}^{s}T.\ (\mathrm{FType}(h, \iota, f) =_o {}^{s}T\ \wedge\ h, \iota \vdash h(\iota\,.\,f)\ :\ {}^{s}T)}{h \vdash \iota\ \ \mathsf{OK}}\ \textsc{wfa\_\_def}$$

$\boxed{h\ \ \mathsf{OK}}$ well-formed heap

$$\frac{\forall \iota \in \mathrm{dom}(h)\,.\ h \vdash \iota\ \ \mathsf{OK}}{h\ \ \mathsf{OK}}\ \textsc{wfh\_\_def}$$

$\boxed{h, {}^{r}\Gamma : {}^{s}\Gamma\ \ \mathsf{OK}}$ runtime and static environments correspond

$$\frac{\begin{array}{l}{}^{r}\Gamma = \big\{\overline{X_l \mapsto {}^{r}T_l}\ ;\ \mathtt{this} \mapsto \iota\,,\ \overline{pid \mapsto v_q}\big\} \\ {}^{s}\Gamma = \big\{\overline{X_l \mapsto {}^{s}N_l}\,,\ \overline{X'_k \mapsto \_}\ ;\ \mathtt{this} \mapsto \mathtt{self}\ C\texttt{<}\overline{X'_k}\texttt{>}\,,\ \overline{pid \mapsto {}^{s}T_q}\big\} \\ h\ \ \mathsf{OK} \qquad {}^{s}\Gamma\ \ \mathsf{OK} \qquad h,\ \iota \vdash \overline{{}^{r}T_l}\ \ \mathsf{strictly\ OK} \\ \mathrm{dyn}\big(\overline{{}^{s}N_l}, h, {}^{r}\Gamma, \emptyset\big) =_o \overline{{}^{r}T'_l} \qquad h \vdash \overline{{}^{r}T_l} <:\ \overline{{}^{r}T'_l} \\ h, {}^{r}\Gamma \vdash \iota :\ \mathtt{self}\ C\texttt{<}\overline{X'_k}\texttt{>} \qquad h, {}^{r}\Gamma \vdash \overline{v_q} :\ \overline{{}^{s}T_q}\end{array}}{h, {}^{r}\Gamma : {}^{s}\Gamma\ \ \mathsf{OK}}\ \textsc{wfrse\_\_def}$$

Bibliography

[1] M. Abi-Antoun and J. Aldrich. Compile-time views of execution structure based on ownership. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2007.

[2] M. Abi-Antoun and J. Aldrich. Ownership domains in the real world. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2007.

[3] M. Abi-Antoun and J. Aldrich. Static extraction of sound hierarchical runtime object graphs. In *Types in Language Design and Implementation (TLDI)*, 2009.

[4] R. Agarwal and S. D. Stoller. Type inference for parameterized race-free Java. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 2937 of *LNCS*, pages 149–160. Springer-Verlag, 2004.

[5] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools, Second Edition*. Addison-Wesley, 2007.

[6] J. Aldrich. *Using Types to Enforce Architectural Structure*. PhD thesis, University of Washington, 2003.

[7] J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *LNCS*, pages 1–25. Springer-Verlag, 2004.

[8] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 311–330. ACM Press, 2002.

[9] J. Aldrich, R. J. Simmons, and K. Shin. SASyLF: An educational proof assistant for language theory. In *Functional and Declarative Programming in Education*, 2008.

[10] P. S. Almeida. Balloon types: Controlling sharing of state in data types. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, pages 32–59. Springer-Verlag, 1997.

[11] P. S. Almeida. *Control of Object Sharing in Programming Languages*. PhD thesis, Imperial College London, 1998.

[12] C. Andrea, Y. Coady, C. Gibbs, J. Noble, J. Vitek, and T. Zhao. Scoped types and aspects for real-time systems. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 4067 of *LNCS*, pages 124–147. Springer-Verlag, 2006.

[13] A. Banerjee and D. A. Naumann. Representation independence, confinement, and access control. In *Principles of Programming Languages (POPL)*, pages 166–177. ACM Press, 2002.

[14] A. Banerjee and D. A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. Technical Report 2004-14, Stevens Institute of Technology, 2004.

[15] A. Banerjee, D. A. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 5142 of *LNCS*, pages 387–411. Springer-Verlag, 2008.

[16] M. Bär. Practical runtime Universe type inference. Master's thesis, Department of Computer Science, ETH Zurich, 2006.

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

[18] M. Barnett, D. A. Naumann, W. Schulte, and Q. Sun. 99.44% pure: Useful abstractions in specification. In *Formal Techniques for Java-like Programs (FTfJP)*, pages 51–60, 2004.

[19] P. Bazzi. Integration of Universe type system tools into Eclipse. Semester project, Department of Computer Science, ETH Zurich, 2006.

[20] G. M. Bierman, M. J. Parkinson, and A. M. Pitts. An imperative core calculus for Java and Java with effects. Technical Report 563, University of Cambridge Computer Laboratory, 2003.

[21] A. Birka and M. D. Ernst. A practical type system and language for reference immutability. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM Press, 2004.

[22] B. Bokowski and J. Vitek. Confined types. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 82–96. ACM Press, 1999.

[23] C. Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, MIT, 2004.

[24] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 211–230. ACM Press, 2002.

[25] C. Boyapati, R. Lee, and M. Rinard. Safe runtime downcasts with ownership types. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2003.

[26] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Principles of Programming Languages (POPL)*, pages 213–223. ACM Press, 2003.

[27] C. Boyapati, A. Salcianu, W. Beebee Jr., and M. Rinard. Ownership types for safe region-based memory management in real-time Java. In *Programming language design and implementation (PLDI)*, pages 324–337. ACM Press, 2003.

[28] J. Boyland. Alias burying: Unique variables without destructive reads. *Software—Practice and Experience*, 31(6):533–553, 2001.

[29] J. Boyland, J. Noble, and W. Retert. Capabilities for aliasing: A generalisation of uniqueness and read-only. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 2072 of *LNCS*, pages 2–27. Springer-Verlag, 2001.

[30] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. In *Formal Methods for Industrial Critical Systems (FMICS)*, volume 80 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 73–89. Elsevier, 2003.

[31] L. Burdy, M. Huisman, and M. Pavlova. Preliminary design of BML: A behavioral interface specification language for Java bytecode. In *Fundamental Approaches to Software Engineering (FASE)*, 2007.

[32] L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: A developer-oriented approach. In *Formal Methods (FME)*, volume 2805 of *LNCS*, pages 422–439. Springer-Verlag, 2003.

[33] N. Cameron. *Existential Types for Variance — Java Wildcards and Ownership Types*. PhD thesis, Imperial College London, 2009.

[34] N. Cameron and W. Dietl. Comparing Universes and Existential Ownership Types. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2009.

[35] N. Cameron and S. Drossopoulou. Existential Quantification for Variant Ownership. In *European Symposium on Programming Languages and Systems (ESOP)*, 2009.

[36] N. Cameron, S. Drossopoulou, and E. Ernst. A model for Java with wildcards. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 5142 of *LNCS*, pages 2–26. Springer-Verlag, 2008.

[37] N. Cameron, S. Drossopoulou, J. Noble, and M. Smith. Multiple ownership. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 441–460. ACM Press, 2007.

[38] N. Cameron and J. Noble. OGJ Gone Wild. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2009.

[39] R. Cartwright and M. Fagan. Soft typing. In *Programming Language Design and Implementation (PLDI)*, pages 278–292. ACM Press, 1991.

[40] R. Cartwright and M. Felleisen. Program verification through soft typing. *ACM Computing Surveys*, 28(2):349–351, 1996.

[41] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 519–538. ACM Press, 2005.

[42] T. Chen. Extending MultiJava with generics. Master's thesis, Iowa State University, 2004.

[43] Y. Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, Iowa State University, 2003.

[44] Y. Cheon and G. T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In *Software Engineering Research and Practice (SERP)*, pages 322–328. CSREA Press, 2002.

[45] D. G. Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, 2001.

[46] D. G. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 292–310. ACM Press, 2002.

[47] D. G. Clarke, S. Drossopoulou, J. Noble, and T. Wrigstad. Tribe: A simple virtual class calculus. In *Aspect-Oriented Software Development (AOSD)*, 2007.

[48] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM Press, 1998.

[49] D. G. Clarke, M. Richmond, and J. Noble. Saving the world from bad beans: deployment-time confinement checking. In *Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, pages 374–387. ACM Press, 2003.

[50] D. G. Clarke and T. Wrigstad. External uniqueness is unique enough. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 2743 of *LNCS*, pages 176–200. Springer-Verlag, 2003.

[51] C. Clifton, T. Millstein, G. T. Leavens, and C. Chambers. MultiJava: Design rationale, compiler implementation, and applications. *Transactions on Programming Languages and Systems (TOPLAS)*, 28(3), 2006.

[52] D. R. Cok. Adapting JML to generic types and Java 1.6. In *Specification and Verification of Component-Based Systems (SAVCBS)*, pages 27–34, 2008.

[53] D. R. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS)*, volume 3362 of *LNCS*, pages 108–128. Springer-Verlag, 2004.

[54] D. R. Cok and G. T. Leavens. Extensions of the theory of observational purity and a practical design for JML. In *Specification and Verification of Component-Based Systems (SAVCBS)*, pages 43–50, 2008.

[55] D. Cunningham, W. Dietl, S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers. Universe types for topology and encapsulation. In *Formal Methods for Components and Objects (FMCO)*, volume 5382 of *LNCS*, pages 72–112. Springer-Verlag, 2008.

[56] D. Cunningham, S. Drossopoulou, and S. Eisenbach. Universe Types for Race Safety. In *Verification and Analysis of Multi-threaded Java-like Programs (VAMP)*, pages 20–51, 2007.

[57] D. L. Detlefs, K. R. M. Leino, and G. Nelson. Wrestling with rep exposure. SRC Research Report 156, Digital Systems Research Center, 1998.

[58] K. K. Dhara and G. T. Leavens. Preventing cross-type aliasing for more practical reasoning. Technical Report Nr. 01-02a, Department of Computer Science, Iowa State University, 2001.

[59] W. Dietl, S. Drossopoulou, and P. Müller. Formalization of Generic Universe Types. Technical Report 532, Department of Computer Science, ETH Zurich, 2006.

[60] W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 4609 of *LNCS*, pages 28–53. Springer-Verlag, 2007.

[61] W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In *International Workshop on Foundations and Developments of Object-Oriented Languages (FOOL/WOOD)*, 2007.

[62] W. Dietl and P. Müller. Exceptions in ownership type systems. In *Formal Techniques for Java-like Programs (FTfJP)*, pages 49–54, 2004.

[63] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, 2005.

[64] W. Dietl and P. Müller. 2007 State of the Universe Address. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2007.

[65] W. Dietl and P. Müller. Runtime Universe type inference. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2007.

[66] W. Dietl and P. Müller. Ownership type systems and dependent classes. In *Foundations of Object-Oriented Languages (FOOL)*, 2008.

[67] W. Dietl, P. Müller, and A. Poetzsch-Heffter. A type system for checking applet isolation in Java Card. In *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS)*, volume 3362 of *LNCS*, pages 129–150. Springer-Verlag, 2004.

[68] W. Dietl, P. Müller, and D. Schregenberger. Universe Type System — Quick-Reference. 2008.

[69] S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers. A unified framework for verification techniques for object invariants. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 5142 of *LNCS*, pages 412–437. Springer-Verlag, 2008.

[70] B. Emir, A. J. Kennedy, C. Russo, and D. Yu. Variance and generalized constraints for C♯ generics. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 4067 of *LNCS*, pages 279–303. Springer-Verlag, 2006.

[71] E. Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999.

[72] E. Ernst. Family polymorphism. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 2072 of *LNCS*, pages 303–326. Springer-Verlag, 2001.

[73] E. Ernst, K. Ostermann, and W. R. Cook. A virtual class calculus. In *Principles of programming languages (POPL)*, pages 270–282. ACM Press, 2006.

[74] M. D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington, Department of Computer Science and Engineering, 2000.

[75] M. D. Ernst. Static and dynamic analysis: Synergy and duality. In *Workshop on Dynamic Analysis (WODA)*, pages 24–27, 2003.

[76] M. D. Ernst. Type annotations specification (JSR 308). `http://types.cs.washington.edu/jsr308/`, 2008.

[77] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, 2007.

[78] M. Fähndrich and R. DeLine. Adoption and focus: practical linear types for imperative programming. In *Programming Language Design and Implementation (PLDI)*, pages 13–24. ACM Press, 2002.

[79] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Programming Language Design and Implementation (PLDI)*, pages 234–245. ACM Press, 2002.

[80] M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*, pages 241–269. Springer-Verlag, 1999.

[81] J. N. Foster and D. Vytiniotis. A theory of Featherweight Java in Isabelle/HOL. In G. Klein, T. Nipkow, and L. Paulson, editors, *Archive of Formal Proofs*. `http://afp.sf.net`, 2006.

[82] A. Fürer. Combining runtime and static Universe type inference. Master's thesis, Department of Computer Science, ETH Zurich, 2007.

[83] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[84] V. Gasiunas, M. Mezini, and K. Ostermann. Dependent classes. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 133–152. ACM Press, 2007.

[85] V. Gasiunas, M. Mezini, and K. Ostermann. vcn - a calculus for multidimensional virtual classes, 2007. `www.st.informatik.tu-darmstadt.de/static/pages/projects/mvc/index.html`.

[86] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, third edition, 2005.

[87] D. Graf. Implementing purity and side effect analysis for Java programs. Semester project, Department of Computer Science, ETH Zurich, 2006.

[88] D. Greenfieldboyce and J. S. Foster. Type qualifier inference for Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 321–336. ACM Press, 2007.

[89] P. J. Guo, J. H. Perkins, S. McCamant, and M. D. Ernst. Dynamic inference of abstract types. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 255–265. ACM Press, 2006.

[90] C. Haack, E. Poll, J. Schäfer, and A. Schubert. Immutable objects for a Java-like language. In *Programming Languages and Systems*, volume 4421 of *LNCS*, pages 347–362. Springer-Verlag, 2007.

[91] T. Hächler. Static fields in the Universe type system. Semester project, only available in German, Department of Computer Science, ETH Zurich, 2004.

[92] T. Hächler. Applying the Universe type system to an industrial application. Master's thesis, Department of Computer Science, ETH Zurich, 2005.

[93] J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. Parkinson. Behavioral interface specification languages. Technical Report CS-TR-09-01, School of EECS, University of Central Florida, 2009.

[94] S. Herrmann. Gradual encapsulation. *Journal of Object Technology (JOT)*, 7(9):47–68, 2008.

[95] S. Herrmann, C. Hundt, and M. Mosconi. ObjectTeams/Java language definition version 1.2 (OTJLD). `http://www.objectteams.org/def/1.2/`, 2009.

[96] D. Hirschkoff, T. Hirschowitz, D. Pous, A. Schmitt, and J.-B. Stefani. Component-oriented programming with sharing: Containment is not ownership. In *Generative Programming and Component Engineering*, volume 3676 of *LNCS*, pages 389–404. Springer-Verlag, 2005.

[97] J. Hogg. Islands: Aliasing protection in object-oriented languages. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 271–285. ACM Press, 1991.

[98] J. Hogg, D. Lea, A. Wills, D. de Champeaux, and R. Holt. The Geneva convention on the treatment of object aliasing. *OOPS Messenger, Report on ECOOP'91 workshop W3*, 3(2):11–16, 1992.

[99] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, 2001.

[100] A. Igarashi and M. Viroli. Variant path types for scalable extensibility. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 113–132. ACM Press, 2007.

[101] ISO/IEC standard 23270:2006. Programming language C♯. `http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html`, 2006.

[102] B. Jacobs. Weakest precondition reasoning for Java programs with JML annotations. *Journal of Logic and Algebraic Programming*, 58:61–88, 2004.

[103] B. Jacobs, F. Piessens, K. R. M. Leino, and W. Schulte. Safe concurrency for aggregate objects with invariants. In *Software Engineering and Formal Methods (SEFM)*, pages 137–147. IEEE Computer Society, 2005.

[104] B. Jacobs and E. Poll. A logic for the Java modeling language JML. In *Fundamental Approaches to Software Engineering (FASE)*, volume 2029 of *LNCS*, pages 284–299. Springer-Verlag, 2001.

[105] I. T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *Formal Methods (FM)*, 2006.

[106] N. Kellenberger. Static Universe type inference. Master's thesis, Department of Computer Science, ETH Zurich, 2005.

[107] A. Kennedy and D. Syme. Design and Implementation of Generics for the .NET Common Language Runtime. In *Programming Language Design and Implementation (PLDI)*, pages 1–12. ACM Press, 2001.

[108] M. Klebermaß. An Isabelle formalization of the Universe type system. Master's thesis, Technical University Munich and ETH Zurich, 2007.

[109] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. Technical Report 0400001T.1, National ICT Australia, 2004. `http://www4.informatik.tu-muenchen.de/~nipkow/pubs/Jinja/`.

[110] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *Transactions on Programming Languages and Systems (TOPLAS)*, 28(4):619–695, 2006.

[111] N. Krishnaswami and J. Aldrich. Permission-based ownership: encapsulating state in higher-order typed languages. In *Programming language design and implementation (PLDI)*, pages 96–106. ACM Press, 2005.

[112] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[113] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06-rev27, Iowa State University, Department of Computer Science, 2004.

[114] G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 19(2):159–189, 2007.

[115] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, D. M. Zimmerman, and W. Dietl. JML reference manual. 2008.

[116] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *LNCS*, pages 491–516. Springer-Verlag, 2004.

[117] K. R. M. Leino, P. Müller, and A. Wallenburg. Flexible immutability with frozen objects. In *Verified Software: Theories, Tools, and Experiments (VSTTE)*, volume 5295 of *LNCS*, pages 192–208. Springer-Verlag, 2008.

[118] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):121–141, 1979.

[119] Y. D. Liu and S. Smith. Pedigree types. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2008.

[120] Y. Lu and J. Potter. A type system for reachability and acyclicity. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *LNCS*, pages 479–503. Springer-Verlag, 2005.

[121] Y. Lu and J. Potter. On ownership and accessibility. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 4067 of *LNCS*, pages 99–123. Springer-Verlag, 2006.

[122] Y. Lu and J. Potter. Protecting representation with effect encapsulation. In *Principles of programming languages (POPL)*, pages 359–371. ACM Press, 2006.

[123] Y. Lu, J. Potter, and J. Xue. Validity invariants and effects. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 4609 of *LNCS*, pages 202–226. Springer-Verlag, 2007.

[124] F. Lyner. Runtime Universe type inference. Master's thesis, Department of Computer Science, ETH Zurich, 2005.

[125] K. Ma and J. S. Foster. Inferring aliasing and encapsulation properties for Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 423–440. ACM Press, 2007.

[126] O. L. Madsen and B. Møller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM Press, 1989.

[127] O. Mallo. MultiJava, JML, and generics. Semester project, Department of Computer Science, ETH Zurich, 2006.

[128] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, first edition, 1988.

[129] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

[130] M. Meyer. Interaction with ownership graphs. Semester project, Department of Computer Science, ETH Zurich, 2005.

[131] A. Milanova. Static inference of Universe Types. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2008.

[132] N. Mitchell. The runtime structure of object ownership. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 4067 of *LNCS*, pages 74–98. Springer-Verlag, 2006.

[133] S. E. Moelius and A. L. Souter. An object ownership inference algorithm and its application. In *Mid-Atlantic Student Workshop on Programming Languages and Systems (MASPLAS)*, pages 6.1–6.9, 2004.

[134] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, Fernuniversität Hagen, 2001.

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

[136] P. Müller. Reasoning about object structures using ownership. In *Verified Software: Theories, Tools, Experiments (VSTTE)*, volume 4171 of *LNCS*, pages 93–104. Springer-Verlag, 2007.

[137] P. Müller and A. Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In *Programming Languages and Fundamentals of Programming*, pages 131–140. Fernuniversität Hagen, 1999. Technical Report 263.

[138] P. Müller and A. Poetzsch-Heffter. A type system for controlling representation exposure in Java. In *Formal Techniques for Java Programs (FTfJP)*. Technical Report 269, Fernuniversität Hagen, 2000.

[139] P. Müller and A. Poetzsch-Heffter. A type system for checking applet isolation in Java Card. In *Formal Techniques for Java Programs*, 2001.

[140] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.

[141] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular specification of frame properties in JML. *Concurrency and Computation: Practice and Experience*, 15:117–154, 2003.

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

[143] P. Müller and A. Rudich. Ownership transfer in Universe Types. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 461–478. ACM Press, 2007.

[144] Y. Müller. Testcases for the Universe type system compiler. Project assistant, Summer 2004.

[145] S. Nägeli. Ownership in design patterns. Master's thesis, Department of Computer Science, ETH Zurich, 2006.

[146] D. A. Naumann. Observational purity and encapsulation. *Theoretical Computer Science*, 376:205–224, 2007.

[147] M. Niklaus. Static Universe type inference using a SAT-solver. Master's thesis, Department of Computer Science, ETH Zurich, 2006.

[148] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[149] J. Noble. Visualising objects: Abstraction, encapsulation, aliasing, and ownership. In *Software Visualization*, volume 2269 of *LNCS*, pages 607–610. Springer-Verlag, 2002.

[150] J. Noble, J. Vitek, and J. M. Potter. Flexible alias protection. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 1445 of *LNCS*. Springer-Verlag, 1998.

[151] P. Nonava. A Universe type checker using JSR 308. Semester project, Department of Computer Science, ETH Zurich, 2009.

[152] N. Nystrom, V. Saraswat, J. Palsberg, and C. Grothoff. Constrained types for object-oriented languages. In *Object-oriented Programming Systems Languages, and Applications (OOPSLA)*, pages 457–474. ACM Press, 2008.

[153] M. Odersky. *The Scala Language Specification, Version 2.7*. Programming Methods Laboratory, EPFL, Switzerland, 2008.

[154] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 2743 of *LNCS*, pages 201–224. Springer-Verlag, 2003.

[155] J. Östlund, T. Wrigstad, D. G. Clarke, and B. Åkerblom. Ownership, uniqueness, and immutability. In *Objects, Components, Models and Patterns*, volume 11 of *Lecture Notes in Business Information Processing*, pages 178–197. Springer-Verlag, 2008.

[156] M. Ottiger. Runtime support for generics and transfer in Universe types. Master's thesis, Department of Computer Science, ETH Zurich, 2007.

[157] M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 201–212, 2008.

[158] A. Poetzsch-Heffter, K. Geilmann, and J. Schäfer. Infering ownership types for encapsulated object-oriented program components. In *Program Analysis and Compilation, Theory and Practice*, volume 4444 of *LNCS*, pages 120–144. Springer-Verlag, 2007.

[159] A. Poetzsch-Heffter and J. Schäfer. Modular specification of encapsulated object-oriented components. In *Formal Methods for Components and Objects*, volume 4111 of *LNCS*, pages 313–341. Springer-Verlag, 2006.

[160] A. Poetzsch-Heffter and J. Schäfer. A representation-independent behavioral semantics for object-oriented components. In *Formal Methods for Open Object-Based Distributed Systems*, volume 4468 of *LNCS*, pages 157–173. Springer-Verlag, 2007.

[161] A. Potanin. *Generic Ownership: A Practical Approach to Ownership and Confinement in Object-Oriented Programming Languages*. PhD thesis, Victoria University of Wellington, 2007.

[162] A. Potanin, J. Noble, D. G. Clarke, and R. Biddle. Generic ownership for generic Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 311–324. ACM Press, 2006.

[163] A. Potanin, J. Noble, D. G. Clarke, and Biddle R. Featherweight generic confinement. In *Foundations of Object-Oriented Languages (FOOL)*, 2004.

[164] J. Quinonez, M. S. Tschantz, and M. D. Ernst. Inference of reference immutability. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 5142 of *LNCS*, pages 616–641. Springer-Verlag, 2008.

[165] D. Rayside, L. Mendel, and D. Jackson. A dynamic analysis for revealing object ownership and sharing. In *Workshop on Dynamic Analysis (WODA)*, pages 57–64. ACM Press, 2006.

[166] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. *Logic in Computer Science*, page 55, 2002.

[167] C. Saito, A. Igarashi, and M. Viroli. Lightweight family polymorphism. *Journal of Functional Programming*, 18:285–331, 2007.

[168] A. Salcianu and M. C. Rinard. Purity and side effect analysis for Java programs. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 3385 of *LNCS*, pages 199–215. Springer-Verlag, 2005.

[169] A. Schaad. Universe type system for Eiffel. Semester project, Department of Computer Science, ETH Zurich, 2006.

[170] J. Schäfer and A. Poetzsch-Heffter. A parameterized type system for simple loose ownership domains. *Journal of Object Technology (JOT)*, 6(5):71–100, 2007.

[171] J. Schäfer and A. Poetzsch-Heffter. CoBoxes: Unifying active objects and structured heaps. In *Formal Methods for Open Object-Based Distributed Systems*, volume 5051 of *LNCS*, pages 201–219. Springer-Verlag, 2008.

[172] J. Schäfer, M. Reitz, J.-M. Gaillourdet, and A. Poetzsch-Heffter. Linking programs to architectures: An object-oriented hierarchical software model based on boxes. In *The Common Component Modeling Example*, volume 5153 of *LNCS*, pages 238–266. Springer-Verlag, 2008.

[173] D. Schneider. Testing tool for compilers. Semester project, Department of Computer Science, ETH Zurich, 2007.

[174] D. Schregenberger. Runtime checks for the Universe type system. Semester project, Department of Computer Science, ETH Zurich, 2004.

[175] D. Schregenberger. Universe type system for Scala. Master's thesis, Department of Computer Science, ETH Zurich, 2007.

[176] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strniša. Ott: effective tool support for the working semanticist. In *International Conference on Functional Programming (ICFP)*, pages 1–12. ACM Press, 2007.

[177] M. Skoglund. Sharing objects by read-only references. In *Algebraic Methodology and Software Technology (AMAST)*, volume 2422 of *LNCS*, pages 457–472. Springer-Verlag, 2002.

[178] M. Skoglund. *Investigating Object-Oriented Encapsulation in Theory and Practice*. Licentiate thesis, Stockholm University/Royal Institute of Technology, 2003.

[179] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *European Conference on Object-Oriented Programming (ECOOP)*, LNCS, pages 148–172. Springer-Verlag, 2009.

[180] M. Stock. Implementing a Universe type checker in Scala. Master's thesis, Department of Computer Science, ETH Zurich, 2008.

[181] R. Strniša and M. Parkinson. Lightweight Java. `http://www.cl.cam.ac.uk/research/pls/javasem/lj/`, 2009.

[182] A. J. Summers, S. Drossopoulou, and P. Müller. Universe-type-based verification techniques for mutable static fields and methods. *Journal of Object Technology (JOT)*, 8(4):85–125, 2009.

[183] Sun Developer Network. Secure coding guidelines for the Java programming language, version 2.0. `http://java.sun.com/security/seccodeguide.html`.

[184] A. Suzuki. Bytecode support for the Universe type system and compiler. Semester project, Department of Computer Science, ETH Zurich, 2005.

[185] M. S. Tschantz and M. D. Ernst. Javari: Adding reference immutability to Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 211–230. ACM Press, 2005.

[186] C. von Praun and T. R. Gross. Object race detection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 70–82. ACM Press, 2001.

[187] P. Wadler. Linear types can change the world! In M. Broy and C. B. Jones, editors, *Programming Concepts and Methods (PROCOMET)*, 1990.

[188] D. Wellenzohn. Implementation of a Universe type checker in ESC/Java2. Semester project, Department of Computer Science, ETH Zurich, 2005.

[189] A. Wren. Inferring ownership. Master's thesis, Department of Computing, Imperial College, 2003.

[190] T. Wrigstad. *Ownership-Based Alias Management*. PhD thesis, Royal Institute of Technology, Sweden, 2006.

[191] T. Wrigstad and D. G. Clarke. Existential owners for ownership types. *Journal of Object Technology*, 6(4):141–159, 2007.

[192] H. Yan, D. Garlan, B. R. Schmerl, J. Aldrich, and R. Kazman. Discotect: A system for discovering architectures from running systems. In *International Conference on Software Engineering (ICSE)*, pages 470–479, 2004.

[193] Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kieżun, and M. D. Ernst. Object and reference immutability using Java generics. In *European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, 2007.

[194] R. Züger. Generic Universe Types in JML. Master's thesis, Department of Computer Science, ETH Zurich, 2007.

List of Figures

List of Definitions

List of Theorems and Lemmas

Werner M. Dietl

wdietl@gmail.com
http://google.com/profiles/wdietl

Degrees

01/2003: Diplom-Ingenieur, *Salzburg University, Austria*.
Passed both diploma examinations with distinction.

08/2000: Master of Science in Computer Science
Bowling Green State University, OH, USA. Passed with 4.00 GPA.

Education

09/2003 – 07/2009: Doctoral Student
Swiss Federal Institute of Technology ETH Zurich, Switzerland.
PhD program under the supervision of Prof. Peter Müller at the Chair of Programming Methodology (until 08/2008 known as the Software Component Technology Group at the Chair of Software Engineering of Prof. Bertrand Meyer).

10/2001 – 01/2003: Study of Applied Computer Science and Business
Salzburg University, Austria.

08/1999 – 08/2000: Exchange Year
Bowling Green State University, OH, USA.

10/1996 – 08/1999: Study of Applied Computer Science and Business
Salzburg University, Austria.

1991 – 1996: Polytechnical School for Electronics and Computer Science
HTBLA Salzburg, Austria. Passed with distinction.

Employment

11/2009 – Present: Post-Doctoral Research Associate
University of Washington, Seattle, WA, USA.

09/2003 – 10/2009: Research and Teaching Assistant
Swiss Federal Institute of Technology ETH Zurich, Switzerland.

02/2003 – 08/2003: Research Assistant, *Salzburg University, Austria*.

10/2000 – 07/2001: Software Engineer, *Synapta Corporation, Palo Alto, CA, USA*.

08/1999 – 08/2000: Research Assistant, *Bowling Green State University, OH, USA*.

11/1997 – 09/2000: Software Engineer, *SBS Software Ges.m.b.H., Austria*.

Fall 1997 & 1998: Tutor, *Non-procedural Programming, Salzburg University, Austria*.

07/1997 – 09/1997: Software Engineer, *Siemens AG, Austria*.

Summer 1993 & 1995: Student internships at local companies.

Awards and Honors

06/2009: Fellowship for Prospective Researchers from the Swiss NSF.

03/2008: IDEA League Short-Term Research Grant from ETH Zurich.

09/2004: Excellent Diploma Thesis Award from the Austrian Computer Society OCG.

07/1999: Excellence Scholarship from Salzburg University.

06/1999: Travel Grant from the joint-study program, Salzburg University.

02/1999: Bonus for suggested improvement from Siemens AG, Austria.

12/1995: Grant for Students from the Austrian Federal Economic Chamber Salzburg.

End of Document