Containerless Plurals: Separating Number from Type in Object-Oriented Programming

FRIEDRICH STEIMANN, Fernuniversität in Hagen

To let expressions evaluate to no or many objects, most object-oriented programming languages require the use of special constructs that encode these cases as single objects or values. While the requirement to treat these standard situations idiomatically seems to be broadly accepted, I argue that its alternative, letting expressions evaluate to any number of objects directly, has several advantages that make it worthy of consideration. As a proof of concept, I present a core object-oriented programming language, dubbed Num, which separates number from type so that the type of an expression is independent of the number of objects it may evaluate to, thus removing one major obstacle to using no, one, and many objects uniformly. Furthermore, Num abandons null references, replaces the nullability of reference types with the more general notion of countability, and allows methods to be invoked on any number of objects, including no object. To be able to adapt behavior to the actual number of receivers, Num complements instance methods with plural methods, that is, with methods that operate on a number of objects jointly and that replace static methods known from other languages. An implementation of Num in Prolog and accompanying type and number safety proofs are presented.

CCS Concepts: • **Software and its engineering → Object oriented languages**; **Data types and structures**; **Semantics**; • **Theory of computation → Object oriented constructs;**

Additional Key Words and Phrases: Multiplicities in programming, collections, bunches, null-safety, object-relational programming

ACM Reference format:
Friedrich Steimann. 2022. Containerless Plurals: Separating Number from Type in Object-Oriented Programming. *ACM Trans. Program. Lang. Syst.* 44, 4, Article 21 (July 2022), 56 pages.
https://doi.org/10.1145/3527635

1 INTRODUCTION

The evolution of programming languages is marked by the introduction of abstractions suited to rid programming of accidental complexity, that is, of the complexity imposed by the underlying computing machinery rather than the subject of programs [10]. One such complexity, the different coding idioms required when dealing with many or no objects instead of one, has largely resisted abstraction in programming languages, even though other languages, notably modeling languages, handle none, one, and many uniformly [57].

From a traditional viewpoint [40], the different encoding of none, one, and many objects is natural: in programs, objects are represented by expressions, and expressions are regarded as functions, meaning that unless they are partial, they evaluate to precisely one object. In this functional view, there is no place for no or many objects—both must be encoded as special objects (including null)

Author's address: F. Steimann, Fernuniversität in Hagen, 58084 Hagen, Germany; email: steimann@acm.org.

0164-0925/2022/07-ART21
https://doi.org/10.1145/3527635

or containers (tuples, arrays, or collections), whose handling leads to the said coding idioms. Uniformly viewing expressions as relations, on the other hand, would mean that they could evaluate to any number of objects, including one and none.

While some programming languages, functional ones in particular, have come some way in reducing the programming overhead required for handling no or many objects (using, e.g., maybe and list monads [65] or streams [4]), their solutions introduce an indirection that has its own price tag. Specifically, these solutions usually mean (1) a change of type from the object type to the container (or wrapper) type, demoting the content type to a subscript (parameter) of the container type, and (2) the necessity of dealing with the introduced indirection, for instance by wrapping and unwrapping objects. It is instructive to note how this parallels the introduction of pointer types and the creation and dereferencing of pointers for the implementation of dynamic and recursive data structures, which was largely abstracted away by object-oriented programming languages such as Smalltalk and Java, apparently not to their detriment.

It follows that while the case analyses required to handle none, one, and many may take on more convenient forms, they still exist in programs, whereas a language-level abstraction would remove them entirely and hide them in the language implementation. There are pros and cons to both approaches: while the former grants greater flexibility, the latter lowers the accidental complexity of standard cases to zero. In this present work, I pursue the latter, and explore a relational, rather than functional, interpretation of expressions that caters for the uniform treatment of none, one, and many. For this, I let expressions evaluate to any number of objects, without these objects being contained in any way; in my words, I let them evaluate to *containerless plurals*.

Contribution. With this work, I deliver the theoretical underpinnings of my earlier works [56, 57, 59], which laid out the general ideas of programming with numbers of objects, first by inverting the relationship between container and content [56, 59], and later by dropping containers entirely [57]. In the present work, I show that, as previously only claimed, with the strict separation of type and number, two largely independent matters of programming can be set apart at the language level: *objects being of the right type* vs. *objects being in the right number*. I will do this by formally introducing a core object-oriented programming language, dubbed Num, which features *number* as a mode that is *parallel to, rather than encoded in, type*, and which is supported by notions of static number checking and number safety. From an object-oriented programming perspective, number extends the *nullability* of reference types to *countability*; at the same time, introduction of a number specifier *one* (which was previously only proposed [57]) makes sure that values (i.e., instances of value types, such as integers and booleans) can remain uncountable and, specifically, do not become nullable. In addition, Num distinguishes ordinary, or *singular*, methods which, when invoked (in one call) on a number of objects, are executed on each one of them (where as usual, the receivers are accessible in the bodies of the methods through `this`, a singular), from the newly introduced *plural methods* which, when invoked on a number of objects (including none), are executed precisely once and which can access their receivers collectively through `these`, a plural. Plural methods allow direct encodings of certain programming patterns, including methods that respond specifically to being invoked on no object.

Embedding. This work is part of a larger effort aimed at making object-oriented programming more relational. It is motivated by the observation that in software modeling, objects are commonly linked through relations rather than fields or attributes (functions), and that contemporary mainstream object-oriented programming languages (with the exception perhaps of the .NET languages with their language-integrated querying [36]) rely on complex and brittle frameworks for bridging the two worlds [50]. While a full integration of relations and object-oriented programming would require bi-directionality and coverage of higher degree relations, this work

focuses on a prerequisite of such a venture: It introduces the handling of plurals through typed and numbered to-many pointers.

Organization. I begin in Section 2 with a motivating example showcasing some of the fundamental disparities programmers must live with when dealing with one object, on the one side, and many objects, on the other. In Section 3, I introduce the notion of *numbers of objects* that is central to this work, first by motivating it ontologically and then by making it precise. Section 4 provides the intuition behind programming with numbers of objects and shows how it serves the elimination of the disparities identified in Section 2. Section 5 then formally defines the language Num, sketches its implementation in Prolog, and shows how its largely independent type and number systems make Num programs both type and number safe. The design decisions that shaped Num are critically discussed in Section 6, which also reproduces some of the findings of an earlier case study [59]. Section 7 discusses a significant portion of the abundant work by others that is related to, and has influenced, mine. The full specification of the semantics of Num and the proof of its type and number safety are deferred to the appendix.

2 MOTIVATION

As noted in the introduction, this work is motivated by the long-term goal of replacing functions with relations in object-oriented programming. It can hence be viewed from a relational and an object-oriented perspective.

The Relational Perspective. While object-oriented programming is largely functional in character (a field maps an object to, and a method returns, no or one object), object-oriented modeling, and data modeling in general, are typically relational: objects (or entities) are connected through relations (or relationships, or associations), which map one object in one place of a relation to a number of objects in another. To capture the specifics of a domain, relations can be specialized to (partial) functions by so-called *mapping constraints* [35] which, for binary relations, have the form N:1, 1:N, or 1:1, meaning that a relation is in fact a (partial) function mapping an object to at most one object in each place constrained by 1. In this light, object-oriented programming as we know it is a specialization of a more general object-relational programming paradigm in which all relations are constrained to binary, uni-directional, and N:1 (the latter because generally, an arbitrary number of objects can relate to, through a field or method, any one object).

Navigation of relational data relies on a join operator which, in an object-oriented setting, corresponds to field access (which, in case of ternary or higher-degree relations, may be qualified with other objects) [32]. However, while relational join is generally agnostic of mapping constraints (an N:M relation is navigated through the same expression as an N:1 relation), this is not the case in object-oriented programming: here, to implement a (unidirectional) N:M relation, the implementing field must hold a collection, and collection-valued fields are generally navigated differently (using loops or mapping) than single-object valued fields. The same holds for updating: while updating a relation in a relational language uniformly corresponds to adding or removing tuples, updating fields depends again on whether the field is prepared to hold no or one object, or any number of objects contained in a collection. While seamlessly integrating the navigation and updating of relations in object-oriented programming would require a unified handling of no, one, and many objects, it turns out that such a unification stands beautifully on its own: it corresponds to the introduction of the plural to a paradigm that so far offered only the singular for expression.

The Object-Oriented Programming Perspective. The distinction between one and many objects is a fundamental one that can be found in every non-trivial program. One would therefore expect that the support of object-oriented programming languages for this distinction is declarative, or in

```
1  class Subject {
2    final IObserver observer;
3    Subject(IObserver o) {
4      assert o != null;
5      observer = o;
6      IObserver backup = o;
7      o = null;
8      if (···) o = backup;
9    }
10   IObserver getObserver() {
11     return observer;
12   }
13   void notify() {
14     observer.update(this);
15   }
16 }
17 interface IObserver {
18   void update(Subject s);
19   Collection<Element> getElems();
20 }
21 class Observer
22   implements IObserver {···}
23 class Element {···}
24
25 Observer o = new Observer();
26 Subject s = new Subject(o);
27 Collection<Element> es =
28   s.getObserver().getElems();
```

```
1  class Subject {
2    final Collection<IObserver> observers;
3    Subject(Collection<? extends IObserver> os) {
4      assert !os.isEmpty();
5      observers = new ArrayList<>(os);
6      Collection<? extends IObserver> backup = os;
7      os.clear();
8      if (···) os = backup;
9    }
10   Collection<IObserver> getObservers() {
11     return java.util.Collections.
12       unmodifiableCollection(observers);
13   }
14   void notify() {
15     for (IObserver o : observers)
16       o.update(this);
17   }
18 }
19 interface IObserver ···
20 class Observer ···
21 class Element ···
22
23 Collection<Observer> os =
24   new Arrays.asList(new Observer());
25 Collection<Subject> ss =
26   new Arrays.asList(new Subject(os));
27 Collection<Element> es = ss.stream()
28   .flatMap(e -> e.getObservers().stream())
29   .flatMap(e -> e.getElems().stream())
30   .toList();
```

Fig. 1. Left: Implementation of the Observer pattern with a single observer per subject. Right: Same program as left, with many observers (and many subjects in the main code).

other ways sufficiently abstract. However, this is not the case—instead, the necessary differences in code for handling one and many objects are marked by technical detail and appear surprisingly ad hoc. That this is indeed the case is demonstrated by the example of an implementation of the Observer pattern in Java as shown in Figure 1 which, although somewhat contrived, exacerbates nine marked differences between a subject having one (the singular case, left) and many (the plural case, right) observers:

(1) *Number Determines Type.* In line 25 of Figure 1, left, we have the declaration of a variable `o` capable of holding a single object of type `Observer` and its initialization to an instance of that type. If we want the variable to be capable of holding many objects, we cannot just write `Collection<Observer> os = new Observer()` instead, as this would give us a type mismatch error; rather, we need to wrap the new object in a collection first, as in line 24, right. Also, on one object, we can invoke methods directly, as in line 14, left, whereas for many, we have to unwrap them from the collection first, as in line 15, right.

(2) *Number Governs Subtyping.* To a variable declared to hold a single object of a given type, subtyping means that we can assign it objects of the type's subtypes, as is exploited by the method call in line 26, left (which assigns an actual parameter having type `Observer` to the formal parameter of line 3, having type `IObserver`). For variables declared to hold many objects, the same is not possible: in line 3, the formal parameter type must explicitly be

made covariant, thus preventing that objects are added to the collection through the formal parameter (which is an alias to the actual parameter).[1]

(3) *Number Governs Encoding of "no Object".* For a variable capable of holding a single object, that it holds no object is encoded by the value `null`, which may or may not be admissible in the domain (see line 4, left, for a case where it is not); in either case, `null` may also mean that the variable has not been initialized. For a variable capable of holding many objects, the condition that it holds no object is expressed by an empty collection rather than a null pointer (line 4, right); here, the value `null` universally means that the variable has not been initialized.

(4) *Number Governs Null-Safety.* Closely connected is the fact that technically, the safety of the method call in line 14, left, depends on the assertion of line 4—if `observer` where optional, the call would need to be guarded by a test for *not null*. By contrast, for the call on `observers` in line 15 f., right, a corresponding guard is not needed: the case of no observer is gracefully covered by the for-loop. Note that both sides could be aligned by letting `observer` have type `Optional` and using a stream protocol, in Java by calling `stream().forEach(e->e.update(this))` on both `observer` (singular, left) and `observers` (plural, right); however, in the given example (in which `observer` is not meant to be optional) it would introduce an indirection through a container that is hard to justify.

(5) *Number Shapes Chaining.* While number shapes member access as sketched above, the differences in chained member access between singulars and plurals are even greater. For instance, leaving null-safety aside (which, strictly speaking, affects both sides, as long as variables holding containers can also hold `null`), while the simple expression of line 28, left, suffices to retrieve all elements connected to the subject `s` via its (sole) observer, the same expression of line 27 f., right, is significantly more complex (and would remain so even if the more convenient syntax of language-integrated querying [36] were used). Note again that the situation could be aligned by using the stream protocol on the left also, but the price of this alignment, complex code on both sides, seems rather high.

(6) *Number Dictates Encapsulation Strategy.* Whereas it is standard practice that fields holding single objects can be read and written through getters (line 11, left) and setters or constructors (line 5), collections, especially if representing relationships to many objects, are usually considered representation objects [44] and are therefore to be protected by copying, as in lines 5 and 11 f., right.

(7) *Number Governs Sharing.* Through the indirection that comes with the use of containers, two variables can be made to share the same storage location. An example of this is given in lines 6–8, right, where after the assignment of line 6, the variables `os` and `backup` refer to the same collection. Changing which objects `os` refers to in line 7 therefore affects `backup` as well, rendering the assignment in line 8 logically ineffective. This is not so for the singular case on the left, which does not use an indirection through a container (and therefore avoids sharing). For plurals, sharing can be avoided by using explicit copying of collections, which amounts to copying the pointers to many objects, paralleling the copying of the single pointer that is implicit for the singular case.

(8) *Number Governs Call Semantics.* While in Java and many other object-oriented programming languages, method calls are by-value, due to the sharing through containers noted above, when collections are passed calls are effectively by-reference. For instance, the constructor invocation `new Object(o)` of line 26, left, is by-value, meaning that the assignment of `null` to

[1] Note that, while wildcard types may be specific to Java, covariance of containers is generally incompatible with writing into them [62].

the formal parameter of the constructor in line 7 of the body of the constructor has no effect on the actual parameter (regardless of the value of the condition guarding the assignment; see below). This is different for the invocation `new Object(os)` in line 26, right: here, the invocation of `clear()` on the formal parameter (which likewise means setting to no object; cf. above) in line 7 of the body of the constructor affects the actual parameter, which is also cleared.

(9) *Number Determines the Meaning of the* `final` *Modifier.* Even though both the field `observer` and the field `observers` are declared `final` in class `Subject` (line 2, left, and line 2, right), the observers of a subject (right) can freely be changed (only the collection cannot be replaced), whereas the observer (left) cannot.

Surely, some of these disparities can be leveled out in other object-oriented languages with different typing disciplines: for instance, Scala has declaration-site variance so that using its covariantly typed immutable collections reduces or bypasses the above semantic differences 2 and 6–9, and languages with implicit coercions can address difference 1 to some degree. Other language features (such as type classes) allow the reduction of syntactic differences, and the use of higher order functions even for primitive operations such as assignment or member access may push all differences to a library. However, rather than trying to camouflage the difference between one and many using mechanisms generally suited for language extension, the purpose of this work is to acknowledge its fundamental nature, by giving it a central place in the core language, as a (new) mode *number* complementing *type* in declarations.

3 NUMBERS OF OBJECTS

Before I introduce, in Section 4, *numbers of objects* as a programming construct suited to rid programs of the discontinuities identified in Section 2, I invite the reader to a brief ontological excursion intended to give a deeper rationale for some of my design decisions, and introduce the language of plurals on which my formal capture of the language Num in Section 5 depends.

3.1 Ontological Considerations

3.1.1 Containerless Plurals. When Georg Cantor conceived sets, he spoke of them as "jedes Viele, welches sich als Eines denken läßt" [13, p. 587], i.e., as "any many which can be thought of as a one" (my translation). Indeed, a mathematical set reifies a number of (mathematical) objects as one object, making the set subjectable to much of the same mathematical machinery as its members. Before this reification, many objects were just that: many objects.

While sets have proved extremely useful, not only in mathematics, we sometimes need to talk about many objects outside of sets or some other sort of container. For instance, the arithmetic equation $x^2 = 4$ has two solutions, -2 and 2, but the solution is not the set $\{-2, 2\}$, nor is it any other single mathematical object—in fact, there is no *one* solution. Mathematically, this plurality is explained by the fact that the inverse of x^2 is a relation mapping 4 to -2 *and* 2 (and not a function mapping 4 to $\{-2, 2\}$); we note here that -2 *and* 2 is different from $\{-2, 2\}$ and, more generally, that the contents of a set are not a set (see Section 7.1.1 for a discussion of Eric Hehner's *theory of bunches* [27], which is rooted in the same observation).

In programming, occurrences of many objects are usually reified as one using some sort of collection. Just like the introduction of the null pointer, from the programming language perspective this is very convenient, because it ensures that every expression evaluates to precisely one value —sorting out the differences is left to programs. At the program text level, however, many objects may occur, for instance, as arguments to functions or in array initializers; and yet, these plurals (occurrences of many) are typically not first-class citizens of a programming language, and if they

are, they are usually forced into some kind of container (typically a list or an array; e.g., the var args of Java).

Conceptually, forcing occurrences of many objects into a container just for the sake of being able to handle them seems awkward: for instance, I have a wife, two parents, and three children—I do not have a pair of parents, nor a set of children, and harmonizing the situation by saying that I am married to a singleton elevates weirdness to the norm. What is a numerical difference, or *difference in number*, in natural and also many modeling languages (including Entity-Relationship Diagrams [15] and UML [45]; cf. Section 7.5) is a *difference in type and degree of indirection* in programming languages. Arguably, this is a complexity imposed by the language, or an "accidental complexity" [10].

3.1.2 Number as a Grammatical Feature. In the English language, like in many others, the difference between one and many manifests itself in the distinction between singular and plural forms, where singular and plural are expressions of the grammatical category, or feature, *number*. In English, number is a feature of nouns, pronouns, and verbs, and English grammar requires that the subject of a sentence (a noun or a pronoun) and its predicate (a verb) must have the same number. Plural subjects can be constructed from singular nouns using *and* (as in "−2 and 2" above) and some other conjunctions; conversely, many (logical) subjects can be represented by a collective noun, whose number is singular (as in "the set of −2 and 2").

While programming languages do not provide for plurals,[2] or "any many that are *not* thought of as a one", the metalanguages of programming languages make ample use of them: the notations a^*, $a_1 \ldots a_n$, and $\overline{a}$ are ubiquitous in formal language specifications, where they stand for a number of items of the kind of a, that is, for a plural (also referred to as "unenclosed sequence" [55]; cf. Section 7.1.2). It seems reasonable to expect that what is useful for a metalanguage is useful for its object languages also.

3.1.3 Countables and Uncountables. The linguistic distinction between singular and plural is not universal—it applies only to the so-called *countables*. While the conditions for countability vary from language to language and usually involve context, as a rule of thumb, countability requires identity—only things that have identity we can count, and only of those we can have many. For instance, when we say that $x^2 = 4$ has two solutions, this presupposes that we can identify each solution (and that the two solutions are different). Uncountables, on the other hand, are not identifiable; we cannot refer, for instance, to two rains. Yet, identity, and hence countability, can be added to uncountables by wrapping: for instance, we can refer to two occasions of rain.

Programming languages may also distinguish things that have identity and things that do not. For instance, the Java programming language distinguishes between primitive values and reference values [2], where reference values are usually equated with the identities of objects, something primitive values do not have.[3] While regarding numbers, characters, and other primitive values as uncountable[4] may seem ill-devised, we already have this situation in all object-oriented programming languages in which only reference types are nullable—for these languages, *countability is a generalization of nullability*, namely, from covering zero or one to covering any number.

[2]One might argue that programming languages do not come with grammatical categories at all; however, *type* may be viewed as one.

[3]In fact, even in Smalltalk, the language in which "everything is an object", small integers and characters are usually implemented as primitive values ("immediates") rather than (references to) objects, sometimes leading to confusing behavior (small numbers are identical if they are equal, while big numbers are not; note that equating the value of numbers with their identity does not help here). Generally, different languages make different distinctions, but few (if any) manage to fully unify objects and values [58].

[4]Countable and uncountable are not to be confused with countably many and uncountably many.

Consistently extending nullability to primitive (or value) types already requires a significant reworking of language semantics, as demonstrated by its introduction to C♯ [39], which involved the adaptation of all operations on primitive values to cover the absence of a value (cf. Section 7.2), and extending nullability of primitive types to countability will almost inevitably lead to array programming, which is however not containerless (see Section 7.2.5). Hence, in this present work, I adopt the distinction of countables and uncountables in that I do not cater for numbers of primitive values, unless they are given identities through boxing (cf. Section 6.2).[5]

3.1.4 Linguistic Note. Dealing with many objects without making (implicit) use of containers is a mental exercise. To invite the reader to participate in this exercise, I use the term "number of objects" for containerless plurals, or occurrences of any number of objects that are not thought of as a one, basically because the word "number" is not a collective noun like "bunch" [27], "sequence" [55], or "multitude" [59]. As part of the exercise, all uses of the term "number of objects" in this writing have the (grammatical) number plural, so as to make clear that the term is *not* used collectively (that the many are not treated as a one). When I use the single word "number", I always mean the grammatical category *number*, with expressions yet to be introduced; to denote the result of counting objects (the "dynamic number"), I will use the word "count". The word "number" has the advantage over the word "multiplicity" (which I used in prior works [56, 57, 59]) of being both a noun and a verb, allowing me to use it in analogy to "type"; in particular, I will use "numbering" to refer to the assignment of numbers to expressions and to number checking, and I will use "well-numbered" to express that whatever this term attributes, abides by the rules of numbering.

3.2 Metalanguage for Numbers of Objects

To write about numbers of objects N, I use the grammar

$$N ::= \epsilon \mid o\,N\,,$$

where o ranges over single, countable objects and ϵ denotes zero objects, or the absence of an object ("no object"). I write $o_1 \ldots o_n$ for n objects o_1 through o_n, where the o_i must be unique (pairwise non-identical) and where n may be zero (in which case $o_1 \ldots o_n \equiv \epsilon$). I call n the *count* of $o_1 \ldots o_n$. Note that I use juxtaposition without a separator to denote numbers of more than one object; this is to avoid the impression that numbers of objects are lists in disguise (with the separator as the list constructor). Also, note that the occurrence of a single object counts as a number of objects, namely, as the special case in which the count is 1. For disambiguation, I sometimes parenthesize numbers of objects; however, one cannot have numbers of numbers of objects—there is no nesting of numbers of objects, and zero or more than one objects are not an object.[6]

Numbers of objects are similar to strings, one difference being that no object may occur (and hence can count) twice (where singularity is judged by object identity). While this restriction may appear arbitrary, it is justified by the above ontological motivation and by the fact that numbers of objects are used to specify relations (see the beginning of Section 2, and Section 3.2.2).

3.2.1 Operations on Numbers of Objects. On numbers of objects $N = o_1 \ldots o_n$, I define the unary operator $|o_1 \ldots o_n| := n$ (meaning that $|N|$ is the count of N). Furthermore, I define the binary operator $\propto$, read as "among", as

$$N \propto o_1 \ldots o_n \quad \Leftrightarrow \quad N = \epsilon \;\vee\; N = o\,N' \wedge \exists 1 \le i \le n \,.\, o = o_i \wedge N' \propto o_1 \ldots o_n.$$

[5]Note that in many object-oriented programming languages including Java, boxing is a prerequisite to storing primitive values in containers other than arrays; in these languages, making the boxing of primitive values a prerequisite for having numbers of them would be consistent with their design.

[6]Readers acquainted with Hehner's bunches [26, 27] I refer to Section 7.1.1 for a discussion of analogies and differences.

To construct numbers of objects from numbers of objects, I define the binary operators *object addition* ($\oplus$) and *object subtraction* ($\ominus$) as

$$N_1 \oplus N_2 \quad := \quad \begin{cases} N_1 & \text{if } N_2 = \epsilon \\ N_1 \oplus N_2' & \text{if } N_2 = o\,N_2' \text{ and } o \propto N_1 \\ (N_1\,o) \oplus N_2' & \text{if } N_2 = o\,N_2' \text{ and } o \not\propto N_1 \end{cases}$$

and

$$N_1 \ominus N_2 \quad := \quad \begin{cases} N_1 & \text{if } N_1 = \epsilon \\ N_1' \ominus N_2 & \text{if } N_1 = o\,N_1' \text{ and } o \propto N_2\,, \\ o\,(N_1' \ominus N_2) & \text{if } N_1 = o\,N_1' \text{ and } o \not\propto N_2 \end{cases}$$

respectively. Note that ϵ is the neutral element of object addition, meaning that $N_1 \oplus \ldots \oplus N_n = \epsilon$ for $n = 0$.

3.2.2 Using Numbers of Objects for Encoding Functions and Relations. As noted in Section 3.1.1, relations can be viewed as mappings from one to a number of objects: for instance, the relation $\{\,(x, y) \mid x = y^2\,\}$ maps 4 to -2 and 2. Functions are then mappings to a number of objects whose count is constrained to 1 (total functions) or ≤ 1 (partial functions), where mapping to ϵ means that the function is undefined for the argument (see also Hehner [26, p. 29] for a corresponding use of bunches).

3.2.3 Ordering of Numbers of Objects. Even though neither "containerless plural" nor "number of objects" suggests an ordering of the denoted objects, as syntactic entities, numbers of objects are necessarily ordered (due to the linear nature of textual syntax, which is preserved operationally by the above specifications of $\oplus$ and $\ominus$). Because orderedness eliminates a source of nondeterminacy in programming with numbers of objects, and because orderedness is a prerequisite to using numbers of objects for expressing "unenclosed sequences" in the specification of Num (i.e., as constructs of the metalanguage), I will henceforth assume that numbers of objects are ordered.

4 PROGRAMMING WITH NUMBERS OF OBJECTS

Before formally defining the language Num in Section 5, I provide some basic intuition of programming with numbers of objects, by guiding the reader through a couple of examples. The examples use the syntax of Num as shown in Figure 2. In a nutshell, the reader may assume that a small core of Java has been extended with number declarations and expressions evaluating to numbers of objects, governed by static numbering that complements (standard) static typing. Specifically:

- In declarations of variables, fields, and methods, reference (or class) types C ("countables", as opposed to the value types `bool` and `int`, referred to as "uncountables") are paired with a number specifier η.[7] Type and number specifier together are referred to as a *range* ρ. Note that for uncountables, the range does not comprise a number specifier.
- The number specifiers of Num are: `!`, read as *one* and meaning exactly one object; `?`, read as *optional* and meaning no or one object; `*`, read as *many* (or *any*) and meaning any number of objects; and `-`, read as *none* and meaning no object.
- On number specifiers the reader may assume a partial order $\leq$ as the smallest reflexive and transitive relation that includes `-` $\leq$ `?`, `!` $\leq$ `?`, and `?` $\leq$ `*` (see Table 1(a) for a depiction); together with the subtype relation, this partial order governs assignment compatibility.

[7] I use Greek eta (η), rather than nu (ν), for *number* to avoid confusion with Latin v, which, beginning in Section 5.3, I will use for values.

P	::=	$K_1 \ldots K_n$	program
K	::=	`class` C `extends` C' `{`$F_1 \ldots F_n\ M_1 \ldots M_{n'}$`}`	class definition
F	::=	$\rho\ f$ `=` ie`;`	field definition
M	::=	$\rho\ m\ \eta$ `(`V_1`,` $\ldots$ `,`V_p`)` `{`$s_1 \ldots s_n$ `return` e`;}`	method definition
V	::=	$\rho\ x$	variable declaration
ρ	::=		**ranges:**
		`bool` | `int`	uncountables
	|	$C\ \eta$	countables
η	::=		**numbers:**
		`-`	none
	|	`!`	one
	|	`?`	optional (none or one)
	|	`*`	many (or any number)
s	::=		**statements:**
		`if (`e`) {`$s_1 \ldots s_n$`} else {`$s'_1 \ldots s'_{n'}$`}`	if-then-else
	|	`while (`e`) {`$s_1 \ldots s_n$`}`	while
	|	$\texttt{this}^C.f$ `=` e`;`	field assignment
	|	${e_0}^C.m$`(`e_1`,` $\ldots$ `,`e_p`);`	method invocation
	|	x `=` e`;`	variable assignment
e	::=		**expressions:**
		ie	initialization expression
	|	x	variable access
	|	`this` | `these`	receiver access
	|	$\texttt{this}^C.f$	field access
	|	${e_0}^C.m$`(`e_1`,` $\ldots$ `,`e_n`)`	method application
	|	e_1 `+`$^\kappa$ e_2	addition
	|	e_1 `-`$^\kappa$ e_2	subtraction
	|	e_1 `==` e_2	test for equality or identity
	|	`(`C`)` e	type cast
	|	`(`η`)` e	number cast
	|	`|`e`|`	count
	|	`(`e`)`	parenthesized expression
ie	::=		**initialization expressions:**
		`true` | `false`	booleans
	|	i	integers
	|	`no` C	no object
	|	`new` C	new object
κ	::=	int | obj	**kinds**

Fig. 2. Syntax of Num. The metavariables C, m, f, x, and their derivatives C' and so on, range over class names, method names, field names, and variable names, respectively. Superscripts C and κ are worked out by typing.

— In method definitions, the name of a method is followed by a number specifier η, where `!` declares a *singular method* and `*` declares a *plural method*. Note that the number specifier is not part of the method name; instead, for the names of plural methods, plural verb forms should be considered.[8]

[8]The method number specifier may actually be viewed as a *receiver annotation*, i.e., as an annotation of `this` (which becomes `these` for plural methods; see below).

— Num has no `void` type; that nothing is returned by a method is declared using a range whose number specifier is `-` (often `Object-`).
— Num has no `null` keyword; it is replaced by the expression `no` *C*, meaning "no object of class *C*".
— This is complemented by `these`; the former is reserved for singular, the latter for plural methods. In plural methods, `these` denotes the number of objects that make the receiver.
— Num has no constructors. Instead, all fields are initialized, upon creation of an object (using `new` *C*), to the values of initialization expressions provided at their definition site.

Note that the metalanguage used in Figure 2 for the specification of Num's syntax includes the language of numbers of objects as introduced in Section 3.2, accounting for "unenclosed sequences" [55] (cf. Section 7.1.2) and improvising the insertion of separators in places as usual. For instance, $K_1 \ldots K_n$ stands for a number of classes (including none).

The basic idea of programming with numbers of objects is that

— instead of evaluating to one object, expressions having a countable type evaluate to numbers of objects; that
— their count ("dynamic number") is statically constrained by a number system (analogous to how the dynamic types of the objects are constrained by a type system); and that
— occurrences of none, one, and many objects are handled alike.

For member access, this means that the access is lifted from one object to any number of objects (including none), collecting the results (if any) as a number of objects; for assignment, this means that the pointers to the number of objects an expression evaluates to are copied into the variable (field, formal parameter) on the left-hand side of the assignment (in analogy to how for standard assignment of reference values, one pointer is copied). For fields having countable types, numbering means that they are no longer interpreted as functions mapping the owner of the field to precisely one object or `null`, but as relations mapping the owner to a number of objects (including zero and one objects as special cases; cf. Section 3.2.2).

4.1 Scrapping Containers

Using Num for the motivating example from Section 2, the noted differences in handling none, one, and many objects disappear, and materialize in declarations as differences in number, as highlighted in Figure 3. Specifically:

(1) *Number Does not Determine Type.* The declarations of the fields `observer` and `observers` as well as of the temporaries `o` and `os` of Figure 1, left (lines 2 and 25) and right (lines 2 and 23), are replaced by the declarations `IObserver! observer`, `IObserver* observers`, `Observer? o`, and `Observer* os` in Figure 3, lines 2 and 24, meaning that the fields and temporaries differ in number, but not in type. The assignment `os = new Observer` (line 24, right) is therefore well-typed, as is the method invocation `observers.update(this)` (line 14, right). Hence, the difference between singular and plural has become one of declaration. Note that the assignments in line 24 are also well-numbered, since the number of the `new` expression is `!`, whereas that of the variables is `?` and `*`, and both $! \leq ?$ and $! \leq *$.
(2) *Number Does not Govern Subtyping.* Since number has been separated from type, the two assignments `this.observer = (!) o` and `this.observers = os` (lines 5 in Figure 3, where the left-hand side is a supertype of the right-hand side) are type correct. Technically, the assignment of `os` does not produce lurking dynamic type errors (comparable to the array store exception of Java or C♯), since there is no container that is aliased by the assignment and to which objects could be added.

```
1  class Subject extends Object {
2    IObserver! observer = new Observer;
3    Object- init!(IObserver? o) {
4      assert false |o| == 0;
5      this.observer = (!) o;
6      IObserver! backup = (!) o;
7      o = no Observer;
8      if (...) {o = backup;} else {}
9    }
10   IObserver! getObserver!() {
11     return this.observer;
12   }
13   Object- notify!() {
14     this.observer.update(this);
15   }
16 }
17 class IObserver extends Object {
18   Object- update!(Subject! s) {...}
19   Element* getElems!() {...}
20 }
21 class Observer extends IObserver {...}
22 class Element extends Object {...}
23
24 Observer? o = new Observer;
25 Subject! s = new Subject; s.init(o);
26 Element* es = s.getObserver().getElems();
```

```
class Subject extends Object {
  IObserver* observers = new Observer;
  Object- init!(IObserver* os) {
    assert false |os| == 0;
    this.observers = os;
    IObserver* backup = os;
    os = no Observer;
    if (...) {os = backup;} else {}
  }
  IObserver* getObservers!() {
    return this.observers;
  }
  Object- notify!() {
    this.observers.update(this);
  }
}
class IObserver extends Object {
  Object- update!(Subject! s) {...}
  Element* getElems!() {...}
}
class Observer extends IObserver {...}
class Element extends Object {...}

Observer* os = new Observer;
Subject* ss = new Subject; ss.init(os);
Element* es = ss.getObservers().getElems();
```

Fig. 3. Implementation of the two versions of the Observer pattern of Figure 1 in Num (semantic differences underlined). Left: with exactly one observer per subject. Right: Same program as left, with any number of observers (and subjects in the main code). For fairness of comparison with Java, `return no Object` has been elided from `Object-` (void) methods, and temporaries have been declared inline (not as formals, as would be required by Num; see text). `assert` is not formalized, but implemented in Num.

(3) *Encoding of "no Object" Is Universal.* Expressions evaluating to no object evaluate to just that (i.e., a number of objects with count 0), independently of their static number being ? or *. Testing an expression for "no object" is thus universally testing its count for 0, as in line 4 of Figure 3.

(4) *Null-Safety is Universal.* Since there is no difference in representation of no object between the singular and the plural case, and because accessing members on no object is safe (see above), there is no dependence of null-safety on number like that worked out in Figure 1. In addition, the ! annotation on field `observer` guarantees that in line 16, left, the method invocation has a receiver, independently of the guarantee of the assertion of line 5 (which was essential in the example of Figure 1). For the plural case (right), no such guarantee is given by numbering; this would require an additional number specifier (+, for *one or more*).

(5) *Number Does not Shape Chaining.* Just like single method invocations are syntactically indistinguishable for singular and plural receivers (see item 1 above), the shape of chained method invocations does not depend on the return number of any of the involved methods. Hence, the invocation expression of lines 27 f. in Figure 1, right, is replaced by `ss.getObservers().getElems()` which, except for the (linguistic) plural forms of the names, equals that of the corresponding expression for the singular case (Figure 3, line 26).

(6) *Encapsulation Strategy is Universal.* Because of the containerlessness and because of the copy semantics of the assignment of many objects (cf. above), there is no representation that is specific to the plural case and that needs to be protected by copying or some other means.

(7) *Non-Sharing is Universal.* For the same reasons, there is no sharing that is specific to the plural case—numbers of objects are never shared (in the sense that they are aliased as a whole; individual objects can still be aliased).
(8) *Call Semantics is Universal.* For the same reasons again, call semantics is by-value in all cases.
(9) *The Meaning of* `final` *is Universal.* Even though Num does not feature a `final` modifier, Num's containerlessness makes it clear that the number of objects held by a variable declared as `final` cannot be changed, independently of its static number.

Note that for the singular, case, the assertion in line 4, left, has become obsolete, because the cast to number `!` in the next line, which is required by number checking, has the same effect—it will fail at runtime should `o` contain no object. The cast is required by the number of `observer`, `!`, which protects the invocation of `update(this)` in line 14 from silently failing (by doing nothing). As noted above, the same does not hold for the plural case. More generally, in programs in which a variable is allowed to hold "no object", but silently doing nothing when accessing a member on the variable is not acceptable, a number cast to `!` (or `+`) can be inserted in the access expression; a number cast exception will then replace for the null pointer exception that one would get from a corresponding Java program. For instance, replacing the (chained) member access expression of Figure 3, line 26, left, with `(!)((!)s.getObserver()).getElems()` will fail if `s` or `s.getObserver()` evaluate to no object (but note that, because Num is number safe, for this to happen, `s` and `getObserver` would need to be declared with number `?`).

4.2 Plural Methods

While invoking a singular method on a number of objects means that it is invoked on each object individually, invoking a plural method means that it is invoked on the objects jointly. Specifically, in the bodies of plural methods, `these` refers to the number of objects jointly constituting the receiver of a method call, so that plural methods can implement coordinated behavior of many objects, and can let the number of objects on which a method is invoked "respond with one voice". For instance, the among operator $\propto$ from Section 3.2 is implemented by

```
class Object { bool areAmong*(Object* those) { return |these - those| == 0; } }
```

where `-` and `|·|` are Num implementations of the operators defined in Section 3.2 (note the plural verb form, `areAmong`, matching the plural subject, `these`). Since plural methods are invoked precisely once, independently of the count of objects they are invoked on, plural methods with uncountable return types can be invoked on any receiver number; for instance, `(no Object).areAmong(no Object)` is legal (and performs as expected). Singular methods with uncountable return types on the other hand can only be invoked on receivers with static number `!`.

While invocations of singular methods are dispatched individually on the dynamic type of each object among the receiving number of objects, plural methods are necessarily dispatched on the static type of the receiver expression (this is necessary because if the count of the receiver objects is 0, a dynamic type cannot be determined; for counts higher than 1, a decision would need to be made as to which method is selected, meaning that possible overridings of the method become ineffective). Plural methods are hence reminiscent of static methods in other languages, with the difference that the receiver is a number of objects (where in Java, static methods do not have a receiver, and for Smalltalk's class methods, the receiver is a class). In fact, `no C.m()` in Num is equivalent to Java's `((C) null).m()`, provided that `m` is a plural method of class `C` in Num and a static method of `C` in Java, respectively. However, while accessing static methods through objects is considered bad practice in Java, for the plural methods of Num, access through a number of objects is the only possible way. Also note that, since numbers of objects with counts different

from 1 do not constitute objects (they have no "group identity"), they have no joint state so that we cannot associate fields with numbers of objects.

Plural methods can reflect on the number of receivers a method is invoked on. This allows code like

```
Object- printNumberSensitive*() {
  if (|these| == 0) { these.printBlank(); }
  else { if (|these| == 1) { these.printYourself(); }
  else { these.printAll(); } }
  return no Object;
}
```

which is reminiscent of pattern matching on number (rather than type) and where "printBlank" and "printAll" are the names of plural methods, while "printYourself" is the name of a singular method. Note how this generalizes the Null Object pattern [21] to covering many objects also.

4.3 Binary Methods

While plural methods may be used to extend the notion of binary methods [11] to numbers of objects, as in

```
class Object {
  bool equal*(Object* those) {
    return these.areAmong(those) && those.areAmong(these);
  }
}
```

other binary methods more naturally extend to collections of objects, or "contained plurals". For instance, addition of numbers extends to vector addition, where each vector is a (singular) object in its own right. However, such operations are the domain of array programming, which is not the domain of this article (cf. the discussions in Sections 6.2 and 7.2.5).

There may still be cases in which one wants to lift a method invocation over pairs of objects formed from the cross product of two numbers of objects, where one is the receiver of the corresponding method and the other its argument. Such can be implemented in Num using double dispatching [30], as in

```
class Pair extends Object {
  Pairable! a = new Pairable; Pairable! b = new Pairable;
  Pair! set!(Pairable! a, Pairable! b) { this.a = a; this.b = b; return this; }
}
class Pairable extends Object {
  Pair* crossProduct!(Pairable* as) { return as.pairedWith(this); }
  Pair! pairedWith!(Pairable! b) { return new Pair.set(this, b); }
}
```

Here, class `Pair` provides a container for two objects, held in fields `a` and `b` having type `Pairable` and number `!`. Note that since their number is `!`, `a` and `b` must be initialized to new objects of the required type (which will later be overwritten by invoking the singular method `set` on a `Pair` object, which returns the modified receiver for convenience). Class `Pairable` provides two singular methods letting `Pairable` objects form instances of class `Pair`. The first, `crossProduct!(Pairable* bs)`, accepts many objects (a plural) as argument, and dispatches invocation of the second method, `Pair! pairedWith!(Pairable! b)`, to these objects individually, using `this` as the (singular) argument. In the body of `pairedWith`, `(this, b)` is therefore a pair of two objects. Invoking `crossProduct` on a plural, with another plural as arguments, as in the expression

`(o1 + o2).crossProduct(o3 + o4)` (where + is object addition as defined in Section 3.2), dispatches the invocation to `o1` and `o2`, each time with arguments `o3` `o4` (two objects), and yields the number of pairs (o3, o1) (o4, o1) (o3, o2) (o4, o2).

4.4 Implementing Iterable Collections with Numbers of Objects

While it would seem natural to make numbers of objects iterable, the iteration that is inherent in method invocation on numbers of objects is generally sufficient. That this is indeed the case is demonstrated by the following bridge from numbers of objects to collections (implemented as a linked list), which can then be made iterable as usual:

```
class Collectable extends Object {
  Object- collectIn!(Collection! c) { c.add(this); return no Object; }
  Collection! asCollection*() { return these.collectWith(new Collection); }
  Collection! collectWith*(Collection! c) { these.collectIn(c); return c; }
}
class Collection extends Object {
  CollectionElement? first = no CollectionElement;
  CollectionElement? last = no CollectionElement;
  Collection! add!(Collectable* os) {
    if (|os| == 1) {
      if (|this.first| == 0) {
        this.first = new CollectionElement.setElement((!) os);
        this.last = this.first;
      } else {
        this.last.setNext(new CollectionElement.setElement((!) os));
        this.last = this.last.getNext();
      }
    }
    else { os.collectIn(this); }
    return this;
  }
}
class CollectionElement extends Object {
  CollectionElement? next = no CollectionElement;
  Collectable! element = new Collectable;
  CollectionElement! setElement!(Collectable! o) { this.element = o; return this; }
  Collectable! getElement!() { return this.element; }
  CollectionElement! setNext!(CollectionElement! ce) { this.next=ce; return this; }
  CollectionElement? getNext!() { return this.next; }
}
```

This code equips `Collectable` objects with two methods, a singular method `collectIn!(Collection!)` that, when invoked on a number of objects, adds each one to the collection passed as parameter, and a plural method `asCollection*()` that, when invoked on a number of objects, uses `collectIn` to return a new collection holding the objects. `collectIn!(Collection! c)` forwards its receiver to the method `add!(Object* os)` invoked on the collection `c` to which the object is to be added. For convenience, `add` accepts any number of objects in `os`, yet in its body dynamically distinguishes between one object (detected by the condition `|os| == 1`) and other object counts. In the case of one object, it adds the object to its representation (which has the form of a linked list, with pointers to the first and last collection element), which requires a number downcast of `os` from `*` to `!`; in all other cases, it calls `collectIn` on `os`, thereby implicitly looping over all objects in `os` and (recursively) calling `add` on the collection witch each one as (singular) argument.

Note that while `Collection` would need to be generic to capture the type of its elements, plurals do not require parametric polymorphism for typing numbers of objects.

4.5 Maps, Filters, and Folds

The collection APIs of languages like Smalltalk, C♯, Scala, or Java (the latter via streams) provide not only for mapping over collections (as Num does natively for all numbers of objects), but also for filtering, folding, and other convenient higher-order operations that replace for boilerplate code. For consistency (and depending on language), some of these operations are offered for optionals also. In Num on the other hand, plurals have the type of their constituting objects, so that filters and folds must be members of (or implemented in) these types. For instance, filtering objects of type `T` is implemented in Num (enhanced with multiple return statements) by a method

```
T? filter!() { if (cond) { return this; } else { return no T; } }
```

in class `T` and by applying `filter()` to a number of objects having type `T`, which yields precisely the objects among them satisfying *cond*. Folds can be implemented similarly: for instance, given the plural method

```
int count*(Counter! c) { these.inc(c); return c.value(); }
```

e`.count(new Counter)` replaces for |*e*|, provided that there is a suitably defined class `Counter` and `Object` implements a suitably defined method `Object- inc!(Counter! c)`. More generally, higher-order functions on numbers of objects could be implemented by extending Num with mix-ins and closures; yet, the need for different, or individual, functions (as offered by different mix-ins) may be seen as indication that the number of objects treated by such functions has a nature, that is, that it is one rather than many objects (a singular rather than a plural), which should be represented by a specific, meaning-carrying collection (cf. the discussion in Section 6.2).

5 THE LANGUAGE NUM

To work out the details of introducing numbers of objects to programming, I define static and dynamic semantics for Num (with syntax shown in Figure 2 of Section 4) and describe the implementation of an interpreter of Num in Prolog. Num builds on Middleweight Java (MJ) [7], a core calculus for Java adding control structures (including blocks), updatable variables (including fields), object identity and the null pointer to Featherweight Java (FJ) [29]. Num adds primitive values and numbers of objects to MJ, but drops blocks and constructors (which do not contribute to the essence of Num), and discards null pointers. Note that, as for the specification of syntax in Figure 2, I will use the language of numbers of objects from Section 3.2 in the metalanguage of the specification of semantics also.

5.1 Preliminaries

The syntax of Num places a number of restrictions on programs that are common in core calculi such as FJ [29], MJ [7], ClassicJava [20], or Bali [42]:

- All classes defined by a program explicitly declare a superclass, which can be the predefined class `Object`, which has neither members (fields or methods) nor a superclass.
- All method bodies must be concluded by `return` *e*`;`, which, although not a statement *s* by the syntax definition, will be treated as one in all other contexts.
- The formal parameters of a method are the only local variables. Additional variables, if needed, can be added as additional parameters (which can be passed initializing arguments by wrapper methods).
- All members must be accessed through the dot operator (there is no default receiver). The superscript *C* is a placeholder for the static type of the superscripted receiver, as worked out by typing; likewise, superscript κ (with values *int* and *obj*) serves the disambiguation of the overloaded operators + and −.

— To keep the specification of Num small, fields must be accessed through `this`; they are instance private, as in Smalltalk. To access fields of other objects or numbers of objects, accessor methods must be used.

For the following specifications of the static and dynamic semantics of Num, it is assumed that its programs satisfy the following sanity constraints, which I will not formalize, but which are nevertheless necessary (together with well-typedness and well-numberedness as defined below) for a program to be well-formed:

— Class names must be unique.
— The superclass relation established by class definitions must be acyclic.
— Fields must not be hidden in subclasses (by using the same names).
— The names of the variables serving as the formal parameters of a method must be pairwise different, and must differ from "this", "these", "rec", and "ret".
— Methods must not be overloaded.

For the conditions of overriding, see Sections 5.2.1 and 5.2.2.

As usual, I also define some data structures and auxiliary functions that support the specification of the semantics of Num. Specifically, I represent type and number environments as well as heaps and variable stores as maps, where a map $\mathcal{M}$ is a number of mappings $k \mapsto w$ of keys k to values w, where the keys must be pairwise distinct. For $\mathcal{M} = (k_1 \mapsto w_1) \ldots (k_n \mapsto w_n)$, I define $dom(\mathcal{M}) := k_1 \ldots k_n$ (note the use of numbers of objects in place of sets; for *dom*, this means that it is a relation; cf. Section 3.2.2). I write $\mathcal{M}[k]$ for looking up a map $\mathcal{M}$ under key k, yielding the w that k maps to; I write $\mathcal{M}[k \mapsto w]$ for updating a map $\mathcal{M}$ at the position k, yielding a new map $\mathcal{M}'$ such that $\mathcal{M}'[k'] = w$ if $k' = k$ and $\mathcal{M}'[k'] = \mathcal{M}[k']$ otherwise. For expanding $\mathcal{M}$ with $k \mapsto w$, where $k \notin dom(\mathcal{M})$, I write $(k \mapsto w)\,\mathcal{M}$ (note again that this is genuine syntax of numbers of objects, as introduced in Section 3.2; specifically, ϵ denotes "no mappings", i.e., the empty map).

To access the various parts of the definition of a program P, I write $P[C]$ to select the class definition K `= class` C `extends` C' `{`$\cdots$`}` from P and $K[m]$ to select the method definition $\rho\ m\ \eta$ `(`$\cdots$`) {`$\cdots$`}` from K. Based on these selectors, I define lookup functions as shown in Figure 4:

— *fields*(C) and *method*$(C.m)$ retrieve the fields defined or inherited by class C (represented as a number of objects, including ϵ meaning "no field") and the method m defined or inherited by C, respectively.
— *rng*$(C.f)$ and *sig*$(C.m)$ retrieve the range ρ of field f and the signature $\eta\ \rho_1 \ldots \rho_n \mapsto \rho$ of method m defined or inherited by class C.
— Ranges ρ are projected to their components by *typ*$(\cdot)$ (selecting types) and *num*$(\cdot)$ (selecting numbers); both are overloaded to map method signatures to their type and number components, respectively. Note that for uncountables, *num*$(\cdot)$ maps ρ to η_ϵ rather than ϵ (which would express undefinedness; cf. Section 3.2.2); this reification of "no number" is required by the application of *num*$(\cdot)$ to method signatures, from which "no number" must be retrievable.
— *params*$(C.m)$ and *body*$(C.m)$ retrieve the formal parameters and the statements from the body of method m defined or inherited by class C.

Note that $P[C]$, $K[m]$, and the functions depending on them may be undefined, namely, if C or $C.m$ are not defined in P (where undefinedness is written using $\perp$ in Figure 4). Here, undefinedness is to be distinguished from evaluating to ϵ: *params*$(C.m) = \epsilon$ means that $C.m$ has no parameters, which is a regular case, whereas *params*$(C.m) = \perp$ means that $C.m$ does not exist, which may

member retrieval

$$\frac{}{fields(\texttt{Object}) = \epsilon} \qquad \frac{P[C] = \texttt{class}\ C\ \texttt{extends}\ C'\ \{\rho_1\, f_1 = ie_1; \ldots \rho_n\, f_n = ie_n; M_1\ \ldots\ M_n\}}{fields(C) = (\rho_1\, f_1 = ie_1) \ldots (\rho_n\, f_n = ie_n) \oplus fields(C')}$$

$$\frac{P[C][m] = \rho\ m\ \eta\ (V_1, \ldots, V_p)\ \{s_1\ \ldots\ s_n\}}{method(C.m) = P[C][m]} \qquad \frac{P[C][m] = \bot \quad P[C] = \texttt{class}\ C\ \texttt{extends}\ C'\ \{\cdots\}}{method(C.m) = method(C'.m)}$$

range & signature

$$\frac{fields(C) = (\rho_1\, f_1 = ie_1) \ldots (\rho\, f = ie) \ldots (\rho_n\, f_n = ie_n)}{rng(C.f) = \rho} \qquad \frac{method(C.m) = \rho\ m\ \eta\ (\rho_1\, x_1, \ldots, \rho_p\, x_p)\ \{s_1\ \ldots\ s_n\}}{sig(C.m) = \eta\ \rho_1\ \ldots\ \rho_p \mapsto \rho}$$

type & number selection

$$\frac{\rho = C\ \eta}{typ(\rho) = C} \qquad \frac{\rho = int}{typ(\rho) = int} \qquad \frac{\rho = bool}{typ(\rho) = bool} \qquad \frac{\rho = C\ \eta}{num(\rho) = \eta} \qquad \frac{\rho = int}{num(\rho) = \eta_\epsilon} \qquad \frac{\rho = bool}{num(\rho) = \eta_\epsilon}$$

$$\frac{sig(C.m) = \eta\ \rho_1\ \ldots\ \rho_p \mapsto \rho}{typ(sig(C.m)) = typ(\rho_1)\ \ldots\ typ(\rho_p)\ typ(\rho)} \qquad \frac{sig(C.m) = \eta\ \rho_1\ \ldots\ \rho_p \mapsto \rho}{num(sig(C.m)) = \eta\ num(\rho_1)\ \ldots\ num(\rho_p)\ num(\rho)}$$

parameters and body of method

$$\frac{method(C.m) = \rho\ m\ \eta\ (\rho_1\, x_1, \ldots, \rho_p\, x_p)\ \{s_1\ \ldots\ s_n\}}{params(C.m) = x_1\ \ldots\ x_p} \qquad \frac{method(C.m) = \rho\ m\ \eta\ (V_1, \ldots, V_p)\ \{s_1\ \ldots\ s_n\}}{body(C.m) = s_1\ \ldots\ s_n}$$

Fig. 4. Lookup functions. Each is defined as the smallest function satisfying its given rules. Note that some functions are partial (i.e., undefined for certain arguments; see text).

cause "stuckness" of evaluation (Section 5.5). To avoid notational clutter, P remains implicit in most specifications, here and in the following.

5.2 Static Semantics

I divide static semantics into typing and numbering. Typing includes name binding, which is required, and cannot generally be done, by numbering. Apart from this, numbering is independent from typing, as will be seen.

5.2.1 Typing and Name Binding. We first need some additional definitions. Given the syntax of types,

$$\tau ::= bool \mid int \mid C,$$

the rules defining the subtype relation are

$$\frac{}{\tau <: \tau} \qquad \frac{\texttt{class}\ C\ \texttt{extends}\ C'\ \{\cdots\} \qquad C' <: C''}{C <: C''},$$

meaning that, as usual, class names serve as types and `extends` clauses feed the subtype relation. For object addition, we also need the definition of the least common supertype (*lcs*) of two types, which is given by

$$lcs(C, C') := \begin{cases} C & \text{if } C' <: C \\ lcs(C'', C') & \text{otherwise, if } P[C] = \texttt{class}\ C\ \texttt{extends}\ C''\ \{\cdots\} \end{cases}.$$

Typing of Expressions. Typing judgments for expressions have the form $\mathcal{T} \vdash e : \tau$, where the type environment $\mathcal{T}$ is a map from variable names to types. The full set of type rules, which are

mostly standard, is given in Appendix A.1; here, I will present only those that might need extra explanation:

— The type of both `this` and `these` is kept under the same key *rec* (for receiver) in the type environment $\mathcal{T}$:

$$\textsc{T-This}\frac{\mathcal{T}[rec] = C}{\mathcal{T} \vdash \boxed{\texttt{this}} : C} \qquad \textsc{T-These}\frac{\mathcal{T}[rec] = C}{\mathcal{T} \vdash \boxed{\texttt{these}} : C}.$$

As regards typing, the uses of `this` and `these` are interchangeable; their correct use in the bodies of singular and plural methods will be controlled by numbering (see below).

— The lookup of fields and methods required by field access and method application labels the receivers with their static type, which is required by numbering (Section 5.2.2) and the static binding of plural method applications (Section 5.3). I only show the rule for method application here:

$$\textsc{T-MAppl}\frac{\mathcal{T} \vdash e_0 : C \qquad (\mathcal{T} \vdash e_i : \tau_i)_{i=1..p} \qquad typ(sig(C.m)) = \tau'_1 \ldots \tau'_p\, \tau \qquad \left(\tau_i <: \tau'_i\right)_{i=1..p}}{\mathcal{T} \vdash \boxed{e_0{}^{C}.m(e_1, \ldots, e_p)} : \tau}.$$

— Uses of the operators + and – overloaded for integers and objects are likewise disambiguated by superscripting, making sure that the left and the right operand are of the same kind κ (the rules for integers are not shown here). While adding numbers of objects gives the result type the least common supertype of the addends' types, for subtraction, it is the minuend's type (and the subtrahend's type is unconstrained):

$$\textsc{T-ObjAdd}\frac{\mathcal{T} \vdash e_1 : C_1 \qquad \mathcal{T} \vdash e_2 : C_2 \qquad C = lcs(C_1, C_2)}{\mathcal{T} \vdash \boxed{e_1 +^{obj} e_2} : C} \qquad \textsc{T-ObjSub}\frac{\mathcal{T} \vdash e_1 : C_1 \qquad \mathcal{T} \vdash e_2 : C_2}{\mathcal{T} \vdash \boxed{e_1 -^{obj} e_2} : C_1}.$$

— Type cast, number cast, and count are well-typed only for countable *e*; a number cast does not affect the type of an expression. Unlike Java and some of its core calculi, Num does not provide for explicit up-casts.

$$\textsc{T-TCast}\frac{\mathcal{T} \vdash e : C' \qquad C <: C'}{\mathcal{T} \vdash \boxed{(C)\ e} : C} \qquad \textsc{T-NCast}\frac{\mathcal{T} \vdash e : C}{\mathcal{T} \vdash \boxed{(\eta)\ e} : C} \qquad \textsc{T-Count}\frac{\mathcal{T} \vdash e : C}{\mathcal{T} \vdash \boxed{|e|} : int}.$$

Type Checking of Statements. Typing judgments for statements have the form $\mathcal{T} \vdash s_1 \ldots s_n \checkmark_\tau$, where $s_1 \ldots s_n$ is a number of statements, which I call "sequence of statements" only as a concession to custom. Again, the type rules are mostly standard, and are shown in full only in Appendix A.1; here, I show the rule for `return`, which bounds the type of the return expression with the type stored in the type environment under the entry *ret*:

$$\textsc{T-Return}\frac{\mathcal{T} \vdash e : \tau \qquad \tau <: \mathcal{T}[ret]}{\mathcal{T} \vdash \boxed{\texttt{return}\ e\texttt{;}}\ \checkmark_\tau}.$$

Note that the single statement `return` *e*`;` is subsumed under the above form of judgments for statements using numbers of statements; had the form used a conventional sequence, this would not be the case.

Type Checking of Definitions. The rules for type-checking the definitions of methods, fields, classes, and programs are again found in Appendix A.1; of these, only the rule for methods,

$$\text{T-MethDef}\ \frac{\begin{array}{c}(rec \mapsto C)(ret \mapsto typ(\rho))(x_1 \mapsto typ(\rho_1)) \ldots (x_p \mapsto typ(\rho_p)) \vdash s_1 \ \ldots \ s_n \checkmark_\tau \\ P[C] = \texttt{class}\ C\ \texttt{extends}\ C'\ \cdots \\ method(C'.m) = \bot \vee typ(sig(C'.m)) = typ(sig(C.m))\end{array}}{M = \boxed{\rho\ m\ \eta\ (\rho_1\, x_1, \ldots, \rho_p\, x_p)\ \{s_1 \ \ldots \ s_n\}}\ \checkmark_\tau \text{ in } C},$$

needs explanation:

- The typing judgment of methods, $M\checkmark_\tau$ in C, depends on the class C in which the method is defined ("in C"). The type environment $\mathcal{T}$ of method bodies is provided by *rec* (for `this` or `these`), *ret* (for the value to be returned), and the formal parameters of the method.
- Syntax dictates that the last statement in the method body, s_n, is a return statement; that its returned expression has the expected type is ensured by the type of *ret* in conjunction with T-Return (see above).
- The premises of T-MethDef require that if m is also defined for the superclass C' of C, it must have the same type signature (the override condition). Note that this applies to both singular and plural methods, thus allowing plural methods (which will be statically bound; cf. Section 4.2) to be hidden.

Note that there is no type rule that makes use of number. Typing (including name binding) is therefore independent of numbering.

5.2.2 Numbering. For the numbering of programs, I introduce a number environment $\mathcal{N}$, a numbering relation # relating expressions e to their (static) number η, and numbering judgments, defined as the analogs of the typing environment, relation, and judgments. Because expressions e having an uncountable type cannot be assigned a number, I extend the definition of η to include η_ϵ (introduced in Section 5.1 as a reification of "no number"); following Section 3.2.2, $e \,\#\, \eta_\epsilon$ means that e has no number (the numbering relation does not map e to a number), which is a defined condition. This device lets uncountable expressions be subsumed under judgments $\mathcal{N} \vdash e \,\#\, \eta$, thereby avoiding numerous case analyses based on countability.

On numbers η, Table 1(a) defines a partial order $<\#$, which I call the *subnumber relation*. Note how including η_ϵ in this relation, although somewhat ad hoc (strictly, "no number" cannot be a subnumber of "no number"), covers uncountability. The table also defines (in similarly ad hoc ways including η_ϵ) how (static) numbers combine for singular method invocation as well as adding and subtracting numbers of objects, as will be needed for the numbering of corresponding expressions (see below).

Numbering of Expressions. As for typing, the full set of rules are only shown in the appendix (A.2). However, with static numbering being novel, I will explain a larger fraction of the rules here:

- Uncountables do not have a number:

$$\text{N-True}\ \frac{}{\mathcal{N} \vdash \boxed{\texttt{true}} \,\#\, \eta_\epsilon} \qquad \text{N-False}\ \frac{}{\mathcal{N} \vdash \boxed{\texttt{false}} \,\#\, \eta_\epsilon} \qquad \text{N-Int}\ \frac{}{\mathcal{N} \vdash \boxed{i} \,\#\, \eta_\epsilon}.$$

- Numbers of `new` and `no` expressions are axiomatic:

$$\text{N-NewObj}\ \frac{}{\mathcal{N} \vdash \boxed{\texttt{new}\ C} \,\#\, !} \qquad \text{N-NoObj}\ \frac{}{\mathcal{N} \vdash \boxed{\texttt{no}\ C} \,\#\, -}.$$

— The number rules for `this`, `these`, and field access,

$$\textsc{N-This}\frac{\mathcal{N}[rec] = !}{\mathcal{N} \vdash \boxed{\texttt{this}} \# !} \qquad \textsc{N-These}\frac{\mathcal{N}[rec] = *}{\mathcal{N} \vdash \boxed{\texttt{these}} \# *} \qquad \textsc{N-Field}\frac{\mathcal{N}[rec] = ! \quad \eta = num(rng(C.f))}{\mathcal{N} \vdash \boxed{\texttt{this}^C.f} \# \eta},$$

include checking that they are used in the appropriate contexts, namely, in singular or plural methods, as dictated by the number of *rec* stored in $\mathcal{N}$ (which has been set up by the number rule for method definitions, N-MethDef; see below).

— Both singular and plural methods (distinguished by η'_0, the first number of the method number signature) can be applied to receiver expressions of any number:

$$\textsc{N-MAppl}\frac{\begin{array}{c}\mathcal{N} \vdash e_0 \# \eta_0 \qquad \eta_0 \neq \eta_\epsilon \qquad (\mathcal{N} \vdash e_i \# \eta_i)_{i=1..p} \\ num(sig(C.m)) = \eta'_0\, \eta'_1\, \ldots\, \eta'_p\, \eta' \qquad \left(\eta_i <\# \eta'_i\right)_{i=1..p} \qquad \eta'_0 = ! \wedge \eta = \eta_0.\eta' \vee \eta'_0 = * \wedge \eta = \eta'\end{array}}{\mathcal{N} \vdash \boxed{e_0{}^C.m(e_1, \ldots, e_p)} \# \eta}.$$

While for a singular method the number of the method application expression depends on that of the receiver expression and the declared return number of the method as defined by Table 1(b), for plural methods, it equals the declared return number of the method. Note that, according to Table 1(b), application of a singular method returning uncountables (i.e., $\eta' = \eta_\epsilon$) requires that the receiver has static number $!$.

— The numbers resulting from addition and subtraction are to be read from Table 1(c) and (d), respectively:

$$\textsc{N-Add}\frac{\mathcal{N} \vdash e_1 \# \eta_1 \quad \mathcal{N} \vdash e_2 \# \eta_2 \quad \eta = \eta_1 + \eta_2}{\mathcal{N} \vdash \boxed{e_1 \,{}^{+\kappa}\, e_2} \# \eta} \qquad \textsc{N-Sub}\frac{\mathcal{N} \vdash e_1 \# \eta_1 \quad \mathcal{N} \vdash e_2 \# \eta_2 \quad \eta = \eta_1 - \eta_2}{\mathcal{N} \vdash \boxed{e_1 \,{}^{-\kappa}\, e_2} \# \eta}.$$

— Only single objects (i.e., numbers of objects whose count is 1) and uncountables can be tested for identity or equality, respectively:

$$\textsc{N-EqId}\frac{\mathcal{N} \vdash e_1 \# \eta \quad \mathcal{N} \vdash e_2 \# \eta \quad \eta = ! \vee \eta = \eta_\epsilon}{\mathcal{N} \vdash \boxed{e_1 == e_2} \# \eta_\epsilon}.$$

— Finally, type cast, number cast, and count are only defined for countables:

$$\textsc{N-TCast}\frac{\mathcal{N} \vdash e \# \eta \quad \eta \neq \eta_\epsilon}{\mathcal{N} \vdash \boxed{(C)\, e} \# \eta} \qquad \textsc{N-NCast}\frac{\mathcal{N} \vdash e \# \eta' \quad \eta <\# \eta'}{\mathcal{N} \vdash \boxed{(\eta)\, e} \# \eta} \qquad \textsc{N-Count}\frac{\mathcal{N} \vdash e \# \eta \quad \eta \neq \eta_\epsilon}{\mathcal{N} \vdash \boxed{|e|} \# \eta_\epsilon}.$$

Note that, just as for type casts, number casts must be downcasts.

Number Checking of Statements. The number checking rules for statements are almost a verbatim copy of the corresponding type checking rules from Appendix A.1, replacing $\mathcal{T}$ and the typing relations with $\mathcal{N}$ and the numbering relations. I therefore do not show them here; they can be found in Appendix A.2.

Number Checking of Definitions. Similarly, the rules for number checking the definitions of a program closely follow those of type checking, and are therefore deferred to Appendix A.2. The only marked difference is that the rule for method definitions,

$$\textsc{N-MethDef}\frac{\begin{array}{c}\Big((rec \mapsto \eta)\ (ret \mapsto num(\rho))\ (x_1 \mapsto num(\rho_1)) \ldots (x_p \mapsto num(\rho_p))\Big) \vdash s_1\ \ldots\ s_n \checkmark_\eta \\ \eta = ! \vee \eta = * \qquad P[C] = \texttt{class}\, C\, \texttt{extends}\, C' \cdots \\ method(C'.m) = \bot \vee num(sig(C'.m)) = num(sig(C.m))\end{array}}{M = \boxed{\rho\, m\, \eta\ (\rho_1\, x_1, \ldots, \rho_p\, x_p)\ \{s_1\ \ldots\ s_n\}} \checkmark_\eta \text{ in } C},$$

Table 1. Tabular Definitions of (a) the Subnumber Relation $<\#$; (b) the Number of Singular Method Application Expressions; (c) the Number of Add Expressions; and (d) the Number of Subtract Expressions

(a) $\eta_1 <\# \eta_2$

η_1 \ η_2	–	!	?	*	η_ϵ
–	✓		✓	✓	
!		✓	✓	✓	
?			✓	✓	
*				✓	
η_ϵ					✓

(b) $\eta_1.\eta_2$

η_1 \ η_2	–	!	?	*	η_ϵ
–	–	–	–	–	$\bot$
!	–	!	?	*	η_ϵ
?	–	?	?	*	$\bot$
*	–	*	*	*	$\bot$
η_ϵ	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

(c) $\eta_1 + \eta_2$

η_1 \ η_2	–	!	?	*	η_ϵ
–	–	!	?	*	$\bot$
!	!	*	*	*	$\bot$
?	?	*	*	*	$\bot$
*	*	*	*	*	$\bot$
η_ϵ	$\bot$	$\bot$	$\bot$	$\bot$	η_ϵ

(d) $\eta_1 - \eta_2$

η_1 \ η_2	–	!	?	*	η_ϵ
–	–	–	–	–	$\bot$
!	!	?	?	?	$\bot$
?	?	?	?	?	$\bot$
*	*	*	*	*	$\bot$
η_ϵ	$\bot$	$\bot$	$\bot$	$\bot$	η_ϵ

All tables have been extended to cover uncountables, i.e., the case that $\eta = \eta_\epsilon$ (no number), thereby avoiding numerous case analyses in the specification of Num's semantics. Note that η_ϵ in a table cell means "no number"; it is to be distinguished from $\bot$, meaning "undefined".

adds a condition that the method number must be either ! (for singular methods) or * (for plural methods). Note how this number also defines the number of *rec*, which is used for the number checking of `this` and `these` (rules N-This, N-These, and N-Field above).

Except for using the type annotations (superscript C) elaborated by typing (Section 5.2.1) for the purpose of member lookup, numbering of expressions does not depend on type. This comes at the expense of some redundancy, though: for instance, $\eta \neq \eta_\epsilon$ in N-TCast and N-Count is already ensured by requiring a class type in T-TCast and T-Count. Future presentations of type *and* number systems may therefore consider conflating the two, by specifying rules for ranges ρ rather than types τ and numbers η; here, it would have obfuscated the fact that number and type are largely orthogonal.

5.3 Dynamic Semantics

Because of the difference in their underlying relations, I divide the dynamic semantics of Num into the evaluation of expressions and the execution of statements. For both, I use big-step operational semantics, employing a big-step relation $\Rightarrow$ overloaded accordingly. Big steps keep the specification of Num small, while at the same time providing for its straightforward mapping to the implementation of a Num interpreter. In fact, the rules presented here are a transcription of a Prolog implementation of such an interpreter, which will be described in Section 5.4.

5.3.1 Values. The values v that expressions e can evaluate to are specified by the grammar

$$v ::= b \mid i \mid l_1 \ldots l_n ,$$

where b are booleans, i are integers, and $l_1 \ldots l_n$ are numbers of locations (or object identifiers) l representing numbers of objects on a heap $\mathcal{H}$ (with ϵ meaning no location).

5.3.2 Context of Evaluation and Execution. Both evaluation of expressions and execution of statements are performed in the context of a program P, the heap $\mathcal{H}$, and a locals store $\mathcal{L}$. Of these, the (constant) program P mostly remains implicit. $\mathcal{H}$ and $\mathcal{L}$ are defined as follows:

- A heap $\mathcal{H}$ is a map from locations l to objects o, where an object o is a map from fields f to values v, including a special field f_χ mapping o to the class C of which o is an instance. For example, $o = (f_\chi \mapsto C)(f \mapsto v)$ is an object that is an instance of class C with one field f having value v.
- A locals (or variable) store $\mathcal{L}$ is a map $\mathcal{M} = (x_1 \mapsto \rho_1\, v_1) \ldots (x_n \mapsto \rho_n\, v_n)$ from variable names to pairs of the variables' ranges and values. I redefine map update for locals stores so

that only the value (and not the range) is updated: if $\mathcal{L}[x] = \rho\, v$, then $(\mathcal{L}[x \mapsto v'])[x] = \rho\, v'$. Note that by including ranges ρ, $\mathcal{L}$ combines a store with type and number environments, as will be needed by the type and number safety proofs given in the appendix; store updating does not change these environments.

Following the example of Nipkow and von Oheimb [42], I define the dynamic semantics of expressions and statements via mutually inductive rules. I begin with the evaluation of expressions.

5.3.3 Regular Evaluation of Expressions. Since evaluating expressions may update the heap (by creating new objects), but not the locals store (variable assignment is not an expression and method application creates a new locals store that is discarded upon method termination), regular evaluation judgments have the form $\langle \mathcal{H}, \mathcal{L}, e\rangle \Rightarrow \langle \mathcal{H}', v\rangle$. The regular evaluation rules are mostly standard and are provided in full in Appendix A.3.1; here, I only walk through method application. Note that the evaluation order of the premises is left to right, and that all side conditions appear as premises (i.e., rules are L-attributed in the sense of Ibraheem and Schmidt [28]).

Application of plural methods (identified by $*$ in the first place of their signatures, as required by the first side condition of below rule) resembles standard (object-oriented) method application, the only differences being that the lookup of the method body uses the static type of the receiver, C, and that the receiver, presented by the pseudo-variable *rec*, is a plural:

$$\text{E-PlurMAppl}\ \frac{\begin{array}{c} sig(C.m) = *\,\rho_1\ \ldots\ \rho_p \mapsto \rho \qquad params(C.m) = x_1\ \ldots\ x_p \qquad body(C.m) = s_1\ \ldots\ s_b \\ \langle \mathcal{H}, \mathcal{L}, e_0\rangle \Rightarrow \langle \mathcal{H}_0, v_0\rangle \qquad (\langle \mathcal{H}_{i-1}, \mathcal{L}, e_i\rangle \Rightarrow \langle \mathcal{H}_i, v_i\rangle)_{i=1..p} \\ \mathcal{L}' = (rec \mapsto C* \, v_0)\ (ret \mapsto \rho\ \_)\ (x_1 \mapsto \rho_1\, v_1) \ldots (x_p \mapsto \rho_p\, v_p) \\ \langle \mathcal{H}_p, \mathcal{L}', s_1\ \ldots\ s_b\rangle \Rightarrow \langle \mathcal{H}', \mathcal{L}''\rangle \qquad \mathcal{L}''[ret] = \_\, v \end{array}}{\langle \mathcal{H}, \mathcal{L}, \boxed{e_0{}^{C}.m(e_1, \ldots, e_p)}\rangle \Rightarrow \langle \mathcal{H}', v\rangle}.$$

Note how the rule creates a new locals store, which may be updated by the statements of the method body (via the step $\langle \mathcal{H}_p, \mathcal{L}', s_1\ \ldots\ s_b\rangle \Rightarrow \langle \mathcal{H}', \mathcal{L}''\rangle$; see Section 5.3.5) and from which the returned value is fetched (by reading the pseudo-variable *ret* from $\mathcal{L}''$).

Application of singular methods (identified by ! in the first place of their signatures) returning uncountables (identified by $num(\rho) = \eta_\epsilon$) is handled by the rule

$$\text{E-SingMApplU}\ \frac{\begin{array}{c} sig(C.m) = !\,\rho_1\ \ldots\ \rho_p \mapsto \rho \qquad num(\rho) = \eta_\epsilon \qquad params(C.m) = x_1\ \ldots\ x_p \\ \langle \mathcal{H}, \mathcal{L}, e_0\rangle \Rightarrow \langle \mathcal{H}_0, v_0\rangle \qquad v_0 = l \qquad (\langle \mathcal{H}_{i-1}, \mathcal{L}, e_i\rangle \Rightarrow \langle \mathcal{H}_i, v_i\rangle)_{i=1..p} \\ C' = \mathcal{H}_0[l][f_\chi] \qquad body(C'.m) = s_1\ \ldots\ s_b \\ \mathcal{L}' = (rec \mapsto C'!\ l)\ (ret \mapsto \rho\ \_)\ (x_1 \mapsto \rho_1\, v_1) \ldots (x_p \mapsto \rho_p\, v_p) \\ \langle \mathcal{H}_p, \mathcal{L}', s_1\ \ldots\ s_b\rangle \Rightarrow \langle \mathcal{H}', \mathcal{L}''\rangle \qquad \mathcal{L}''[ret] = \_\, v \end{array}}{\langle \mathcal{H}, \mathcal{L}, \boxed{e_0{}^{C}.m(e_1, \ldots, e_p)}\rangle \Rightarrow \langle \mathcal{H}', v\rangle},$$

which demands that the receiver expression e_0 evaluates to a single location l (representing a single object), as required by the uncountability of the expression (which evaluates to an integer or a boolean).

Finally, the application of singular methods returning countables (identified by $num(\rho) \neq \eta_\epsilon$) is handled by the rule

$$\text{E-SingMApplC}\ \frac{\begin{array}{c} sig(C.m) = !\ \rho_1\ \ldots\ \rho_p \mapsto \rho \qquad num(\rho) \neq \eta_\epsilon \qquad params(C.m) = x_1\ \ldots\ x_p \\ \langle \mathcal{H}, \mathcal{L}, e_0 \rangle \Rightarrow \langle \mathcal{H}_0, v_0 \rangle \qquad\qquad v_0 = l_1\ \ldots\ l_r \\ \left(\begin{array}{c} C_i = \mathcal{H}_0[l_i][f_\chi] \quad body(C_i.m) = s_1\ \ldots\ s_{b_i} \\ \left(\langle \mathcal{H}_{(i-1)(p+1)+j-1}, \mathcal{L}, e_j \rangle \Rightarrow \langle \mathcal{H}_{(i-1)(p+1)+j}, v_{i,j} \rangle\right)_{j=1..p} \\ \mathcal{L}_i = (rec \mapsto C_i\,!\ l_i)\ (ret \mapsto \rho\ \_)\ (x_1 \mapsto \rho_1\ v_{i,1}) \ldots (x_p \mapsto \rho_p\ v_{i,p}) \\ \langle \mathcal{H}_{i(p+1)-1}, \mathcal{L}_i, s_1\ \ldots\ s_{b_i} \rangle \Rightarrow \langle \mathcal{H}_{i(p+1)}, \mathcal{L}'_i \rangle \quad \mathcal{L}'_i[ret] = \_\ v_i \end{array} \right)_{i=1..r} \\ \mathcal{H}' = \mathcal{H}_{r(p+1)} \qquad\qquad v = v_1 \oplus \ldots \oplus v_r \end{array}}{\langle \mathcal{H}, \mathcal{L}, \boxed{e_0{}^C.m(e_1, \ldots, e_p)} \rangle \Rightarrow \langle \mathcal{H}', v \rangle},$$

which does not place constraints on the count of the value of the receiver expression (which must nevertheless be countable, i.e., evaluate to locations). The behavior expressed by this rule conforms to the behavior of the for-loop (or the corresponding `forEach` construct) as used in the example of Figure 1, right, line 15, meaning that the arguments of a method invocation are evaluated once per object among the receivers. While this definition may seem arbitrary, it has the advantage of giving the programmer a choice between "evaluate arguments once per receiver object" and "evaluate arguments exactly once", the latter either by going through a plural method (with formals evaluated once according to E-PlurMAppl) that forwards to the singular method through `these`, or by caching the actual parameters that shall only be evaluated once in temporary variables assigned before the method invocation (and used in that invocation; see Section 6.2 for a discussion of more principled solutions giving programmers control over evaluation semantics).

5.3.4 Exceptional Evaluation. If evaluation runs into an exceptional condition (in Num, a bad type or number downcast exclusively), it cannot proceed regularly. Therefore, I introduce exceptional evaluation judgments, which have the form $\langle \mathcal{H}, \mathcal{L}, e \rangle \Rightarrow \uparrow Exception$, where I call *Exception* a *conceded exception*. For instance, the regular evaluation rule for number casts,

$$\text{E-NCast}\ \frac{\langle \mathcal{H}, \mathcal{L}, e \rangle \Rightarrow \langle \mathcal{H}', v \rangle \qquad v = l_1\ \ldots\ l_n \qquad \eta = *\ \vee\ (\eta = ?\wedge n \leq 1)\ \vee\ (\eta = !\wedge n = 1)\ \vee\ (\eta = -\wedge n = 0)}{\langle \mathcal{H}, \mathcal{L}, \boxed{(\eta)\ e} \rangle \Rightarrow \langle \mathcal{H}', l_1\ \ldots\ l_n \rangle},$$

is complemented by the exceptional evaluation rule

$$\text{E-NCastE}\ \frac{\langle \mathcal{H}, \mathcal{L}, e \rangle \Rightarrow \langle \mathcal{H}', v \rangle \qquad v = l_1\ \ldots\ l_n \qquad \neg\,(\eta = *\ \vee\ (\eta = ?\wedge n \leq 1)\ \vee\ (\eta = !\wedge n = 1)\ \vee\ (\eta = -\wedge n = 0))}{\langle \mathcal{H}, \mathcal{L}, \boxed{(\eta)\ e} \rangle \Rightarrow \uparrow NumberCastException}.$$

Exceptional evaluation is propagated by generic propagation rules (see, e.g., [14, 18, 33]). For instance, the rule

$$\text{E-EqId}\ \frac{\langle \mathcal{H}, \mathcal{L}, e_1 \rangle \Rightarrow \langle \mathcal{H}'', v_1 \rangle \qquad \langle \mathcal{H}'', \mathcal{L}, e_2 \rangle \Rightarrow \langle \mathcal{H}', v_2 \rangle \qquad v = (v_1 = v_2)}{\langle \mathcal{H}, \mathcal{L}, \boxed{e_1 == e_2} \rangle \Rightarrow \langle \mathcal{H}', v \rangle}$$

for regular evaluation is complemented by the rules

$$\frac{\langle \mathcal{H}, \mathcal{L}, e_1 \rangle \Rightarrow \uparrow X}{\langle \mathcal{H}, \mathcal{L}, \boxed{e_1 == e_2} \rangle \Rightarrow \uparrow X} \qquad \frac{\langle \mathcal{H}, \mathcal{L}, e_1 \rangle \Rightarrow \langle \mathcal{H}'', v_1 \rangle \qquad \langle \mathcal{H}'', \mathcal{L}, e_2 \rangle \Rightarrow \uparrow X}{\langle \mathcal{H}, \mathcal{L}, \boxed{e_1 == e_2} \rangle \Rightarrow \uparrow X}$$

for propagating exceptions (where X is a new metavariable ranging over exceptions). A fully generic scheme of exception propagation is provided in Appendix A.3.4.

5.3.5 Regular Execution of Statements. Execution of statements does not return values, but may update the heap (by assigning to a field of an object) and the locals store (by assigning to a variable and by returning from a method). Regular execution judgments are therefore of the form $\langle \mathcal{H}, \mathcal{L}, s_1 \ \ldots \ s_n \rangle \Rightarrow \langle \mathcal{H}', \mathcal{L}' \rangle$. Because the statement execution rules are standard, I have deferred all but one (shown below) to Appendix A.3.3.

5.3.6 Exceptional Execution. While the execution of statements is never itself the source of conceded exceptions in Num, statements need to propagate exceptions coming from the evaluation of contained expressions or statements. The corresponding exceptional execution judgments have the form $\langle \mathcal{H}, \mathcal{L}, s_1 \ \ldots \ s_n \rangle \Rightarrow \uparrow X$. In the case of the return statement, whose regular execution is covered by the rule

$$\text{E-Return}\ \frac{\langle \mathcal{H}, \mathcal{L}, e \rangle \Rightarrow \langle \mathcal{H}', v \rangle \qquad \mathcal{L}' = \mathcal{L}[ret \mapsto v]}{\langle \mathcal{H}, \mathcal{L}, \boxed{\texttt{return}\ e\texttt{;}} \rangle \Rightarrow \langle \mathcal{H}', \mathcal{L}' \rangle},$$

the exception propagation rule is

$$\frac{\langle \mathcal{H}, \mathcal{L}, e \rangle \Rightarrow \uparrow X}{\langle \mathcal{H}, \mathcal{L}, \boxed{\texttt{return}\ e\texttt{;}} \rangle \Rightarrow \uparrow X}.$$

The general scheme for exception propagation through statements is the same as that for expressions, as shown in Appendix A.3.4.

5.3.7 Running Programs. A program P is run by taking a start expression e and evaluating $\langle \epsilon, \epsilon, e \rangle$ against P. A typical start expression would be `no` C`.main(no Object)`, assuming that the class C has a plural method named "main" that takes any number of arguments. The possible outcome of evaluation is either a pair of a heap $\mathcal{H}$ and a value v, a conceded exception, or undefined, where undefinedness is either due to divergence (non-termination of evaluation), or to a definition hole in the specification of $\Rightarrow$, that is, of evaluation or execution ("stuckness").

5.4 Implementation of a Num Interpreter in Prolog

The specification of Num in the preceding sections naturally lends itself to an implementation in Prolog. In fact, the Prolog implementation of Num and its formal specification evolved hand in hand, and the proof of Num's safety is in close correspondence to a proof of the Prolog-based interpreter not failing on well-formed programs (see Section 5.5).

5.4.1 Parsing and Static Checking. The parsing of Num programs using Prolog's definite clause grammars (DCGs) requires a few modifications to the grammar of Figure 2, notably the spelling out of repeats (numbers of nonterminals) and the removal of the left recursion in the definition of expressions. As usual, the predicates of the DCG representing nonterminals are extended by an argument used to construct the abstract syntax tree, whose nodes are represented by Prolog terms and in which numbers of nodes are represented by lists. Using mapping over lists, translating the sanity, type, and number checks for Num programs specified in Sections 5.1 and 5.2 to Prolog rules is straightforward.

5.4.2 Interpretation. The big-step relation $\Rightarrow$ of Section 5.3 is naturally encoded by Prolog terms of the forms

`(P, H, L, e) => (Hp, V)` and `(P, H, L, s) => (Hp, Lp)`

where the program `P` is explicit and e and s stand for terms representing the abstract syntactic patterns of expressions and statements, respectively. The relation is then defined by

implementing the evaluation and execution rules of Section 5.3 as Prolog clauses with terms of the above form serving as clause heads and as subgoals[9] in the clause bodies, where they are complemented by Prolog implementations of the side conditions. The sum of these clauses constitutes a Prolog program that works as an interpreter for Num.

The mapping of regular evaluation and execution rules to Prolog clauses is straightforward. For instance, the rule E-Count maps to the clause

```
(P, H, L, count(E)) => (Hp, N) :-
  (P, H, L, E) => (Hp, V),
  V = Locs, is_list(Locs), length(Locs, N).
```

where the syntactical pattern $|e|$ from E-Count is represented by the term `count(E)` (note how numbers of objects are represented as lists of locations). Exceptional evaluation of expressions, such as that of rule E-NCastE, is implemented by Prolog clauses of the form

```
(P, H, L, ncast(N, E)) => _ :-
  (P, H, L, E) => (Hp, V),
  V = Locs, is_list(Locs), length(Locs, Cnt),
  not (N='*' ; (N='?',Cnt=<1) ; (N='!',Cnt=1) ; (N='-',Cnt=0)),
  throw('NumberCastException').
```

where resolving the subgoal `throw/1` causes immediate abortion of the goal, propagating the thrown exception to the next enclosing exception handler or, if there is none (as is the case of the Num interpreter), to the the evaluation of the start expression. Note how this exhibits the same behavior as Num's exception propagation rules, without needing to implement them.

5.4.3 Running Programs with the Prolog-Based Interpreter. In accord with Section 5.3.7, Num programs are run in Prolog by resolving the goal `(`P`, [], [],` e`) => (Hp, V)`, in which P and e are ground terms representing a program and the start expression, respectively, and `Hp` and `V` are free variables. Compared to an evaluation on paper (using the inference rules of Section 5.3), the use of Prolog as proof engine affords us two conveniences:

(1) built-in propagation of exceptions as described above and
(2) materialization of stuckness as a regular result, *fail*, and its automatic propagation to the top level.

The latter draws on Prolog's closed-world assumption, i.e., on the fact that if Prolog cannot resolve a goal (viz. take an evaluation or execution step, or evaluate a side condition), it responds with failure rather than stuckness. This means that when given program P and expression e, the Prolog-based Num interpreter built from just the regular evaluation and execution rules complemented with the rules for throwing conceded exceptions will do one of four things: (1) succeed with `Hp` and `V` instantiated to a heap and a value, respectively, (2) abort with a conceded exception, (3) diverge, or (4) fail. The last should be ruled out if P and e are well-typed and well-numbered.

5.5 Safety of Num

The primary purpose of the number system of Num is to guarantee that (dynamic) counts are faithful to (static) numbers, that is, that all expressions having a countable (i.e., reference) type always evaluate to counts of objects that are covered by the numbers derived by Num's number system for these expressions, and that all expressions having an uncountable (i.e., value) type

[9] *Subgoal* is a Prolog term referring to a predicate that is to be resolved, or proved, in the course of proving a `goal` (predicate) matching the head of the rule in whose body the subgoal occurs.

evaluate to one value. For variables and fields this means that the count of the objects held by them is subsumed by their declared number. A different, yet equally important guarantee is that the number rules provided in Section 5.2 sufficiently guard the evaluation and execution rules of Section 5.3, meaning that they admit only programs that can be interpreted by these rules (so that interpretation does not get stuck). Of course, these guarantees are the exact analogs of the guarantees given by type safety, so that what we want is *number safety*. Given that Num's number system has been crafted in analogy to its type system, and given that this type system is standard, it is immediately clear how number safety can be proved. The only hurdle to be taken is that for a proof based on big-step evaluation and execution rules, stuckness must be distinguished from divergence.

5.5.1 Stuckness as Failure. As noted above, implementing the evaluation and execution rules as Prolog clauses implicitly provides for an explicit capture of being stuck: provided that rule selection is deterministic, under Prolog's closed-world assumption, stuckness is equivalent to failure, except that the latter is a regular result that, similar to a thrown exception, is propagated to the start expression.[10] By adding explicit *failure rules* [17, 18, 47] that share the shape of exception rules and hence can draw on exception propagation as introduced in Section 5.3.4, failure (as the result of running the Prolog-based interpreter) is reproduced in the inference rules that make the specification of dynamic semantics. For instance, complementing the rule

$$\text{E-Var}\ \frac{\mathcal{L}[x] = \rho\, v}{\langle \mathcal{H}, \mathcal{L}, \boxed{x} \rangle \Rightarrow \langle \mathcal{H}, v \rangle} \qquad \text{with} \qquad \frac{\neg(\mathcal{L}[x] = \rho\, v)}{\langle \mathcal{H}, \mathcal{L}, \boxed{x} \rangle \Rightarrow \uparrow \mathit{fail}}$$

catches the failure of the side condition of E-Var that causes stuckness, as exceptional evaluation yielding the *unconceded* exception *fail*.[11] The general scheme of failure rules for the evaluation of expressions and the execution of statements is given in Appendix A.3.5.

5.5.2 Type and Number Safety. With stuckness captured as failure, we are equipped to state type and number safety of Num as follows:

> For every well-typed, well-numbered, and otherwise well-formed program P and every start expression e that is well-typed and well-numbered with respect to P, if evaluation of $\langle \epsilon, \epsilon, e \rangle$ terminates, then we have $\neg(\langle \epsilon, \epsilon, e \rangle \Rightarrow \uparrow \mathit{fail})$, and if $\langle \epsilon, \epsilon, e \rangle \Rightarrow \langle \mathcal{H}, v \rangle$, then $\mathcal{H}$ and v are well-typed and well-numbered.

Note that, as usual, this phrasing of safety includes divergence and exceptional evaluation producing a conceded exception as safe cases. A complete capture of the safety theorem (including formal captures of well-typedness and well-numberedness of $\mathcal{H}$ and v), together with its proof, will be given in Appendix B.

6 DISCUSSION

6.1 Countables and Uncountables

The separation into countables and uncountables may seem dogmatic, yet means that when dealing with primitive values (uncountables), one can largely ignore the number dimension, saving one from oddities such as branching on no boolean. If one rejects this high-level separation, one could have a slightly less dogmatic variant of Num, in which the types `bool` and `int` (or any value

[10] The mechanisms are not the same: unlike exceptions, failure is "propagated" via backtracking, so that equivalence depends on the absence of choice points. This is granted by the premise that rule selection is deterministic.

[11] For this, the evaluation of side conditions also assumes a closed world, meaning that undefinedness, here caused by $x \notin \mathit{dom}(\mathcal{L})$, is interpreted as falseness.

type for that matter) must be paired with number `!`. This would require only minor adaptations of numbering: syntactically, η would no longer be optional (so that η_ϵ can be dropped), and the numbering rules would need to make sure that value-typed members are accessed on `!` receivers exclusively, thereby mixing numbering with typing. As an alternative to this static restriction, dynamic semantics could insert a multiplicity cast of the receiver to `!`, causing a number cast exception where member access would otherwise yield not one, but no or many primitive values. If, on the other hand, one preferred to have arbitrary numbers of values (booleans and integers), one would need to be prepared for a major reworking of larger parts of the language (cf. the introduction of the struct `Nullable<T>` in C♯ discussed in Section 7.2).

That said, the separation of value types and reference types does not preclude us from using wrapper types (such as `Integer` in Java) to turn uncountables into countables. However, I do not suggest the use of auto-boxing for this purpose; instead, in accord with the ontological considerations of Section 3.1.3, I suggest that wrapping is an explicit choice of the programmer: to create an object with identity for the sake of making it identifiable and, hence, countable. My recent essay [58] provides further ontological arguments for this.

6.2 Plurals vs. Containers

Numbers of objects, or containerless plurals, as featured by Num unify the handling of no, one, and many objects, without the drawbacks that would come with the use of containers. However, in situations in which numbers of objects have additional meaning attached, keeping them in containers is likely the better choice. This is typically the case when a collective noun is used to denote the plural: "sequence" implies not only that the objects that make the sequence are arranged in a specific order, but also that the particular order makes the sequence (not the objects!) different from others. Likewise, "priority queue" implies that the objects are sorted, and so on. In fact, each of these collective nouns gives rise to a special type of collection, which is also a data type, i.e., it does not only describe the structure in which the objects are contained, but also the operations that are defined on the structure (as opposed to those defined on the objects). Programs that build on these operations should use collections instead of numbers of objects.

And yet, there are also situations in which programmers would like numbers of objects to exhibit some properties that can be had with collections without buying into the data types and the reification that comes with them. For these cases, making the container type a parameter of the object type (which amounts to the inversion of the relationship of container and content [56, 59]) lets the programmer control the nature of the plural, that is, whether it is ordered, sorted, or has duplicates; and also its specific performance profile (e.g., linked list, array, or parallel access). To operate on a (hidden) container using its own protocol, it can be accessed through special syntax [59]; in all other contexts, the type of the number of objects is the type of its objects, as in Num.

Another disadvantage of the plurals of Num is that their implementation commits to one evaluation strategy for method invocations on numbers of objects, whereas when using containers and loops, other strategies can be programmed. While the built-in evaluation strategy will avoid significant complexity in some programs (by pushing it to the compiler), it may mean new complexity in others, namely, if they need a different evaluation strategy. For instance, singular method invocation on many objects (E-SingMApplC) prescribes that the actual parameters are (re-)evaluated for each receiver, which may not always be desired; to have a single evaluation instead, one would either need to cache the actuals in variables before the invocation or go through a plural method forwarding to the singular method. More advanced languages building on the ideas of Num should

consider the use of *adverbs*, as advocated by Ungar and Adams [63] for the language Ly (cf. Section 7.2.6), to give programmers control over evaluation strategies.

6.3 Elimination of Type Cast Exceptions

While the introduction of number has eliminated null pointer exceptions, it could have also eliminated type cast exceptions, by redefining (*C*) *e* so that it returns all and only those objects among *e* that can be cast to *C*. For expressions *e* having number `?` or `!`, (*C*) *e* in Num would then provide for a null-free implementation of the type cast *e* `as` *C* in C♯ and other languages (which evaluates to `null` if the cast does not succeed). While this would leave Num with a single conceded exception, the number cast exception, it would mix typing with numbering, so that I did not use this opportunity of advertising the power of numbering here.

6.4 Practicality, Utility, and Performance Penalty

In previous work [59], we refactored the core of the design-pattern rich JUnit 4 framework to using multitudes of objects (the forerunners of numbers of objects; cf. Section 7.1). The work showed that the refactoring is pervasive: apart from the obvious opportunities for code simplification (loop removal, removal of tests for not `null`, unification of `add`, and `addAll` methods), we also discovered that small additional refactorings can lead to further opportunities of making the code less keyword-laden, or more "fluent" [22]. For instance, while rewriting

```
for (Runner each : fRunners) each.run(notifier);
```

in Java to

```
fRunners.run(notifier);
```

in Num exploits that the type of `fRunners` is `Runner` (and that method invocation on many receivers means method invocation on each), replacing

```
for (Runner runner : fRunners) spec.addChild(runner.getDescription());
```

with

```
spec.addChild(fRunners.getDescription());
```

additionally exploits that `addChild(·)` can be generalized to accepting any number of descriptions by changing its formal parameter's declaration from `Description!` to `Description*` (note that adding many descriptions, instead of one, to many descriptions does not require a further change of numbers). Furthermore, by moving the method `asTest(Description)` from a class that implements a test adapter to class `Description` (where, given that the method converts a description to a test, one might think it belongs), the code

```
for (Description child : description.getChildren()) suite.addTest(asTest(child));
```

from the adapter can be refactored to

```
suite.addTest(description.getChildren().asTest(this));
```

a change that is reminiscent of rewriting a sentence from passive to active voice which, in Num, is generally possible because in Num, not only the object (parameter), but also the subject (receiver) of a predicate (method) can be a plural. In one particularly impressive example that additionally involves free conversion from (containerless) numbers of objects to lists (containers; see Section 6.2), the size of the code could be more than halved (see Steimann et al. [59] for the details). Overall, we found that with growing experience, more and more opportunities of using plurals for the better of the code became apparent.

Since we expected a performance penalty from the copying of many pointers that assignment means (where using containers only one pointer would need to be copied), we devised an alternative implementation of multitudes of objects that used lazy copying (similar to copy-on-write, but beware that Java's copy-on-write collections serve a very different purpose). However, detailed performance measurements on the benchmarks used in the study [59] showed (1) that the performance degradation caused by eager copying was negligible when compared to the original, container-full design (less than 1% in three subjects and approximately 5% in one subject) and (2) that the introduction of lazy copying itself caused a notable performance overhead. Closer inspection showed that this was due to the fact that half of the collections copied in the subject programs were empty and less than 1% had sizes larger than 10, so that the cost of (eager) copying was lower than the cost of creating a lazy collection and going through the additional indirection that its use means.

6.5 Backward Compatibility

Not surprisingly, the case study [59] also showed that like the use of type annotations, the use of number annotations propagates through programs (including its used libraries and frameworks) and, therefore, that adopting numbers of objects in existing programs means a significant refactoring effort. To allow for a smooth transition to programming with numbers of objects, our earlier works [56, 57, 59] introduced a default number specifier, called *bare*, which expressed the same number constraint as `@option` (no or one object), but meant that `null` was used for encoding no object. A program with only bare (default) number specifiers therefore compiled and behaved exactly like a standard Java program, including the possibility of running into null pointer exceptions; number safety, meaning the exclusion of these exceptions, required the introduction of `@option` annotations (meaning that accessing no object had no effect, as in this present work).

To avoid the challenges of dealing with optional primitive values obtained by accessing members having primitive types on receivers with static number `@option` (see Section 6.1), such members could only be accessed on bare receivers, which, as noted above, would throw a null pointer exception rather than return an optional value. For accessing these members on `@option` receivers nevertheless, the receivers had to be downcast to bare first. Vice versa, for accessing them as numbers of objects, bare expressions had to be upcast to `@option`. Availability of a number specifier *bare* will probably be as important for the integration of un-numbered with numbered programs as the introduction of the `dynamic` type was for the integration of dynamically typed languages with C♯ [6].

6.6 Application to Other Programming Paradigms

As noted in the introduction, the long-term goal of this work is making object-oriented programming more relational. Therefore, it has been embedded in an object-oriented, imperative setting. However, the discontinuities that come with the use of containers, specifically those induced by the indirection and the change in type that this means, also exist in functional languages. On the other hand, functional languages usually offer monadic types, including `Optional` (or `Maybe`) and `List`, which come with special syntax that makes their use more convenient than other container types and, equally importantly, which are programmer-defined and hence more flexible than numbers of objects built into the language, as in Num. And yet, monads still mean indirection and the necessity of wrapping and unwrapping at the program level. Logic programming languages such as Prolog implement plurals (or relations) through backtracking (cf. Section 7.3), yet also make heavy use of lists; where lists are not domain-level concepts (cf. Section 6.2), one could consider replacing them with numbers of objects (for instance, for representing the results of `findall` predicates).

Ultimately, however, extending Prolog with plurals would mean the introduction of "numbers of terms" which are not themselves terms, which would require a reworking of the language from the bottom up, including its logical foundations.

7 RELATED WORK

7.1 Containerless Plurals

7.1.1 Bunch Theory. Hehner [26, 27] developed a general *theory of bunches* of which the numbers of objects defined in Section 3.2 are a special case. Specifically, Hehner's bunches are collections of elements that are conceived after sets, but are simpler in that they are always flat—while sets may comprise sets, bunches cannot be elements of bunches.[12] Also, as for the numbers of objects on which Num builds, a bunch consisting of a single element, called an *elementary bunch*, and the element are indistinguishable. While the original definition of bunches [27] provided definitions of "element of" ($\in$) and "sub-bunch of" ($\subseteq$), in a reprise, Hehner [26] conflated $\in$ and $\subseteq$ for bunches into one inclusion relation (:) that corresponds to the definition of "among" ($\propto$) in Num. Bunches do not distinguish between countables and uncountables; instead, the theory of bunches lifts arithmetic and other operators to bunches of values [26, 27] (see below for discussions of this practice).

Originally, Hehner [27, p. 26] suggested that bunches can be elements of sets (meaning that they "can be thought of as a one"). However, this contradicts the view of bunches as "the contents of a set" [26, pp. 14&17], because for a set containing bunches, the bunch given by the contents of this set would consist of bunches, and hence not be flat. By contrast, Num allows the implementation of collections of numbers of objects; for instance, in the example of Section 4.4, the number of the field `CollectionElement.element` can be changed from `!` to `*` without problems.[13] One should be aware; however, that this overloads the meaning of the method `Collection.add!(Collectable *os)`: as currently defined, it would still add each object among `os` as a separate collection element, whereas one might want it to mean alternatively that all objects are added in a single new place of the collection, which is something rather different.

7.1.2 Strings. Strings of words can also be considered containerless plurals, or "unpackaged sequences" [26, p. 17], but are more general than bunches since they allow duplicates. The same holds for the "unenclosed sequences" identified by Steele [55] as a recurring data structure of language specification, which uses various notations ($a*$, $\overline{a}$, $a_1 \ldots a_n$), and indeed, instances of unenclosed sequences appear to be strings. It turns out that Hehner [27] also suggested to use bunches in language specification; that bunches do not allow duplicates is not a problem if the elements of such a bunch are distinct nodes of a syntax tree; in fact, the specification of Num in Section 5 used numbers of objects as "unenclosed sequences".

7.1.3 Object-Oriented Programming with Multiplicities. While this present work emphasizes containerlessness, my original ideas [56] and their prototype implementation in Java [59] rested on the inversion of the relationship between container and content: rather than making the element type a subscript to the container type (the parametric polymorphism of containers), the container type was made a subscript to the element type. For instance, `@any(ArrayList) Observer`

[12] In fact, as pointed out by Hehner [26, p. 17]: "All sets are elements; not all bunches are elements; that is, the difference between sets and bunches." Substituting "element" with "object", the same holds for numbers of objects: Only a singular number of objects (i.e., a number of objects with count 1) is an object.

[13] Note that this does not make numbers of objects objects: The place of a collection is a variable like any other (except perhaps that it appears to be unnamed), and may thus hold any numbers of objects. On the other hand, given that numbers of objects with static numbers other than `!` cannot be tested for identity (N-EqId), implementing collections of numbers of objects with identity-based set semantics is impossible.

`observers` declared a variable (or field) `observers` that could hold any number of observers, where the observers were stored—under the hood of the language—in an instance of class `ArrayList`. Upon assignment, the contents of the original container were copied into the target container, thereby marrying the container's covariance with mutability. The compiler kept the containers an implementation secret of the language, hiding the indirection through them just like it hides the indirection through references to objects; however, to interface numbers of objects (which I called *multitudes of objects* back then) with Java collections and streams, the language extension provided operations for explicitly packing numbers of objects in, and unpacking them from, collections. One practical advantage of this earlier work was that the programmer was given control over the nature of multitudes, specifically whether they should be ordered or whether duplicates should be allowed (cf. the discussion in Section 6.2); this present work is more purist in that it makes clear that numbers of objects do not have a nature—they are not themselves objects. Also, while the pragmatic approach adopted by the earlier work may seem attractive for practical implementations of programming with numbers of objects, the strictly containerless approach presented in this present article is better suited for a formal study of the core ideas. Finally, Num features a *one* annotation (`!`), which is required to make access of value-typed members on numbers of objects statically safe; this was previously missing.

Native Multiplicities. An alternative interpretation of my earlier captures of numbers of objects [56, 57, 59] can be found in the so-called *native multiplicities* of Harkes's *relations language* devised for relational data modeling and querying [24, 25]. Native multiplicities follow this earlier work in that they separate type from multiplicity, but differ in that they extend multiplicity to primitive types. Harkes also provides type and number checking rules together with a safety proof for a sublanguage [25], which does, however, not have updatable stores. Also, native multiplicities lift all binary operations to the Cartesian product of their operands, which Hehner [27] also has for bunches. While forming the Cartesian product is in line with using bunches, or multi-values, for encoding nondeterminism (see Section 7.3), it has to be set off from array programming, which needs to consider pairwise application, and hence size checking (see Section 7.2.5).

7.2 Encoding Number in Type

7.2.1 Monads. Functional languages like Haskell and F♯ come with special syntax for monads freeing the use of optional and list types from much of the notational clutter induced by the necessary wrapping and unwrapping. Yet, using monads for dealing with no or many objects is still different from code dealing with one (unwrapped) value. On the other hand, `Maybe` and `List` monads are frequently seen in formal language specifications and accompanying (mechanized) safety proofs (e.g., [1, 42]); to demonstrate that the use of numbers of objects is not limited to (object-oriented) programming, I have used them in the specification of Num also.

7.2.2 Streams and Sequences. The experimental languages Xen [37] and Cω [4], the forerunners of adding relational-style querying to object-oriented programming in C♯ [5], introduced streams as immutable, ordered, homogeneous collections (containers) of zero or more values (where values include primitive values, structs, and objects). Streams are constructed from base types using the same number specifiers as Num, and for a type T, the subtype axioms $T! <: T$, $T <: T?$, and $T? <: T*$ provide for assignment compatibility ($T!$ was dropped in Cω). The integration of T means that single objects (or values) and `null` can be assigned to variables having a sequence type. Even though streams are themselves objects, they are always flat, meaning that there are no streams of streams. Afforded by immutability, streams over reference types are covariant and can be aliased (meaning that a variable having type $T*$ can alias an object having type $S*$, if $S <: T$). Streams

gain their power from "generalized member access" [4], i.e., the lifting of member access over streams, allowing path expressions as in Xpath, OCL [46], Alloy [32], or indeed Num. However, the integration of stream types into a standard type system required various ad hoc exceptions to the type rules (including flattening; see [57] for a critical account) which may be considered evidence that number should rather not be encoded in type. For instance, according to the type rules of Cω, `Object <: Object* <: Object`.

Like the streams of Xen and Cω, the sequences of JavaFX™ [61] are flat, covariant containers, but unlike those streams, they have value semantics (they are copied upon assignment) and thereby afford mutability. As in Xen and Cω, single objects (or values) and `null` can be assigned to variables having a sequence type, but unlike in Cω (or Num), there is no way to statically constrain the length of sequences; specifically, there is no *option* (`?`) or *one* (`!`) number modifier. In fact, the type of a sequence is incompatible with its element type, so that a sequence cannot occur where an object having its element type is required, and the members (methods and fields) of the element type cannot be accessed on expressions having the sequence type, unlike in Num, where expressions evaluating to numbers of objects have a common supertype of the objects, so that in terms of typing, any number of objects can replace for a single object (number is orthogonal to type).

7.2.3 Optional Values. C♯ has structs that, like primitive types, are value types (as opposed to the reference types of objects). The struct `Nullable<T>` implements a generic nullable value type that extends other value types `T` with a `null` value [39], so as to be able to interface relational databases (in which values may be `null`) [36]. `Nullable<T>` defines an access protocol similar to that of `Optional<T>` for reference types in Java, but special syntax allows its use without the method invocation clutter. Interestingly, an expression having type `Nullable<T>` can be cast to the underlying value type `T`; such a cast causes a runtime exception if the value of the expression is `null`. This is largely analogous to Num's number cast to `!` (see Section 5.3), although it has been reserved for reference types.

In C♯, the operators defined for primitive types `T` are lifted to their corresponding `Nullable<T>` types, yielding results in accord with the same operations in database languages dealing with null values, meaning the introduction of a ternary logic for booleans. It follows that `Nullable<bool>` cannot be used as the condition in if-statements, just as `Nullable<int>` cannot be used for indexed array access. Also, boxing of nullable values does not result in instances of an `Optional` type (which C♯ does not offer), but rather in the boxed payload, where the value `null` is "boxed" as the null pointer. Num, by contrast, treats primitive types as uncountable, and hence as not nullable; in fact, following an ontological argument that connects countability to identifiability [58], it would treat all value types (including structs) as uncountable—to form a plural, values would need to be boxed, that is, given identity.

7.2.4 Not Null and Safe Initialization. While primitive types are usually not nullable by default, non-nullable reference types are only beginning to be seen in mainstream programming languages. In Kotlin,[14] a Java dialect that is popular in the Android programming ecosystem, types are non-null by default; nullable types must be constructed from them, and are accompanied by a variant of member access, called *safe call*, that yields `null` if the receiver is `null` and a wrapper if the accessed member has a primitive (aka uncountable) type, thereby making it nullable (corresponding to how uncountables can be made countable in this work; see Section 6.2). Safe calls occurring on the left-hand side of an assignment may also evaluate to `null`; in that case, the expression on the right-hand side is not evaluated. To support assignment of nullables to non-nullables, Kotlin has a unary not-null assertion operator, `!!` (which corresponds to Num's number cast `(!)`, except that

[14] https://kotlinlang.org/docs/kotlin-docs.pdf.

it throws a null pointer and not a number cast exception), and a binary null coalescing operator (`?:`, also `??` in C♯ and other languages), which returns the first argument if it is not null and the second otherwise. While these features of Kotlin approximate the semantics of Num for optional expressions, they still require special syntax and, more importantly, they are not generalized to many objects.

Where static checking for non-nullness is available, it is usually considered an extended form of type checking, justified by regarding a non-nullable type as a subtype of the corresponding nullable type [60]. By contrast, Num separates number from type (but preserves, through sub-numbering, that number `!`, for *one*, can occur wherever `?`, for *optional*, is expected), and extends (non-)nullability to numbering, thereby not only covering the absence of a value, but also the presence of many.

While static checking for not null, like static type checking, is generally limited by the possibilities of static analysis, one particulary unrelenting problem is that of initializing recursive, including circular, data structures [19, 51, 60]. Num's approach to this problem is simplistic: It has no constructors and initializes all fields with literal values or new objects specified at the fields' declaration site. Here, all `!` fields must be inialized with new objects that may serve as placeholders, but which, in case of recursive data structures, may lead to divergence. However, while this is less than ideal, I follow [60] here and regard non-nullness and safe object initialization as different matters; Num's focus is on the former and its extension to covering not only null and not-null, but also many. Safe circular initialization, which is related to the safe update of bi-directional pointers, is yet to be added.

7.2.5 Array Programming. The main goal of Num is preparing object-oriented programming for its development into object-relational programming. Indeed, that variables can hold, and expressions can evaluate to, flat and uncontained numbers of objects support the uniform navigation of relational and network data models (objects graphs) with arbitrary multiplicities (cf. Section 7.5). In other domains, however, in which many objects also play a central role, containers are essential carriers of meaning, and therefore must not be dismissed. For instance, computing with containers is the essence of array programming, as supported by languages such as APL [31]. In these languages, arrays are single values, each comprising a number of other values (which may be scalars or arrays). Operations defined on scalars are lifted to arrays in programmer-controllable ways (e.g., pairwise or cross product), resulting in very compact programs. Contemporary static typing allows capturing not only the dimension, but also the size of each dimension of an array in a (dependent) type [23, 53], thus making array programming not only type- and dimension-, but also size-safe. Even though size safety roughly corresponds to number safety as promoted by this work, I deliberately depart from array programming, for the simple reason that arrays are semantic, or meaning-carrying, containers that cannot be dropped: A vector for instance is a mathematical object, and as such not just a number of scalars (cf. the discussion in Section 6.2). Num caters for numbers of vectors (or containers), but again, this is to be distinguished from a vector of vectors, or a matrix, which are likewise single (singular) objects. Because containers are essential to array programming, this work, which promotes containerlessness, is not a competitor.

7.2.6 Ensembles and Adverbs. The experimental language Ly aims at harnessing multi-core processors by organizing numbers of objects in ensembles [63]. An ensemble receiving a message delegates it to its members, which process it as specified by a so-called *adverb* (e.g., parallel, serial, or pairwise), unless the message (e.g., `size`) is sent to the ensemble as a whole (using special syntax; this corresponds to the invocation of plural methods in Num, which does however not require special syntax at the call site). In Ly, a singleton ensemble is not the same as its member object and

it would appear that if Ly were statically typed, the two would have different types. Static typing would, however, stand in the way of automatic delegation of methods from ensembles to their members, unless Ly adopted the type and number system of Num, which would give ensembles the type of their members. In fact, it seems that this would solve a problem Ungar and Adams [63] have observed with empty ensembles, namely that they accept messages that would have caused runtime type errors had they been populated; in Num, all expressions evaluating to zero objects have the static type of the objects they could alternatively evaluate to.

7.3 Nondeterminacy and Variation

Providing many objects where there would be expected only one can be used to express nondeterminacy [9, 54] or variation [16, 38] in programs. In fact, one use of bunches suggested by Hehner [26, p. 89] and also by Morris and Bunkenburg [41] is the representation of nondeterminacy: A bunch can present a choice of possible values, just like a probability distribution (random variable) or a fuzzy set can. The numbers of objects presented by this present work could also be interpreted in this way; yet, as noted in the introduction, my goal is to make object-oriented programming more relational (but see the discussion of Prolog below for how this can be viewed as two sides of the same coin).

Generally, deterministic programming languages can be made nondeterministic by introduction of a choice operator [9, 54], which expresses the (nondeterministic) selection from a number of values. Embedding such an operator into a language gives rise to several non-trivial design decisions; however, the most basic perhaps being whether in computations, "a variable is always bound to exactly one value or is bound to a set of possible values" [54, p. 518]. That this choice is essential can be seen from the simple example of computing the value of $x + x$ where the value of x is nondeterministically chosen from the set $\{1, 2\}$: If x is always bound to exactly one value (dubbed "singular semantics" by Søndergaard and Sestoft [54]), $x + x$ evaluates to 2 or 4, whereas if x is bound to the set $\{1, 2\}$ ("plural semantics"), $x + x$ evaluates to $\{2, 3, 4\}$. There are various ways of reducing the combinatorial complexity introduced by plural semantics and the need to form the Cartesian product of choices, including choice elimination [16], narrowing [9], and the introduction of choice dimensions [16] or variability contexts [38]. This present work, although adopting plural semantics, is careful to not introduce Cartesian products (even though Num can compute them if needed; see Section 4.3).

Singular semantics is adopted by (sequential) Prolog, which uses backtracking to go through all possible choices (bindings of variables). However, nondeterminacy is only one interpretation of Prolog programs—the other is that Prolog is relational, i.e., it computes relations rather than functions [64]. This present work is also concerned with implementing relations: specifically, an object's field holding many objects implements a relationship between the field owner and the objects held by the field. This relationship is simultaneous and between all objects; specifically, it is not meant as a choice. If it is interpreted as one nevertheless, one needs to be clear about what the absence of an object ("no object") means, specifically if the language does not have a notion of failure as a regular result (as Prolog does).

7.4 First Class Relationships

Until this day, relationships are typically implemented in object-oriented programming languages by using coding patterns [43], which include the use of fields for directed N:1 relationships. This is despite the fact that very early in the rise of object-oriented programming, Rumbaugh [48] already argued, with good reasons, for the lifting of relationship encodings to the level of a first class language construct. For this purpose, he introduced relations as instances of a special class `Relation` that has fields holding a relation declaration (i.e., the types of the participants, role names, car-

dinalities, etc.), as well as a field holding the extension of the relation (i.e., its tuples). Unlike in many other approaches that followed, an instance of `Relation` represents a relation, not a tuple; standard operations Rumbaugh defined on these instances included the adding and removal of tuples, indexed access to tuples of the relation, and scanning of the relation (iterating over its tuples). Later, Rumbaugh also added so-called propagation attributes to relations, which allowed the controlled recursive propagation of certain method invocations through object graphs [49]. While Rumbaugh's proposals amount to embedding a native implementation of (parts of) a relational database system in object-oriented programs, Num's approach of implementing to-many pointers is lightweight.

Bierman and Wren's RelJ [8] is based on a notion of relationships as first class types whose instances, called relationship instances, are tuples. These tuples, which—like objects—can have state and behavior, are created and returned by adding a pair of objects to a relationship. Navigation of a relationship always results in a set; since sets have value semantics, the result type of navigation is covariant with the target type of the navigation. However, member, including relationship, access is not lifted to sets; specifically, sets cannot be the source of navigation, so that navigation cannot be chained as in Num. Bierman and Wren also suggest how multiplicities could be restricted statically, using *one* and *many* annotations (analogous to Num's ? and *); because their relationship update is additive, the number invariant imposed by *one* is enforced by changing the semantics of adding to a relation with that of replacing an instance of a relation (destructive update, or assignment; no formalization provided). By contrast, Num does not provide for non-destructive updates; however, similar problems will also be faced when extending its uni-directional to-many pointers to bi-directional.

In the language Rumer, references to objects are completely expelled from so-called entity types (conventional classes), and objects are related exclusively through relationship types [3]. It follows that only relationships know which entities are related (referred to as stratification in [3]). Entity and relationship types have associated extent types, which must be instantiated and populated explicitly. Relationships can be nested, and relationship extents can be owned by relationships, so that they cannot escape the owning relationship. While owned relationship extents bear some resemblance to numbers of objects (which likewise cannot be aliased), Rumer's approach seems rather heavy weight—in particular, with all knowledge about relationships fully encapsulated in relationships (so that objects are ignorant of whether an how they are related), much of an application's logic (including that captured in most methods) has to be moved to relationships, with objects being degraded mostly to passive data containers with identity. This means a fundamental paradigm shift for object-oriented programming, where Num advocates a more lightweight approach, in which fields implement one direction of a binary relation.

7.5 Multiplicities in Modeling and Query Languages

In modeling languages such as the Unified Modeling Language (UML) [45], the Object Constraint Language (OCL) [12, 46], or Alloy [32], number constraints are expressed as multiplicities. While multiplicities can be arbitrary sets of natural numbers, the multiplicities $[0, 1]$, $[1, 1]$, and $[0, \infty)$ (corresponding to ?, !, and * in various other languages, including Num) are particularly popular.

OCL [12, 46], which is used to express well-formedness conditions of UML models, allows the navigation (dereferencing) of attributes and associations independently of their multiplicity, using the same dot operator. However, OCL contains numbers of objects in collections; the difference between one and many objects is conjured away by allowing collection operations to be applied to single objects also (meaning that they are coerced to singleton containers). Extra wrapping is not necessary in Alloy [32], which does not distinguish between scalars and sets, but instead represents scalars as singleton sets (so that the difference between $\subseteq$ and $\in$ disappears, leaving

only $\subseteq$). Except for the use of sets as containers, this is close to the numbers of objects of this present work, and indeed, Alloy is relational; yet, it is not a programming language.

While Alloy and UML are relational, other data, XML- and DOM-based in particular, is tree-structured. Navigation in trees is supported by powerful query languages such as jQuery, which is implemented as a JavaScript library offering a fluent query API. As in OCL, Num and some other languages, in jQuery, no, one, and many objects are navigated using the same expressions, which means that, as discussed in Section 4.1 for Num, queries may "fail silently": navigation through partial expressions resulting in "no object" lead to no object, and executing a method on no object does nothing (rather than flag an error). To address this, Lerner et al. [34] have devised a static type system for jQuery programs in which so-called multiplicities annotate container types with abstract size information `0`, `1`, `01`, `1+`, and `0+`, which correspond to `-`, `!`, `?`, `+`, and `*` as partly also used by Num and other languages (see above). To statically protect navigation from silent failures, the structure of documents is (locally) encoded by the piecewise definition of so-called type functions `@children`, `@parent`, `@next`, and `@prev`, which let the type checker compute the number and element type of path expressions such as `$.children().next().next()`, where `0` and `0+` indicate that method invocations on the so-annotated expressions might silently fail (and are therefore ill-typed). The rules of combining the numbers in expressions are in accord with those of other works that encode number in type (e.g., Xen and Cω; cf. above) or keep them separate (the native multiplicities of Harkes and Visser [24] and indeed Num; specifically, in accord with Num, accessing values of nodes, through `css("color")`, for instance, requires receiver multiplicity `1`), which may be seen as indication that these rules come naturally (they are in fact abstractions from interval arithmetic).

8 CONCLUSION

For most programs, occurrences of no or many objects are not exceptional, but standard situations. Rather than encoding these standard situations using special constructs, such as null pointers, collections, or monads, I have advocated a general paradigm shift: moving on from single objects, or singulars, to any number of objects, or plurals. While the generalization of *one* to *many* in programming languages has been proposed before, it has relied on the use of containers, and with it on the type system to encode the differences between singular and plural. By contrast, my work introduces number as a largely orthogonal dimension in language design that avoids the quirks of type-based solutions; its very own quirk, that it separates values into countables (objects) and uncountables (non-objects), may be seen as the generalization of a distinction that is already widely established, namely, that between reference types (which may be viewed as expressing limited countability, with counts 0 and 1) and value types (which require wrapping to express the absence of a value). For reference types, the introduction of numbers of objects means the elimination of null pointer dereferences, but not as the result of a special effort, but as part of a general solution that eliminates the use of containers for representing many objects also, thus making none, one, and many degrees on one scale. Number safety guarantees that this new scale is not a source of failure.

APPENDICES

A SEMANTICS OF NUM

A.1 Type Rules

A.1.1 *Type Rules for Expressions.*

$$\text{T-True}\ \frac{}{\mathcal{T} \vdash \boxed{\texttt{true}} : \mathit{bool}} \qquad \text{T-False}\ \frac{}{\mathcal{T} \vdash \boxed{\texttt{false}} : \mathit{bool}} \qquad \text{T-Int}\ \frac{}{\mathcal{T} \vdash \boxed{i} : \mathit{int}}$$

$$\text{T-NoObj}\ \frac{}{\mathcal{T} \vdash \boxed{\texttt{no}\ C} : C} \qquad \text{T-NewObj}\ \frac{}{\mathcal{T} \vdash \boxed{\texttt{new}\ C} : C} \qquad \text{T-Var}\ \frac{\mathcal{T}[x] = \tau}{\mathcal{T} \vdash \boxed{x} : \tau}$$

$$\text{T-This}\ \frac{\mathcal{T}[rec] = C}{\mathcal{T} \vdash \boxed{\texttt{this}} : C} \qquad \text{T-These}\ \frac{\mathcal{T}[rec] = C}{\mathcal{T} \vdash \boxed{\texttt{these}} : C} \qquad \text{T-Field}\ \frac{\mathcal{T}[rec] = C \qquad \tau = typ(rng(C.f))}{\mathcal{T} \vdash \boxed{\texttt{this}^C.f} : \tau}$$

$$\text{T-MAppl}\ \frac{\mathcal{T} \vdash e_0 : C \qquad (\mathcal{T} \vdash e_i : \tau_i)_{i=1..p} \qquad typ(sig(C.m)) = \tau'_1\ \ldots\ \tau'_p\ \tau \qquad \left(\tau_i <: \tau'_i\right)_{i=1..p}}{\mathcal{T} \vdash \boxed{e_0{}^C.m(e_1, \ldots, e_p)} : \tau}$$

$$\text{T-IntAdd}\ \frac{\mathcal{T} \vdash e_1 : int \qquad \mathcal{T} \vdash e_2 : int}{\mathcal{T} \vdash \boxed{e_1\ +^{int} e_2} : int} \qquad \text{T-ObjAdd}\ \frac{\mathcal{T} \vdash e_1 : C_1 \qquad \mathcal{T} \vdash e_2 : C_2 \qquad C = lcs(C_1, C_2)}{\mathcal{T} \vdash \boxed{e_1\ +^{obj} e_2} : C}$$

$$\text{T-IntSub}\ \frac{\mathcal{T} \vdash e_1 : int \qquad \mathcal{T} \vdash e_2 : int}{\mathcal{T} \vdash \boxed{e_1\ -^{int} e_2} : int} \qquad \text{T-ObjSub}\ \frac{\mathcal{T} \vdash e_1 : C_1 \qquad \mathcal{T} \vdash e_2 : C_2}{\mathcal{T} \vdash \boxed{e_1\ -^{obj} e_2} : C_1}$$

$$\text{T-EqID}\ \frac{\mathcal{T} \vdash e_1 : \tau_1 \qquad \mathcal{T} \vdash e_2 : \tau_2 \qquad \tau_1 <: \tau_2 \vee \tau_2 <: \tau_1}{\mathcal{T} \vdash \boxed{e_1 == e_2} : bool}$$

$$\text{T-TCast}\ \frac{\mathcal{T} \vdash e : C' \qquad C <: C'}{\mathcal{T} \vdash \boxed{(C)\ e} : C} \qquad \text{T-NCast}\ \frac{\mathcal{T} \vdash e : C}{\mathcal{T} \vdash \boxed{(\eta)\ e} : C} \qquad \text{T-Count}\ \frac{\mathcal{T} \vdash e : C}{\mathcal{T} \vdash \boxed{|e|} : int}$$

A.1.2 Type Rules for Statements.

$$\text{T-NoStat}\ \frac{}{\mathcal{T} \vdash \boxed{\epsilon}\ \checkmark_\tau} \qquad \text{T-Seq}\ \frac{\mathcal{T} \vdash s\ \checkmark_\tau \qquad \mathcal{T} \vdash s'\ s_1\ \ldots\ s_n\ \checkmark_\tau}{\mathcal{T} \vdash \boxed{s\ s'\ s_1\ \ldots\ s_n}\ \checkmark_\tau}$$

$$\text{T-If}\ \frac{\mathcal{T} \vdash e : bool \qquad \mathcal{T} \vdash s_1\ \ldots\ s_n\ \checkmark_\tau \qquad \mathcal{T} \vdash s'_1\ \ldots\ s'_{n'}\ \checkmark_\tau}{\mathcal{T} \vdash \boxed{\texttt{if}\ (e)\ \{s_1\ \ldots\ s_n\}\ \texttt{else}\ \{s'_1\ \ldots\ s'_{n'}\}}\ \checkmark_\tau} \qquad \text{T-While}\ \frac{\mathcal{T} \vdash e : bool \qquad \mathcal{T} \vdash s_1\ \ldots\ s_n\ \checkmark_\tau}{\mathcal{T} \vdash \boxed{\texttt{while}\ (e)\ \{s_1\ \ldots\ s_n\}}\ \checkmark_\tau}$$

$$\text{T-MethInvoc}\ \frac{\mathcal{T} \vdash e_0{}^C.m(e_1, \ldots, e_r) : \tau}{\mathcal{T} \vdash \boxed{e_0{}^C.m(e_1, \ldots, e_r);}\ \checkmark_\tau} \qquad \text{T-FldAssign}\ \frac{\mathcal{T}[rec] = C \qquad \mathcal{T} \vdash e : \tau \qquad \tau <: typ(rng(C.f))}{\mathcal{T} \vdash \boxed{\texttt{this}^C.f = e;}\ \checkmark_\tau}$$

$$\text{T-VarAssign}\ \frac{\mathcal{T} \vdash e : \tau \qquad \tau <: \mathcal{T}[x]}{\mathcal{T} \vdash \boxed{x = e;}\ \checkmark_\tau} \qquad \text{T-Return}\ \frac{\mathcal{T} \vdash e : \tau \qquad \tau <: \mathcal{T}[ret]}{\mathcal{T} \vdash \boxed{\texttt{return}\ e;}\ \checkmark_\tau}$$

A.1.3 Type Rules for Definitions.

$$\text{T-Prog}\ \frac{(K_i\ \checkmark_\tau)_{i=1..n}}{P = \boxed{K_1\ \ldots\ K_n}\ \checkmark_\tau} \qquad \text{T-Class}\ \frac{(F_i\ \checkmark_\tau)_{i=1..n} \qquad (M_i\ \checkmark_\tau\ \text{in}\ C)_{i=1..n'}}{K = \boxed{C\ \texttt{extends}\ C'\ \{F_1\ \ldots\ F_n\ M_1\ \ldots\ M_{n'}\}}\ \checkmark_\tau}$$

$$\text{T-FldDef}\ \frac{\epsilon \vdash ie : \tau \qquad \tau <: typ(\rho)}{F = \boxed{\rho\ f = ie;}\ \checkmark_\tau}$$

$$\text{T-MethDef}\ \frac{\begin{array}{c}(rec \mapsto C)(ret \mapsto typ(\rho))(x_1 \mapsto typ(\rho_1)) \ldots (x_p \mapsto typ(\rho_p)) \vdash s_1 \ \ldots \ s_n \checkmark_\tau \\ P[C] = \texttt{class}\ C\ \texttt{extends}\ C'\ \cdots \\ method(C'.m) = \bot \vee typ(sig(C'.m)) = typ(sig(C.m))\end{array}}{M = \boxed{\rho\ m\ \eta\ (\rho_1\, x_1, \ldots, \rho_p\, x_p)\ \{s_1 \ \ldots \ s_n\}}\ \checkmark_\tau \text{ in } C}$$

A.2 Number Rules

A.2.1 Number Rules for Expressions.

$$\text{N-True}\ \frac{}{\mathcal{N} \vdash \boxed{\texttt{true}} \# \eta_\epsilon} \qquad \text{N-False}\ \frac{}{\mathcal{N} \vdash \boxed{\texttt{false}} \# \eta_\epsilon} \qquad \text{N-Int}\ \frac{}{\mathcal{N} \vdash \boxed{i} \# \eta_\epsilon}$$

$$\text{N-NoObj}\ \frac{}{\mathcal{N} \vdash \boxed{\texttt{no}\ C} \# -} \qquad \text{N-NewObj}\ \frac{}{\mathcal{N} \vdash \boxed{\texttt{new}\ C} \# !} \qquad \text{N-Var}\ \frac{\mathcal{N}[x] = \eta}{\mathcal{N} \vdash \boxed{x} \# \eta}$$

$$\text{N-This}\ \frac{\mathcal{N}[rec] = !}{\mathcal{N} \vdash \boxed{\texttt{this}} \# !} \qquad \text{N-These}\ \frac{\mathcal{N}[rec] = *}{\mathcal{N} \vdash \boxed{\texttt{these}} \# *} \qquad \text{N-Field}\ \frac{\mathcal{N}[rec] = ! \qquad \eta = num(rng(C.f))}{\mathcal{N} \vdash \boxed{\texttt{this}^C.f} \# \eta}$$

$$\text{N-MAppl}\ \frac{\begin{array}{c}\mathcal{N} \vdash e_0 \# \eta_0 \qquad \eta_0 \neq \eta_\epsilon \qquad (\mathcal{N} \vdash e_i \# \eta_i)_{i=1..p} \\ num(sig(C.m)) = \eta_0'\, \eta_1' \ \ldots \ \eta_p'\, \eta' \qquad \left(\eta_i <\!\# \eta_i'\right)_{i=1..p} \qquad \eta_0' = ! \wedge \eta = \eta_0.\eta' \ \vee \ \eta_0' = * \wedge \eta = \eta'\end{array}}{\mathcal{N} \vdash \boxed{e_0{}^C.m(e_1, \ldots, e_p)} \# \eta}$$

$$\text{N-Add}\ \frac{\mathcal{N} \vdash e_1 \# \eta_1 \qquad \mathcal{N} \vdash e_2 \# \eta_2 \qquad \eta = \eta_1 + \eta_2}{\mathcal{N} \vdash \boxed{e_1\ +^\kappa\ e_2} \# \eta} \qquad \text{N-Sub}\ \frac{\mathcal{N} \vdash e_1 \# \eta_1 \qquad \mathcal{N} \vdash e_2 \# \eta_2 \qquad \eta = \eta_1 - \eta_2}{\mathcal{N} \vdash \boxed{e_1\ -^\kappa\ e_2} \# \eta}$$

$$\text{N-EqId}\ \frac{\mathcal{N} \vdash e_1 \# \eta \qquad \mathcal{N} \vdash e_2 \# \eta \qquad \eta = ! \vee \eta = \eta_\epsilon}{\mathcal{N} \vdash \boxed{e_1\ \texttt{==}\ e_2} \# \eta_\epsilon}$$

$$\text{N-TCast}\ \frac{\mathcal{N} \vdash e \# \eta \qquad \eta \neq \eta_\epsilon}{\mathcal{N} \vdash \boxed{(C)\ e} \# \eta} \qquad \text{N-NCast}\ \frac{\mathcal{N} \vdash e \# \eta' \qquad \eta <\!\# \eta'}{\mathcal{N} \vdash \boxed{(\eta)\ e} \# \eta} \qquad \text{N-Count}\ \frac{\mathcal{N} \vdash e \# \eta \qquad \eta \neq \eta_\epsilon}{\mathcal{N} \vdash \boxed{|e|} \# \eta_\epsilon}$$

A.2.2 Number Rules for Statements.

$$\text{N-NoStat}\ \frac{}{\mathcal{N} \vdash \boxed{\epsilon}\ \checkmark_\eta} \qquad \text{N-Seq}\ \frac{\mathcal{N} \vdash s\ \checkmark_\eta \qquad \mathcal{N} \vdash s'\, s_1 \ \ldots \ s_n\ \checkmark_\eta}{\mathcal{N} \vdash \boxed{s\ s'\, s_1 \ \ldots \ s_n}\ \checkmark_\eta}$$

$$\text{N-If}\ \frac{\mathcal{N} \vdash e \# \eta_\epsilon \qquad \mathcal{N} \vdash s_1 \ \ldots \ s_n\ \checkmark_\eta \qquad \mathcal{N} \vdash s'_1 \ \ldots \ s'_{n'}\ \checkmark_\eta}{\mathcal{N} \vdash \boxed{\texttt{if}\ (e)\ \{s_1 \ \ldots \ s_n\}\ \texttt{else}\ \{s'_1 \ \ldots \ s'_{n'}\}}\ \checkmark_\eta} \qquad \text{N-While}\ \frac{\mathcal{N} \vdash e \# \eta_\epsilon \qquad \mathcal{N} \vdash s_1 \ \ldots \ s_n\ \checkmark_\eta}{\mathcal{N} \vdash \boxed{\texttt{while}\ (e)\ \{s_1 \ \ldots \ s_n\}}\ \checkmark_\eta}$$

$$\text{N-MethInvoc}\ \frac{\mathcal{N} \vdash e_0{}^C.m(e_1, \ldots, e_p) \# \eta}{\mathcal{N} \vdash \boxed{e_0{}^C.m(e_1, \ldots, e_p);}\ \checkmark_\eta} \qquad \text{N-FldAssign}\ \frac{\mathcal{N}[rec] = ! \qquad \mathcal{N} \vdash e \# \eta \qquad \eta <\!\# num(rng(C.f))}{\mathcal{N} \vdash \boxed{\texttt{this}^C.f = e;}\ \checkmark_\eta}$$

$$\text{N-VarAssign}\ \frac{\mathcal{N} \vdash e \# \eta \qquad \eta <\!\# \mathcal{N}[x]}{\mathcal{N} \vdash \boxed{x = e;}\ \checkmark_\eta} \qquad \text{N-Return}\ \frac{\mathcal{N} \vdash e \# \eta \qquad \eta <\!\# \mathcal{N}[ret]}{\mathcal{N} \vdash \boxed{\texttt{return}\ e;}\ \checkmark_\eta}$$

A.2.3 Number Rules for Definitions.

$$\text{N-Prog}\ \frac{\left(K_i \checkmark_\eta\right)_{i=1..n}}{P = \boxed{K_1 \ldots K_n}\ \checkmark_\eta} \qquad \text{N-Class}\ \frac{\left(F_i \checkmark_\eta\right)_{i=1..n} \quad \left(M_i \checkmark_\eta \text{ in } C\right)_{i=1..n'}}{K = \boxed{C\ \texttt{extends}\ C'\ \{F_1 \ldots F_n\, M_1 \ldots M_{n'}\}}\ \checkmark_\eta}$$

$$\text{N-FldDef}\ \frac{\epsilon \vdash ie \,\#\, \eta \qquad \eta <\# num(\rho)}{F = \boxed{\rho\ f = ie;}\ \checkmark_\eta}$$

$$\text{N-MethDef}\ \frac{\begin{array}{c}\left((rec \mapsto \eta)\ (ret \mapsto num(\rho))\ (x_1 \mapsto num(\rho_1)) \ldots (x_p \mapsto num(\rho_p))\right) \vdash s_1 \ldots s_n \checkmark_\eta \\ \eta = ! \vee \eta = * \qquad P[C] = \texttt{class}\, C\ \texttt{extends}\ C' \cdots \\ method(C'.m) = \bot \vee num(sig(C'.m)) = num(sig(C.m))\end{array}}{M = \boxed{\rho\, m\, \eta\ (\rho_1\, x_1, \ldots, \rho_p\, x_p)\ \{s_1 \ldots s_n\}}\ \checkmark_\eta \text{ in } C}$$

A.3 Dynamic Semantics

I distinguish regular and exceptional evaluation of expressions and execution of statements. For the result of exceptional evaluation and execution, I further distinguish between conceded exceptions and failure, where the latter reifies definition holes ("stuckness").

A.3.1 Regular Evaluation of Expressions. Regular evaluation steps have the form $\langle \mathcal{H}, \mathcal{L}, e\rangle \Rightarrow \langle \mathcal{H}', v\rangle$.

$$\text{E-True}\ \frac{}{\langle \mathcal{H},\, \mathcal{L},\, \boxed{\texttt{true}}\rangle \Rightarrow \langle \mathcal{H},\, true\rangle} \qquad \text{E-False}\ \frac{}{\langle \mathcal{H},\, \mathcal{L},\, \boxed{\texttt{false}}\rangle \Rightarrow \langle \mathcal{H},\, false\rangle} \qquad \text{E-Int}\ \frac{}{\langle \mathcal{H},\, \mathcal{L},\, \boxed{\texttt{i}}\rangle \Rightarrow \langle \mathcal{H},\, i\rangle}$$

$$\text{E-NoObj}\ \frac{}{\langle \mathcal{H},\, \mathcal{L},\, \boxed{\texttt{no}\ C}\rangle \Rightarrow \langle \mathcal{H},\, \epsilon\rangle} \qquad \text{E-NewObj}\ \frac{\begin{array}{c}fields(C) = (\rho_1 f_1 = ie_1) \ldots (\rho_n f_n = ie_n) \\ \mathcal{H} = \mathcal{H}_0 \quad (\langle \mathcal{H}_{i-1},\, \mathcal{L},\, ie_i\rangle \Rightarrow \langle \mathcal{H}_i,\, v_i\rangle)_{i=1..n} \quad l \notin dom(\mathcal{H}_n) \\ \mathcal{H}' = (l \mapsto (f_\chi \mapsto C)(f_1 \mapsto v_1) \ldots (f_n \mapsto v_n))\, \mathcal{H}_n\end{array}}{\langle \mathcal{H},\, \mathcal{L},\, \boxed{\texttt{new}\ C}\rangle \Rightarrow \langle \mathcal{H}',\, l\rangle}$$

$$\text{E-Var}\ \frac{\mathcal{L}[x] = \rho\, v}{\langle \mathcal{H},\, \mathcal{L},\, \boxed{x}\rangle \Rightarrow \langle \mathcal{H},\, v\rangle} \qquad \text{E-This}\ \frac{\mathcal{L}[rec] = \rho\, l}{\langle \mathcal{H},\, \mathcal{L},\, \boxed{\texttt{this}}\rangle \Rightarrow \langle \mathcal{H},\, l\rangle} \qquad \text{E-These}\ \frac{\mathcal{L}[rec] = \rho\, l_1 \ldots l_n}{\langle \mathcal{H},\, \mathcal{L},\, \boxed{\texttt{these}}\rangle \Rightarrow \langle \mathcal{H},\, l_1 \ldots l_n\rangle}$$

$$\text{E-Field}\ \frac{\mathcal{L}[rec] = \rho\, l \qquad v = \mathcal{H}[l][f]}{\langle \mathcal{H}, \mathcal{L}, \boxed{\texttt{this}^C.f}\rangle \Rightarrow \langle \mathcal{H}, v\rangle}$$

$$\text{E-SingMApplU}\ \frac{\begin{array}{c}sig(C.m) = !\ \rho_1 \ldots \rho_p \mapsto \rho \qquad num(\rho) = \eta_\epsilon \qquad params(C.m) = x_1 \ldots x_p \\ \langle \mathcal{H}, \mathcal{L}, e_0\rangle \Rightarrow \langle \mathcal{H}_0, v_0\rangle \qquad v_0 = l \qquad (\langle \mathcal{H}_{i-1}, \mathcal{L}, e_i\rangle \Rightarrow \langle \mathcal{H}_i, v_i\rangle)_{i=1..p} \\ C' = \mathcal{H}_0[l][f_\chi] \qquad body(C'.m) = s_1 \ldots s_b \\ \mathcal{L}' = (rec \mapsto C'!\ l)\ (ret \mapsto \rho\ \_)\ (x_1 \mapsto \rho_1\, v_1) \ldots (x_p \mapsto \rho_p\, v_p) \\ \langle \mathcal{H}_p, \mathcal{L}', s_1 \ldots s_b\rangle \Rightarrow \langle \mathcal{H}', \mathcal{L}''\rangle \qquad \mathcal{L}''[ret] = \_\, v\end{array}}{\langle \mathcal{H}, \mathcal{L}, \boxed{e_0{}^C.m(e_1, \ldots, e_p)}\rangle \Rightarrow \langle \mathcal{H}', v\rangle}$$

$$\text{E-SingMApplC}\ \frac{\begin{array}{c} sig(C.m) = !\ \rho_1\ \ldots\ \rho_p \mapsto \rho \qquad num(\rho) \neq \eta_\epsilon \qquad params(C.m) = x_1\ \ldots\ x_p \\ \langle \mathcal{H}, \mathcal{L}, e_0\rangle \Rightarrow \langle \mathcal{H}_0, v_0\rangle \qquad\qquad v_0 = l_1\ \ldots\ l_r \\ \left(\begin{array}{c} C_i = \mathcal{H}_0[l_i][f_\chi] \quad body(C_i.m) = s_1\ \ldots\ s_{b_i} \\ \left(\langle \mathcal{H}_{(i-1)(p+1)+j-1}, \mathcal{L}, e_j\rangle \Rightarrow \langle \mathcal{H}_{(i-1)(p+1)+j}, v_{i,j}\rangle\right)_{j=1..p} \\ \mathcal{L}_i = (rec \mapsto C_i!\ l_i)\ (ret \mapsto \rho\ \_)\ (x_1 \mapsto \rho_1\ v_{i,1}) \ldots (x_p \mapsto \rho_p\ v_{i,p}) \\ \langle \mathcal{H}_{i(p+1)-1}, \mathcal{L}_i, s_1\ \ldots\ s_{b_i}\rangle \Rightarrow \langle \mathcal{H}_{i(p+1)}, \mathcal{L}_i'\rangle \quad \mathcal{L}_i'[ret] = \_\ v_i \end{array}\right)_{i=1..r} \\ \mathcal{H}' = \mathcal{H}_{r(p+1)} \qquad\qquad v = v_1 \oplus \ldots \oplus v_r \end{array}}{\langle \mathcal{H}, \mathcal{L}, \boxed{e_0{}^C.m(e_1, \ldots, e_p)}\rangle \Rightarrow \langle \mathcal{H}', v\rangle}$$

$$\text{E-PlurMAppl}\ \frac{\begin{array}{c} sig(C.m) = *\ \rho_1\ \ldots\ \rho_p \mapsto \rho \qquad params(C.m) = x_1\ \ldots\ x_p \qquad body(C.m) = s_1\ \ldots\ s_b \\ \langle \mathcal{H}, \mathcal{L}, e_0\rangle \Rightarrow \langle \mathcal{H}_0, v_0\rangle \qquad\qquad (\langle \mathcal{H}_{i-1}, \mathcal{L}, e_i\rangle \Rightarrow \langle \mathcal{H}_i, v_i\rangle)_{i=1..p} \\ \mathcal{L}' = (rec \mapsto C*\ v_0)\ (ret \mapsto \rho\ \_)\ (x_1 \mapsto \rho_1\ v_1) \ldots (x_p \mapsto \rho_p\ v_p) \\ \langle \mathcal{H}_p, \mathcal{L}', s_1\ \ldots\ s_b\rangle \Rightarrow \langle \mathcal{H}', \mathcal{L}''\rangle \qquad\qquad \mathcal{L}''[ret] = \_\ v \end{array}}{\langle \mathcal{H}, \mathcal{L}, \boxed{e_0{}^C.m(e_1, \ldots, e_p)}\rangle \Rightarrow \langle \mathcal{H}', v\rangle}$$

$$\text{E-BinOp}\ \frac{\begin{array}{c} \langle \mathcal{H}, \mathcal{L}, e_1\rangle \Rightarrow \langle \mathcal{H}'', v_1\rangle \qquad\qquad \langle \mathcal{H}'', \mathcal{L}, e_2\rangle \Rightarrow \langle \mathcal{H}', v_2\rangle \\ \pm^\kappa = \pm^{int} \implies v = v_1 \pm v_2 \qquad \pm^\kappa = +^{obj} \implies v = v_1 \oplus v_2 \qquad \pm^\kappa = -^{obj} \implies v = v_1 \ominus v_2 \end{array}}{\langle \mathcal{H}, \mathcal{L}, \boxed{e_1 \pm^\kappa e_2}\rangle \Rightarrow \langle \mathcal{H}', v\rangle}$$

$$\text{E-EqID}\ \frac{\langle \mathcal{H}, \mathcal{L}, e_1\rangle \Rightarrow \langle \mathcal{H}'', v_1\rangle \qquad \langle \mathcal{H}'', \mathcal{L}, e_2\rangle \Rightarrow \langle \mathcal{H}', v_2\rangle \qquad v = (v_1 = v_2)}{\langle \mathcal{H}, \mathcal{L}, \boxed{e_1 == e_2}\rangle \Rightarrow \langle \mathcal{H}', v\rangle}$$

$$\text{E-TCast}\ \frac{\langle \mathcal{H}, \mathcal{L}, e\rangle \Rightarrow \langle \mathcal{H}', v\rangle \qquad v = l_1\ \ldots\ l_n \qquad \left(\mathcal{H}'[l_i][f_\chi] <: C\right)_{i=1..n}}{\langle \mathcal{H}, \mathcal{L}, \boxed{(C)\ e}\rangle \Rightarrow \langle \mathcal{H}', l_1\ \ldots\ l_n\rangle}$$

$$\text{E-NCast}\ \frac{\begin{array}{c} \langle \mathcal{H}, \mathcal{L}, e\rangle \Rightarrow \langle \mathcal{H}', v\rangle \qquad\qquad v = l_1\ \ldots\ l_n \\ \eta = *\ \vee\ (\eta = ?\wedge n \leq 1)\ \vee\ (\eta = !\wedge n = 1)\ \vee\ (\eta = -\wedge n = 0) \end{array}}{\langle \mathcal{H}, \mathcal{L}, \boxed{(\eta)\ e}\rangle \Rightarrow \langle \mathcal{H}', l_1\ \ldots\ l_n\rangle}$$

$$\text{E-Count}\ \frac{\langle \mathcal{H}, \mathcal{L}, e\rangle \Rightarrow \langle \mathcal{H}', v\rangle \qquad v = l_1\ \ldots\ l_n}{\langle \mathcal{H}, \mathcal{L}, \boxed{|e|}\rangle \Rightarrow \langle \mathcal{H}', n\rangle}.$$

If for some $\mathcal{H}$, $\mathcal{L}$, and e, $\langle \mathcal{H}, \mathcal{L}, e\rangle \Rightarrow \langle \mathcal{H}', v\rangle$, I will say that evaluation of $\langle \mathcal{H}, \mathcal{L}, e\rangle$ *succeeds*.

A.3.2 Exceptional Evaluation of Expressions. Exceptional evaluation steps have the form $\langle \mathcal{H}, \mathcal{L}', e\rangle \Rightarrow \uparrow X$, where X is either a conceded exception (introduced here), or *fail* (introduced in Appendix A.3.5), or *exhausted* (introduced in Appendix B.5).

$$\text{E-TCastE}\ \frac{\langle \mathcal{H}, \mathcal{L}, e\rangle \Rightarrow \langle \mathcal{H}', v\rangle \qquad v = l_1\ \ldots\ l_n \qquad \neg\left(\mathcal{H}'[l_i][f_\chi] <: C\right)_{i=1\ldots n}}{\langle \mathcal{H}, \mathcal{L}, \boxed{(C)\ e}\rangle \Rightarrow \uparrow TypeCastException}$$

$$\text{E-NCastE}\ \frac{\begin{array}{c} \langle \mathcal{H}, \mathcal{L}, e\rangle \Rightarrow \langle \mathcal{H}', v\rangle \qquad\qquad v = l_1\ \ldots\ l_n \\ \neg\,(\eta = *\ \vee\ (\eta = ?\wedge n \leq 1)\ \vee\ (\eta = !\wedge n = 1)\ \vee\ (\eta = -\wedge n = 0)) \end{array}}{\langle \mathcal{H}, \mathcal{L}, \boxed{(\eta)\ e}\rangle \Rightarrow \uparrow NumberCastException}$$

A.3.3 Regular Execution of Statements. Regular execution steps have the form $\langle \mathcal{H}, \mathcal{L}, s_1 \ \ldots \ s_n \rangle \Rightarrow \langle \mathcal{H}', \mathcal{L}' \rangle$.

$$\text{E-NoStat}\ \frac{}{\langle \mathcal{H},\ \mathcal{L},\ \boxed{\epsilon} \rangle \Rightarrow \langle \mathcal{H},\ \mathcal{L} \rangle} \qquad \text{E-Seq}\ \frac{\langle \mathcal{H},\ \mathcal{L},\ s \rangle \Rightarrow \langle \mathcal{H}'',\ \mathcal{L}'' \rangle \quad \langle \mathcal{H}'',\ \mathcal{L}'',\ s'\ s_1\ \ldots\ s_n \rangle \Rightarrow \langle \mathcal{H}',\ \mathcal{L}' \rangle}{\langle \mathcal{H},\ \mathcal{L},\ \boxed{s\ s'\ s_1\ \ldots\ s_n} \rangle \Rightarrow \langle \mathcal{H}',\ \mathcal{L}' \rangle}$$

$$\text{E-IfT}\ \frac{\langle \mathcal{H}, \mathcal{L}, e \rangle \Rightarrow \langle \mathcal{H}'', v \rangle \quad v = \mathit{true} \quad \langle \mathcal{H}'', \mathcal{L}, s_1\ \ldots\ s_n \rangle \Rightarrow \langle \mathcal{H}', \mathcal{L}' \rangle}{\langle \mathcal{H}, \mathcal{L}, \boxed{\texttt{if (}e\texttt{) \{}s_1\ \ldots\ s_n\texttt{\} else \{}s'_1\ \ldots\ s'_{n'}\texttt{\}}} \rangle \Rightarrow \langle \mathcal{H}', \mathcal{L}' \rangle}$$

$$\text{E-IfF}\ \frac{\langle \mathcal{H}, \mathcal{L}, e \rangle \Rightarrow \langle \mathcal{H}'', v \rangle \quad v = \mathit{false} \quad \langle \mathcal{H}'', \mathcal{L}, s'_1\ \ldots\ s'_{n'} \rangle \Rightarrow \langle \mathcal{H}', \mathcal{L}' \rangle}{\langle \mathcal{H}, \mathcal{L}, \boxed{\texttt{if (}e\texttt{) \{}s_1\ \ldots\ s_n\texttt{\} else \{}s'_1\ \ldots\ s'_{n'}\texttt{\}}} \rangle \Rightarrow \langle \mathcal{H}', \mathcal{L}' \rangle}$$

$$\text{E-While}\ \frac{\langle \mathcal{H}, \mathcal{L}, \texttt{if (}e\texttt{) \{}s_1\ \ldots\ s_n\ \texttt{while (}e\texttt{) \{}s_1\ \ldots\ s_n\texttt{\}\} else \{\}} \rangle \Rightarrow \langle \mathcal{H}', \mathcal{L}' \rangle}{\langle \mathcal{H}, \mathcal{L}, \boxed{\texttt{while (}e\texttt{) \{}s_1\ \ldots\ s_n\texttt{\}}} \rangle \Rightarrow \langle \mathcal{H}', \mathcal{L}' \rangle}$$

$$\text{E-MethInvoc}\ \frac{\langle \mathcal{H}, \mathcal{L}, e_0{}^{C}.m(e_1, \ldots, e_p) \rangle \Rightarrow \langle \mathcal{H}', v \rangle}{\langle \mathcal{H}, \mathcal{L}, \boxed{e_0{}^{C}.m(e_1, \ldots, e_p)\texttt{;}} \rangle \Rightarrow \langle \mathcal{H}', \mathcal{L} \rangle}$$

$$\text{E-FldAssign}\ \frac{\mathcal{L}[rec] = \rho\, l \quad \langle \mathcal{H}, \mathcal{L}, e \rangle \Rightarrow \langle \mathcal{H}'', v \rangle \quad \mathcal{H}' = \mathcal{H}''[l \mapsto (\mathcal{H}''[l][f \mapsto v])]}{\langle \mathcal{H}, \mathcal{L}, \boxed{\texttt{this}^{C}.f = e\texttt{;}} \rangle \Rightarrow \langle \mathcal{H}', \mathcal{L} \rangle}$$

$$\text{E-VarAssign}\ \frac{\langle \mathcal{H},\ \mathcal{L},\ e \rangle \Rightarrow \langle \mathcal{H}',\ v \rangle \quad \mathcal{L}' = \mathcal{L}[x \mapsto v]}{\langle \mathcal{H},\ \mathcal{L},\ \boxed{x = e\texttt{;}} \rangle \Rightarrow \langle \mathcal{H}',\ \mathcal{L}' \rangle} \qquad \text{E-Return}\ \frac{\langle \mathcal{H},\ \mathcal{L},\ e \rangle \Rightarrow \langle \mathcal{H}',\ v \rangle \quad \mathcal{L}' = \mathcal{L}[ret \mapsto v]}{\langle \mathcal{H},\ \mathcal{L},\ \boxed{\texttt{return}\ e\texttt{;}} \rangle \Rightarrow \langle \mathcal{H}',\ \mathcal{L}' \rangle}.$$

If for some $\mathcal{H}$, $\mathcal{L}$, and $s_1 \ \ldots \ s_n$, $\langle \mathcal{H}, \mathcal{L}, s_1 \ \ldots \ s_n \rangle \Rightarrow \langle \mathcal{H}', \mathcal{L}' \rangle$, I will say that execution of $\langle \mathcal{H}, \mathcal{L}, s_1 \ \ldots \ s_n \rangle$ *succeeds*.

A.3.4 Exception Propagation Rules. Exception propagation rules define exceptional evaluation and execution steps conditioned on exceptional evaluation or execution substeps. Their definition is generic: for each rule of A.3.1–A.3.3 having shape

$$\frac{A_1 \quad \cdots \quad A_n}{\langle \mathcal{H}, \mathcal{L}, z \rangle \Rightarrow \_}$$

(where z is either an expression or a sequence of statements) and for the sequence of strictly increasing indices $\langle i_1, \ldots, i_m \rangle$ such that $A_{i_1} \ \ldots \ A_{i_m}$ are the substeps (evaluations or executions) $\langle \mathcal{H}_{i_j}, \mathcal{L}_{i_j}, z_{i_j} \rangle \Rightarrow \langle \_, \_ \rangle$ among $A_1 \ \ldots \ A_n$, we get one propagation rule

$$\frac{A_1 \quad \cdots \quad A_{i_j - 1} \quad \langle \mathcal{H}_{i_j}, \mathcal{L}_{i_j}, z_{i_j} \rangle \Rightarrow \uparrow X}{\langle \mathcal{H}, \mathcal{L}, z \rangle \Rightarrow \uparrow X}$$

for each $1 \leq j \leq m$ (where X is a metavariable ranging over exceptions).

A.3.5 Failure Rules. Failure rules define exceptional evaluation and execution steps where the exception is not a conceded exception, but *fail* (which reifies stuckness, by turning it into an unconceded exception). Their definition is also generic: for each rule

$$\frac{A_1 \quad \cdots \quad A_n}{\langle \mathcal{H}, \mathcal{L}, z\rangle \Rightarrow \_}$$

among those of A.3.1–A.3.3 (where z is either an expression or a sequence of statements), and for the sequence of strictly increasing indices $\langle i_1, \ldots, i_m\rangle$ such that $A_{i_1} \ldots A_{i_m}$ are the side conditions among $A_1 \ldots A_n$, excluding those that serve the disambiguation of rule selection, we get one failure rule

$$\frac{A_1 \quad \cdots \quad A_{i_j-1} \quad \neg A_{i_j}}{\langle \mathcal{H}, \mathcal{L}, z\rangle \Rightarrow \uparrow fail}$$

for each $1 \le j \le m$. Note that it is assumed that $\neg A_{i_j}$ holds true if A_{i_j} is undefined. Since failure rules are exceptional evaluation and execution rules syntactically, they are propagated by exception propagation rules. For $\langle \mathcal{H}, \mathcal{L}, z\rangle \Rightarrow \uparrow fail$, I will say that evaluation or execution of z *fails*.

For rules that need disambiguation for their selection, and where the disambiguating side conditions are not exhaustive (leaving cases in which evaluation may get stuck), we need specific failure rules. Specifically, in the case of E-IfT and E-IfF, we need the rule

$$\frac{\neg\Big(\langle \mathcal{H}, \mathcal{L}, e\rangle \Rightarrow \langle \mathcal{H}'', true\rangle \vee \langle \mathcal{H}, \mathcal{L}, e\rangle \Rightarrow \langle \mathcal{H}'', false\rangle\Big)}{\langle \mathcal{H}, \mathcal{L}, \boxed{\texttt{if (e) \{s}_1 \ldots \texttt{s}_n\texttt{\} else \{s'}_1 \ldots \texttt{s'}_{n'}\texttt{\}}}\rangle \Rightarrow \uparrow fail},$$

and in the case of E-SingMApplU, E-SingMApplC, and E-PlurMAppl, we need

$$\frac{\neg\Big(sig(C.m) = !\ \rho_1 \ldots \rho_p \mapsto \rho \wedge num(\rho) = \_ \ \vee\ \sigma(C.m) = *\ \rho_1 \ldots \rho_p \mapsto \rho\Big)}{\langle \mathcal{H}, \mathcal{L}, \boxed{e_0{}^C.m(e_1, \ldots, e_p)}\rangle \Rightarrow \uparrow fail}.$$

B PROOF OF TYPE AND NUMBER SAFETY

The safety proof follows a standard scheme. Its main contribution is showing that the static numbering of member access, object addition, and object subtraction correctly abstract from the counts of objects seen at runtime, and that the evaluation and execution rules provided cover all well-numbered programs. That this requires detailed case analyses may be taken as indication that, as intended, the accidental complexity of dealing with none, one, and many has been shifted from programs to the implementation of the language.

First, we need to provide definitions of well-typedness and well-numberedness for values and heaps (which were not provided in Section 5) as well as for locals stores.

B.1 Typing and Numbering of Values

The connection between values v (as defined in Section 5.3) and their abstractions *type* and *number*, which has remained implicit so far, is established by the relations $\mathcal{H} \vdash v : \tau$ and $\vdash v \,\#\, \eta$, defined by the rules

$$\text{T-Bool}\frac{}{\mathcal{H} \vdash b : bool} \qquad \text{T-Int}\frac{}{\mathcal{H} \vdash i : int} \qquad \text{T-Objs}\frac{\big(\mathcal{H}[l_i][f_\chi] <: \tau\big)_{i=1..n}}{\mathcal{H} \vdash l_1 \ldots l_n : \tau}$$

$$\text{N-Bool}\frac{}{\vdash b \,\#\, \eta_\epsilon} \qquad \text{N-Int}\frac{}{\vdash i \,\#\, \eta_\epsilon} \qquad \text{N-Objs}\frac{(n = 0 \wedge - <\# \eta) \vee (n = 1 \wedge !\ <\# \eta) \vee (n > 1 \wedge \eta = *)}{\vdash l_1 \ldots l_n \,\#\, \eta}.$$

Note that both typing and numbering of values is polymorphic in the sense that $\forall \tau <: \tau' \,.\, v : \tau \implies v : \tau'$ and $\forall \eta <\# \eta' \,.\, v \,\#\, \eta \implies v \,\#\, \eta'$. Also, note that while typing of values depends on the heap, their numbering is independent of it.

Together with the definition of the subnumer relation $<\#$ in Table 1(a), N-Objs establishes the connection between (static) number and (dynamic) count, which has so far only informally been introduced (in Section 4). Specifically, for $\vdash l_1 \ldots l_n \,\#\, \eta$ we have that $n \in [0,0]$ if $\eta = -$, $n \in [1,1]$ if $\eta = !$, $n \in [0,1]$ if $\eta = ?$, and $n \in [0,\infty)$ if $\eta = *$.

B.2 Well-formedness of Heap and Locals Store

Next, we need to define well-formedness of evaluation and execution contexts. For heaps, well-formedness is defined by

$$\text{W-Heap}\frac{(\mathcal{H}[l]\checkmark)_{l \propto dom(\mathcal{H})}}{\mathcal{H}\checkmark}$$

$$\text{W-Obj}\frac{\begin{array}{c}\mathcal{H}[l] = o \qquad dom(o) = f_\chi\ f_1\ \ldots\ f_n \qquad fields(o[f_\chi]) = (\rho_1 f_1 = ie_1) \ldots (\rho_n f_n = ie_n) \\ (\mathcal{H} \vdash o[f_i] : typ(\rho_i))_{i=1..n} \qquad\qquad (\vdash o[f_i] \,\#\, num(\rho_i))_{i=1..n}\end{array}}{\mathcal{H}[l]\checkmark}.$$

In the context of a well-formed heap $\mathcal{H}$, well-formedness of a locals store $\mathcal{L} = (x_1 \mapsto \rho_1 v_1) \ldots (x_n \mapsto \rho_n v_n)$ is defined by

$$\text{W-Locals}\frac{(\mathcal{H} \vdash v_i : typ(\rho_i))_{i=1..n} \qquad (\vdash v_i \,\#\, num(\rho_i))_{i=1..n}}{\mathcal{H} \vdash (x_1 \mapsto \rho_1 v_1) \ldots (x_n \mapsto \rho_n v_n)\ \checkmark}.$$

B.3 Derived Type and Number Environments

Following the examples of Bierman et al. [4, 7], the locals store $\mathcal{L}$ carries the typing and numbering environments $\mathcal{T}$ and $\mathcal{N}$ used by the type and number rules of Section 5.2, which will be required by the safety proofs. These environments are extracted from $\mathcal{L}$ using the two definitions

$$\frac{\mathcal{L} = (x_1 \mapsto \rho_1\, v_1) \ldots (x_n \mapsto \rho_n\, v_n)}{\mathcal{T}(\mathcal{L}) = (x_1 \mapsto typ(\rho_1)) \ldots (x_n \mapsto typ(\rho_n))} \qquad \frac{\mathcal{L} = (x_1 \mapsto \rho_1\, v_1) \ldots (x_n \mapsto \rho_n\, v_n)}{\mathcal{N}(\mathcal{L}) = (x_1 \mapsto num(\rho_1)) \ldots (x_n \mapsto num(\rho_n))}.$$

B.4 Useful Lemmas

Because the sub-steps and side conditions of a step may alter the heap and the locals store, it is convenient to have the following three lemmas:

Lemma 1. *If $\mathcal{H}\checkmark$, $\mathcal{H} \vdash \mathcal{L}\checkmark$, and $\langle \mathcal{H}, \mathcal{L}, e\rangle \Rightarrow \langle \mathcal{H}', l_1\ \ldots\ l_n\rangle$, then $l_1\ \ldots\ l_n \propto dom(\mathcal{H}')$.*

If expressions evaluate to locations, then these locations exist on the heap.

Proof. Simultaneous induction on the rules for evaluation and execution. Locations l are either new or taken from $\mathcal{H}$ (as values of fields) or taken from $\mathcal{L}$ (as values of variables). New locations are exclusively introduced through E-New, which adds them to the heap. $\mathcal{H}\checkmark$ and $\mathcal{H} \vdash \mathcal{L}\checkmark$ imply that for all locations l stored in $\mathcal{H}$ or $\mathcal{L}$, $l \propto dom(\mathcal{H})$. □

Lemma 2. *If $\mathcal{H} \vdash v : \tau$ and $\langle \mathcal{H}, \mathcal{L}, e\rangle \Rightarrow \langle \mathcal{H}', v'\rangle$ or $\langle \mathcal{H}, \mathcal{L}, s_1\ \ldots\ s_n\rangle \Rightarrow \langle \mathcal{H}', \mathcal{L}'\rangle$, then $\mathcal{H}' \vdash v : \tau$.*

The type of a value does not change if the heap is updated.

Proof. Using T-Obs, T-Bool, and T-Int, simultaneous induction on the evaluation and execution rules. The only rules that change the heap are E-New and E-FldAssign, which neither remove a location nor update f_χ. □

Lemma 3. *If* $\mathcal{H} \vdash \mathcal{L}\checkmark$ *and* $\langle \mathcal{H}, \mathcal{L}, e\rangle \Rightarrow \langle \mathcal{H}', v'\rangle$ *or* $\langle \mathcal{H}, \mathcal{L}, s_1 \ldots s_n\rangle \Rightarrow \langle \mathcal{H}', \mathcal{L}'\rangle$, *then* $\mathcal{H}' \vdash \mathcal{L}\checkmark$.

Heap update does not affect well-formedness of the locals store.

Proof. This follows immediately from W-Locals and Lemma 2. □

B.5 Main Theorem

Having to work with big-step operational semantics, the proof of type and number safety needs to take the possibility of divergence into account, for which the relation $\Rightarrow$ is undefined (or, equivalently, for which no finite derivations exist). It does so by using the device of a non-negative counter c [18, 52] (also called *fuel* [1]), which makes sure that evaluations of expressions and executions of statements always terminate, if only with the result that the counter has been exhausted before regular termination is reached. This is sufficient, since for terminating programs, an initialization of the counter that will not be exhausted can always be found, and non-terminating programs (which exhaust every counter) do not compromise safety.

Following Ernst et al. [18], I define a *finite evaluation and execution relation* $\Rightarrow_c$ by duplicating the evaluation and execution rules of Appendix A.3.1–A.3.5, where each occurrence of $\Rightarrow$ in the premise of a duplicated rule is replaced with $\Rightarrow_c$, and where $\Rightarrow$ in the conclusion of a duplicated rule is replaced with $\Rightarrow_{c+1}$.[15] To the so obtained, counter-based rule set, I add the exhaustion axioms

$$\langle \mathcal{H}, \mathcal{L}, e\rangle \Rightarrow_0 \uparrow \mathit{exhausted} \quad \text{and} \quad \langle \mathcal{H}, \mathcal{L}, s_1 \ldots s_n\rangle \Rightarrow_0 \uparrow \mathit{exhausted}\,,$$

which limit the depth of the derivation trees of $\Rightarrow_d$ to d. Here, *exhausted* is a (conceded) exception that is propagated using the standard exception propagation rules, so that $\Rightarrow_d$ returns *exhausted* if d is too small for the full derivation (which may be infinite).

Given that the failure rules of Appendix A.3.5 cover all cases left by the regular and exceptional evaluation rules of A.3.1–A.3.4, we have that for all d, $\mathcal{H}$, $\mathcal{L}$, and e, we get either $\langle \mathcal{H}, \mathcal{L}, e\rangle \Rightarrow_d \langle \mathcal{H}', v\rangle$ or $\langle \mathcal{H}, \mathcal{L}, e\rangle \Rightarrow_d \uparrow X$, and for all d, $\mathcal{H}$, $\mathcal{L}$, and $s_1 \ldots s_n$, we get either $\langle \mathcal{H}, \mathcal{L}, s_1 \ldots s_n\rangle \Rightarrow_d \langle \mathcal{H}', \mathcal{L}'\rangle$ or $\langle \mathcal{H}, \mathcal{L}, s_1 \ldots s_n\rangle \Rightarrow_d \uparrow X$, where $X \in \{\mathit{TypeCastException}, \mathit{NumberCastException}, \mathit{fail}, \mathit{exhausted}\}$. This allows us to prove the following two, mutually dependent lemmas, one for the evaluation of expressions and one for the execution of statements.[16]

Lemma 4 (Safety of Expression Evaluation). *For expressions* e, *heaps* $\mathcal{H}$, *and locals stores* $\mathcal{L}$ *such that*

$$\mathcal{H}\checkmark \qquad (\mathcal{H}\checkmark:)$$
$$\mathcal{H} \vdash \mathcal{L}\checkmark \qquad (\mathcal{L}\checkmark:)$$
$$\mathcal{T}(\mathcal{L}) \vdash e : \tau \qquad (e\checkmark_\tau:)$$
$$\mathcal{N}(\mathcal{L}) \vdash e \,\#\, \eta \qquad (e\checkmark_\eta:)$$

and for all $d \geq 0$,

(a) $\neg\big(\langle \mathcal{H}, \mathcal{L}, e\rangle \Rightarrow_d \uparrow \mathit{fail}\big)$

[15] The use of c and $c+1$ instead of $c-1$ and c follows Siek [52] and saves us the side condition $c > 0$ in all rules.

[16] Note that in the following (like for most parts of Section 5), the program P and the requirement that P is well-formed (which includes $P\checkmark_\tau$ and $P\checkmark_\eta$; see Section 5.1) are left implicit.

(b) *if* $\langle \mathcal{H}, \mathcal{L}, e\rangle \Rightarrow_d \langle \mathcal{H}', v\rangle$, *then*

$$\mathcal{H}'\checkmark \qquad (:\mathcal{H}'\checkmark)$$
$$\mathcal{H}' \vdash \mathcal{L}\checkmark \qquad (:\mathcal{L}\checkmark)$$
$$\mathcal{H}' \vdash v : \tau \qquad (:v\checkmark_\tau)$$
$$\vdash v \# \eta \qquad (:v\checkmark_\eta)$$

Proof. The proof is by induction on d. For all e, $\langle \mathcal{H}, \mathcal{L}, e\rangle \Rightarrow_0 \uparrow \mathit{exhausted}$ by the exhaustion axioms, so that for $d = 0$, (a) trivially holds and (b) is vacuously true. For $d > 0$, the proof is structured by case analysis on the syntactic forms of expressions, relying on Lemma 5 in the case of method application (where evaluation involves the execution of the method body). In the proof, I will refer to assumptions ($\mathcal{H}\checkmark:$) through ($e\checkmark_\eta:$) jointly as the precondition of an evaluation step, and to ($:\mathcal{H}'\checkmark$) through ($:v\checkmark_\eta$) as its postcondition. Also, to the counter-based duplicate of a rule named E-Name I will refer as E-Name$_c$. Recall that for ease of expression, for $\neg\big(\langle \mathcal{H}, \mathcal{L}, \_\rangle \Rightarrow \uparrow \mathit{fail}\big)$ I will sometimes say that evaluation or execution *does not fail*, and for $\langle \mathcal{H}, \mathcal{L}, \_\rangle \Rightarrow \langle \mathcal{H}', \_\rangle$, I will say that evaluation or execution *succeeds*. Likewise, for side conditions that are satisfied, I will say that they succeed; otherwise, I will say that they fail (which includes the case that they are undefined on given arguments).

`true`

(a) By use of E-True$_c$, evaluation trivially succeeds and therefore does not fail.

(b) Since $\mathcal{H}' = \mathcal{H}$, ($:\mathcal{H}'\checkmark$) and ($:\mathcal{L}\checkmark$) follow directly from ($\mathcal{H}\checkmark:$) and ($\mathcal{L}\checkmark:$); ($:v\checkmark_\tau$) follows directly from ($e\checkmark_\tau:$), T-True, and T-Bool; ($:v\checkmark_\eta$) follows accordingly.

`false` i `no` C

(a) and (b) are analogous to the case of `true`, using rules E-False$_c$, E-Int$_c$, and E-NoObj$_c$.

`new` C

(a) Given the well-formedness of P, the first side condition of E-NewObj$_c$ does not fail and we may assume ($e\checkmark_\tau:$) and ($e\checkmark_\eta:$) for all ie_i. Given ($\mathcal{H}\checkmark:$) and ($\mathcal{L}\checkmark:$) from the precondition of the evaluation step, we may assume by the induction hypothesis that the first subevaluation in the sequence from $i = 1..n$ does not fail and if it succeeds, the postcondition holds for $\mathcal{H}_1$, $\mathcal{L}$, and v_1. This implies the precondition of the next subevaluation in the sequence and so on for all remaining subevaluations. A new location l can always be chosen and the final side condition succeeds (and hence does not fail) because $l \notin \mathit{dom}(\mathcal{H}_n)$. Hence, evaluation of `new C` does not fail.

(b) If the evaluation succeeds, its subevaluations must have succeeded, so that by the induction hypothesis, part (b), and Lemma 2, the postcondition holds for $\mathcal{H}_n$, $\mathcal{L}$, and all v_i. We still need to show that it also holds for $\mathcal{H}'$. We do this separately for each part of the postcondition. ($:\mathcal{H}'\checkmark$): Because l is new and because we know that ($:\mathcal{H}'\checkmark$) already holds for $\mathcal{H}_n$, we must only show W-Obj for the added l. With $f_\chi = C$, the fields $f_1 \ldots f_n$ used in E-NewObj$_c$ and W-Obj are identical; that the fields' values $v_1 \ldots v_n$ assigned by E-NewObj$_c$ satisfy W-Obj follows from T-Prog, N-Prog, T-Class, N-Class, T-FldDef, and N-FldDef, in conjunction with Lemma 2 and ($:v\checkmark_\tau$) and ($:v\checkmark_\eta$) for each v_i as already shown above. ($:\mathcal{L}\checkmark$): This follows directly from Lemma 3 and ($:\mathcal{L}\checkmark$) for $\mathcal{H}_n$ as shown above. ($:v\checkmark_\tau$): follows from T-NewObj (which requires the new object to have type C), the construction of the new object in E-NewObj$_c$ (which sets f_χ to C), and T-Objs. ($:v\checkmark_\eta$) is immediate, since $\vdash l \# !$, as required by N-NewObj.

x

(a) Given ($e\checkmark_\tau$:), ($e\checkmark_\eta$:), T-Var, and N-Var, we know that $x \propto dom(\mathcal{T}(\mathcal{L}))$ and $x \propto dom(\mathcal{N}(\mathcal{L}))$ and hence from the definition of $\mathcal{T}(\mathcal{L})$ and $\mathcal{N}(\mathcal{L})$ that $x \propto dom(\mathcal{L})$, so that the side condition $\mathcal{L}[x] = \rho\, v$ of, and with it evaluation through, E-Var$_c$ succeeds, and therefore does not fail.

(b) $\mathcal{H}' = \mathcal{H}$ so that ($:\mathcal{H}'\checkmark$) and ($:\mathcal{L}\checkmark$) trivially hold; ($:v\checkmark_\tau$) and ($:v\checkmark_\eta$) trivially follow from ($\mathcal{L}\checkmark:$).

`this`

(a) Given ($e\checkmark_\tau$:), ($e\checkmark_\eta$:), T-This, and N-This we can infer, in analogy to variable access, that $rec \propto dom(\mathcal{L})$. Given ($\mathcal{L}\checkmark:$) and N-This, we know that $\mathcal{L}[rec]$ is a single location, so that the side condition and with it evaluation through E-This$_c$ succeeds, and therefore does not fail.

(b) analogous to the case of x above

`these`

(a) and (b) analogous, with `these` replacing `this`, $\star$ replacing ! and $l_1 \ldots l_n$ replacing l.

$\texttt{this}^C.f$

(a) Evaluation is through E-Field$_c$. For the first side condition, see the case of `this`. From ($\mathcal{L}\checkmark:$) we know that $l \propto dom(\mathcal{H})$; ($e\checkmark_\tau$:), ($e\checkmark_\eta$:), and ($\mathcal{H}\checkmark:$) imply (via W-Heap) that $f \propto dom(\mathcal{H}[l])$, so that the second side condition also succeeds. Therefore, evaluation does not fail.

(b) Because $\mathcal{H}' = \mathcal{H}$, ($:\mathcal{H}'\checkmark$) and ($:\mathcal{L}\checkmark$) from the postcondition trivially hold. By ($:\mathcal{H}'\checkmark$) and W-Heap, we have that $\mathcal{H} \vdash v : typ(rng(C.f))$ and $\vdash v \,\#\, num(rng(C.f))$, which is what is prescribed by T-Field and N-Field, hence proving ($:v\checkmark_\tau$) and ($:v\checkmark_\eta$).

$e_0{}^C.m(e_1, \ldots, e_p)$

The selection of the regular evaluation rule for method application, either E-SingMApplU$_c$, E-SingMApplC$_c$, or E-PurMAppl$_c$, is governed by the initial side conditions of each rule which, as is easily checked, are mutually exclusive (so that selection is deterministic). Because for $sig(C.m) = \rho\,\rho_1 \ldots \rho_p \mapsto \rho'$, N-MethInvoc and N-MAppl guarantee that $\rho = ! \vee \rho = \star$, and because the well-definedness of $num(\rho')$ is guaranteed by the syntax of Num, precisely one rule of the rules (and not the failure rule; cf. Appendix A.3.5) is always selected. I distinguish by the different rules.

E-SingMApplU$_c$.

(a) That the third side condition does not fail for well-formed programs is easily checked. By the precondition of the evaluation step, the well-formedness rules T-MAppl and N-MAppl, and the induction hypothesis, part (a), the first subevaluation (determining the receiver) does not fail. If it succeeds, we know from the induction hypothesis, part (b), N-MAppl with $\eta'_0 = !$ and $\eta' = \eta_\epsilon$ (the conditions for this case, where $\eta' = num(\rho)$ via congruence in N-MAppl and E-SingMApplU$_c$), and from Table 1(b) that $\eta_0 = !$ and therefore $\mathcal{N}(\mathcal{L}) \vdash e_0 \,\#\, !$ and (via ($:v\checkmark_\eta$) from the induction hypothesis) that $\vdash v_0 \,\#\, !$, satisfying the fourth side condition $v_0 = l$. Together with T-MAppl and N-MAppl, the postcondition of the first substep (for $\mathcal{H}_0$ and $\mathcal{L}$) implies the precondition of the second (evaluation of the first parameter). Repeatedly applying the same reasoning as for the evaluation of the receiver to all parameter evaluations means that evaluation of parameters does not fail and if it succeeds, we know by the induction hypothesis, part (b), and Lemma 2 that the postcondition holds for $\mathcal{H}_p$, $\mathcal{L}$, and all v_i. The next two side conditions do not fail because of Lemma 1, ($\mathcal{H}\checkmark:$) for $\mathcal{H}_0$, and the well-formedness of the program. The construction of $\mathcal{L}'$ cannot fail; given that as shown, ($:v\checkmark_\tau$) and ($:v\checkmark_\eta$) hold for all v_i (and hence also for l), and _ stands for an arbitrary suitable value, ($\mathcal{L}\checkmark:$) holds for $\mathcal{L}'$ by its construction. That the last substep, execution of the method body, does not fail follows from the induction hypothesis in conjunction with Lemma 5 and the satisfaction of its preconditions ($\mathcal{H}\checkmark:$), ($\mathcal{L}\checkmark:$), ($s\checkmark_\tau$:), and ($s\checkmark_\eta$:) applied to $\mathcal{H}_p$, $\mathcal{L}'$, and $s_1 \ldots s_b$, where the first

two are granted by the above and the latter two, instantiated to $\mathcal{T}(\mathcal{L}') \vdash s_1 \ldots s_b\checkmark_\tau$ and $\mathcal{N}(\mathcal{L}') \vdash s_1 \ldots s_b\checkmark_\eta$, are satisfied given the well-formedness of methods (implied by the well-formedness of P), and the congruence of $\mathcal{T}(\mathcal{L}')$ and $\mathcal{N}(\mathcal{L}')$ with the type and number environments in T-MethDef and N-MethDef. If the execution of statements succeeds, the final side condition also succeeds because updates of $\mathcal{L}'$ cannot remove *ret* or alter its range ρ.

(b) If the evaluation succeeds, its subevaluations must have succeeded, so that we get $(:\mathcal{H}'\checkmark)$ and $(:\mathcal{L}'\checkmark)$ for $\mathcal{H}'$ and $\mathcal{L}''$, as well as $(:\mathcal{L}\checkmark)$ for $\mathcal{H}'$ and $\mathcal{L}$, from Lemmas 5 and 3. From $\mathcal{H}' \vdash \mathcal{L}''\checkmark$ together with the construction of $\mathcal{L}'$ and hence also $\mathcal{L}''$, we can infer $\mathcal{H}' \vdash v : typ(\rho)$ and $\vdash v \# num(\rho)$, where ρ is the return range of $C.m$. This, however, is precisely what is required by T-MAppl and N-MAppl (where, as shown in part (a), $\eta_0 = !$ so that $\eta = \eta'$), meaning that $(:v\checkmark_\tau)$ and $(:v\checkmark_\eta)$ hold for v, the value of *ret* in $\mathcal{L}''$ to be returned by m.

E-SingMApplC$_c$.

(a) The proof proceeds in analogy to that for E-SingMApplU$_c$, the main differences being that e_0 may now evaluate to an arbitrary count of objects (requiring repeated evaluations of the parameters and execution of the method body) and that the non-failure of the final object addition must be proved, which is however trivial, since $\mathcal{L}_i'[ret] = \rho\, v_i$ and $num(\rho) \neq \eta_\epsilon$ mean that all v_i are numbers of objects (note that $\oplus$ has no other type constraints).

(b) The proofs of $(:\mathcal{H}'\checkmark)$ and $(:\mathcal{L}\checkmark)$ for $\mathcal{H}'$ and $\mathcal{L}$ proceed in analogy to the case of E-SingMApplU$_c$. To prove $(:v\checkmark_\tau)$ for v, we must show that given $(:v\checkmark_\tau)$ for each $\mathcal{H}_{i(p+1)}$ and v_i, $\mathcal{H}' \vdash (v_1 \oplus \ldots \oplus v_n) : \tau$, where $\tau = typ(\rho)$ is the return type of the invoked method (and all its overridings). This is however immediate from (i) the fact that for each v_i, $\mathcal{H}' \vdash v_i : \tau$ follows from $(:v\checkmark_\tau)$ (meaning $\mathcal{H}_{i(p+1)} \vdash v_i : \tau$), and Lemma 2, (ii) the definition of object addition in Section 3.2, and (iii) T-Objs. To prove $(:v\checkmark_\eta)$ for v, we must show that given $(:v\checkmark_\eta)$ for each v_i, $\vdash (v_1 \oplus \ldots \oplus v_n) \# \eta$, where $\eta = \eta_0.\eta'$ as defined in Table 1(b), η_0 is the number of the receiver expression e_0, and η' is the return number of the invoked method in N-MAppl and $num(\rho)$ in E-SingMApplC$_c$. To show that $\vdash v \# \eta$, we proceed by case analysis on η_0 and η', reading of $\eta = \eta_0.\eta'$ from Table 1(b).

- If $\eta_0 = -$, then $\eta = -$. Using $\mathcal{N}(\mathcal{L}) \vdash e_0 \# -$ and the induction hypothesis, $r = 0$, so that $v = \epsilon$ and, by N-Objs, $\vdash v \# -$.
- If $\eta_0 = !$, then $\eta = \eta'$. Based on the same reasoning, $r = 1$, so that $v = v_1$, where we already know that $\vdash v_1 \# \eta'$.
- If $\eta_0 = ?$ and $\eta' \neq !$, then $\eta = \eta'$. If $r = 0$, $v = \epsilon$ and $\vdash v \# \eta'$ by N-Objs; if $r = 1$, $\vdash v \# \eta'$ for the same reasons as for the $\eta_0 = !$ case.
- If $\eta_0 = ?$ and $\eta' = !$, then $\eta = ?$. If $r = 0$, $v = \epsilon$ and $\vdash v \# ?$ by N-Objs; if $r = 1$, $v = v_1$, where we already know that $\vdash v_1 \# !$, so that $\vdash v \# ?$ (by N-Objs).
- If $\eta_0 = *$ and $\eta' = -$, then $\eta = -$. Since we know that in this case, $\vdash v_i \# -$ for all v_i, we also know that $v = \epsilon$ and hence $\vdash v \# -$.
- If $\eta_0 = *$ and $\eta' \neq -$, then $\eta = *$. $\vdash v \# *$ trivially holds by N-Objs and the definition of $<\#$ in Table 1(a).

E-PlurMApplC$_c$.

(a) The proof proceeds in analogy to that for E-SingMApplU$_c$, with the relaxation that e_0 is not required to evaluate to a single location.

(b) The proof of $(:\mathcal{H}'\checkmark)$ and $(:\mathcal{L}\checkmark)$ is analogous to the E-SingMApplC$_c$ case. Since in N-MApplC$_c$, $\eta_0' = *$ as per this case, we get $\eta = \eta'$ as for the the E-SingMApplU$_c$ case, so that the proof of $(:v\checkmark_\tau)$ and $(:v\checkmark_\eta)$ is also analogous.

$$\boxed{e_1\ +^\kappa\ e_2} \qquad \boxed{e_1\ -^\kappa\ e_2}$$

The applicable rule, E-BinOp$_c$, covers addition and subtraction for integers and objects. The proofs for addition and subtraction differ slightly; I begin with the former, and for the latter show only the differences.

Addition.

(a) The first two side conditions do not fail because for this case, $\pm = +$ and because from ($e\checkmark_\tau$:) and the well-formedness rules T-IntAdd and T-ObjAdd, we know that κ is either *int* or *obj*. By the precondition of the evaluation step, T-IntAdd, T-ObjAdd, N-Add, and the induction hypothesis, part (a), we may assume that the first subevaluation of E-BinOp$_c$ does not fail. If the first subevaluation succeeds, by the induction hypothesis, part (b), we may assume that its postcondition holds for $\mathcal{H}''$, $\mathcal{L}$, and v_1, which is sufficient to assume that the second subevaluation does not fail either (by the same reasoning as for the first). Again, if it succeeds, we may assume by the induction hypothesis, part (b), and Lemma 2 that: for $\kappa = int$, $\mathcal{H}' \vdash v_1 : int$ and $\mathcal{H}' \vdash v_2 : int$, so that $v_1 + v_2$ is well-defined and the side condition succeeds; for $\kappa = obj$, $\mathcal{H}' \vdash v_1 : C_1$ and $\mathcal{H}' \vdash v_2 : C_2$, so that $v_1 \oplus v_2$ is well-defined and the side condition succeeds, so that (a) follows.

(b) If the evaluation succeeds, we know that both subevaluations must have succeeded, so that by the induction hypothesis, part (b), ($:\mathcal{H}'\checkmark$) and ($:\mathcal{L}\checkmark$) hold for $\mathcal{H}'$ and $\mathcal{L}$. For $\kappa = int$, v has been computed by integer addition so that $\mathcal{H}' \vdash v : int$ and $\vdash v \# \eta_\epsilon$ as required by ($:v\checkmark_\tau$) and ($:v\checkmark_\eta$); for $\kappa = obj$, proving ($:v\checkmark_\tau$) and ($:v\checkmark_\eta$) requires more detailed analyses.

For ($:v\checkmark_\tau$), we need to show that for $v = v_1 \oplus v_2$, $\mathcal{H}' \vdash v : lcs(C_1, C_2)$, where we can assume $\mathcal{H}' \vdash v_1 : C_1$ and $\mathcal{H}' \vdash v_2 : C_2$ by the induction hypothesis. From the definition of *lcs* in Section 5.2.1, we have that $C_1 <: lcs(C_1, C_2)$ and $C_2 <: lcs(C_1, C_2)$, and hence by T-Objs that $\mathcal{H}' \vdash v_1 : lcs(C_1, C_2)$ and $\mathcal{H}' \vdash v_2 : lcs(C_1, C_2)$, and furthermore that $\mathcal{H}' \vdash l : lcs(C_1, C_2)$ for each $l \propto v_1$ and each $l \propto v_2$. ($:v\checkmark_\tau$) thus follows from T-Objs and the definition of $\oplus$ in Section 3.2.

For ($:v\checkmark_\eta$), we need to show that $\vdash v : \eta$ where $\eta = \eta_1 + \eta_2$ according to Table 1(c) and where we can assume $\vdash v_1 \# \eta_1$ and $\vdash v_2 \# \eta_2$ by the induction hypothesis. We proceed by a case analysis based on Table 1(c), which gives us η for each combination of η_1 and η_2.

- All cases yielding $\eta = *$ are trivial ($\vdash v \# *$ always holds by N-Objs).
- If $\eta_1 = -$, then $\eta = \eta_2$. Since we know by N-Objs that in this case, $v_1 = \epsilon$, from the definition of $\oplus$ in Section 3.2 we get $v = v_1 \oplus v_2 = v_2$ and therefore, given $\vdash v_2 \# \eta_2$, $\vdash v \# \eta$, as required.
- The case that $\eta_2 = -$ is analogous.

This already covers all cases of object addition; the case where $\eta_1 = \eta_2 = \eta_\epsilon$ is handled by integer addition and mixing countable with uncountable operands (the undefined cases in Table 1(c) is excluded by typing.

Subtraction.

(a) Analogous to the addition case.

(b) The proofs of ($:\mathcal{H}'\checkmark$) and ($:\mathcal{L}\checkmark$) are analogous to the addition case, as are the proofs of ($:v\checkmark_\tau$) and ($:v\checkmark_\eta$) for $\kappa = int$; those for $\kappa = obj$ must be adapted to the specifics of object subtraction.

For ($:v\checkmark_\tau$), we need to show that for $v = v_1 \ominus v_2$, $\mathcal{H}' \vdash v : C_1$, where we can assume $\mathcal{H}' \vdash v_1 : C_1$ by the induction hypothesis. Since $v_1 \ominus v_2$ can only yield objects that are among v_1 (see its definition in Section 3.2), $\mathcal{H}' \vdash v : C_1$ follows from $\mathcal{H}' \vdash v_1 : C_1$ and T-Objs.

For ($:v\checkmark_\eta$), we need to show that $\vdash v : \eta$ where $\eta = \eta_1 - \eta_2$ according to Table 1(d) and where we can assume $\vdash v_1 \# \eta_1$ and $\vdash v_2 \# \eta_2$ by the induction hypothesis. Since we know from the definition of $\ominus$ in Section 3.2 that $|v_1 \ominus v_2| \leq |v_1|$, ($:v\checkmark_\eta$) trivially holds for all cases in which, according to Table 1(d), $\eta_1 - \eta_2 = \eta_1$. For the remaining cases, we have that $\eta_1 = !$ and $\eta = ?$ and, hence,

that $|v_1| = 1$; according to the definition of object subtraction in Section 3.2, subtracting from one object can only yield no or one object, which is covered by $\eta = ?$, so that $(:v\checkmark_\eta)$ follows in all cases.

$$\boxed{e_1 == e_2}$$

(a) By the precondition of the evaluation step, the well-formedness rules T-EqId and N-EqId, and the induction hypothesis, part (a), we may assume that the first subevaluation of the applicable evaluation rule, E-EqId$_c$, does not fail. If it succeeds, by the induction hypothesis, part (b), we may assume that its postcondition holds for $\mathcal{H}''$, $\mathcal{L}$, and v_1, which is sufficient to assume that the second subevaluation does not fail either (by the same reasoning as for the first). Again, if it succeeds, the side condition, and hence the step, will not fail.

(b) If the evaluation succeeds, we know that its subevaluations must have succeeded, so that by the induction hypothesis, part (b), the postcondition holds for $\mathcal{H}'$ and $\mathcal{L}$. It also holds for v, since by T-Bool and N-Bool, *true* and *false* are typed and numbered as required by T-EqId and N-EqId.

$$\boxed{(C)\ e}$$

(a) Type casts are covered by a regular evaluation rule, E-TCast$_c$, and an exceptional evaluation rule, E-TCastE$_c$, which differ only in the last side condition. By the precondition of the evaluation step, the well-formedness rules T-TCast and N-TCast, and the induction hypothesis, part (a), we may assume that the (common) subevaluation of E-TCast$_c$ and E-TCastE$_c$ does not fail. If the substep succeeds, we can rely on the induction hypothesis, part (b), for assuming that the postcondition of the subevaluation holds for $\mathcal{H}'$, $\mathcal{L}$, and v; specifically, that v is countable ($\eta \neq \eta_\epsilon$), so that the first side condition succeeds. From this, Lemma 1, and $(\mathcal{H}\checkmark:)$ for $\mathcal{H}'$ we can infer that $\mathcal{H}'[l_i][f_\chi]$ is well-defined for all i, so that the second side condition of either E-TCast$_c$ or E-TCastE$_c$ succeeds; in neither case, evaluation fails.

(b) If evaluation succeeds, then by E-TCast$_c$. We know that its subevaluation must have succeeded, so that by assuming the induction hypothesis, part (b), $(:\mathcal{H}'\checkmark)$, $(:\mathcal{L}\checkmark)$, and $(:v\checkmark_\eta)$ hold for $\mathcal{H}'$, $\mathcal{L}$, and v (because $v = l_1 \ldots l_n$), as required. $(:v\checkmark_\tau)$ for $v = l_1 \ldots l_n$ follows directly from T-Objs and the success of the second side condition of E-TCast$_c$.

$$\boxed{(\eta)\ e}$$

(a) Since a number cast to η_ϵ is ruled out by syntax, evaluation does not fail for reasons analogous to those stated for type cast above (eased by the absence of heap access in the second side condition).

(b) If evaluation succeeds, $(:\mathcal{H}'\checkmark)$, $(:\mathcal{L}\checkmark)$, and $(:v\checkmark_\tau)$ hold for $\mathcal{H}'$, $\mathcal{L}$, and $l_1 \ldots l_n$ for reasons analogous to type cast. Because success must have been through E-NCast$_c$, we know that its second side condition holds; from this and N-Objs it follows (by case analysis on η) that $\vdash l_1 \ldots l_n \# \eta$, and thus that $(:v\checkmark_\eta)$ holds for $l_1 \ldots l_n$.

$$\boxed{|\,e\,|}$$

(a) The applicable rule is E-Count$_c$. Given the precondition of the step, the well-formedness rules T-Count and N-Count, and the induction hypothesis, part (a), we may assume that the subevaluation does not fail. If it succeeds, we know from $\eta \neq \eta_\epsilon$ and the numbering of values (N-Objs, N-Bool, and N-Int) that v is a number of objects, so that the side condition $v = l_1 \ldots l_n$ succeeds and (a) holds.

(b) If evaluation succeeds, we know that its subevaluation must have succeeded, so that by assuming the induction hypothesis, part (b), $(:\mathcal{H}'\checkmark)$ and $(:\mathcal{L}\checkmark)$ hold for $\mathcal{H}'$ and $\mathcal{L}$. $(:v\checkmark_\tau)$ and $(:v\checkmark_\eta)$ trivially hold, because by T-Int, $\mathcal{H}' \vdash n : \mathit{int}$ as required by T-Count, and by N-Int, $\vdash n \# \eta_\epsilon$ as required by N-Count.

This concludes all cases and, with it, the proof of Lemma 4. □

Lemma 5 (Safety of Statement Execution). *For sequences of statements $s_1 \ldots s_n$, heaps $\mathcal{H}$, and locals stores $\mathcal{L}$ such that*

$$\mathcal{H}\checkmark \qquad (\mathcal{H}\checkmark:)$$
$$\mathcal{H} \vdash \mathcal{L}\checkmark \qquad (\mathcal{L}\checkmark:)$$
$$\mathcal{T}(\mathcal{L}) \vdash s_1 \ldots s_n\checkmark_\tau \qquad (s\checkmark_\tau:)$$
$$\mathcal{N}(\mathcal{L}) \vdash s_1 \ldots s_n\checkmark_\eta \qquad (s\checkmark_\eta:)$$

and for all $d \geq 0$,

(a) $\neg\big(\langle \mathcal{H}, \mathcal{L}, s_1 \ldots s_n\rangle \Rightarrow_d \uparrow \mathit{fail}\big)$

(b) *if* $\langle \mathcal{H}, \mathcal{L}, s_1 \ldots s_n\rangle \Rightarrow_d \langle \mathcal{H}', \mathcal{L}'\rangle$, *then*

$$\mathcal{H}'\checkmark \qquad (:\mathcal{H}'\checkmark)$$
$$\mathcal{H}' \vdash \mathcal{L}'\checkmark \qquad (:\mathcal{L}'\checkmark)$$

Proof. As for Lemma 4, the proof is by induction on d. For all $s_1 \ldots s_n$, $\langle \mathcal{H}, \mathcal{L}, s_1 \ldots s_n\rangle \Rightarrow_0 \uparrow \mathit{exhausted}$ by the exhaustion axioms, so that (a) trivially holds and (b) is vacuously true for $d = 0$. For $d > 0$, the proof is structured by case analysis on the syntactic forms of (sequences of) statements, relying on Lemma 4 for all statements that contain expressions. As with Lemma 4, I will refer to assumptions ($\mathcal{H}\checkmark:$) through ($s\checkmark_\eta:$) (where ($\mathcal{H}\checkmark:$) and ($\mathcal{L}\checkmark:$) are the same as for Lemma 4) as the precondition of an execution step, and to ($:\mathcal{H}'\checkmark$) and ($:\mathcal{L}'\checkmark$) as its postcondition (where ($:\mathcal{H}'\checkmark$) is the same as for Lemma 4). Also, E-Name$_c$ refers to the counter-based version of E-Name.

$\boxed{\epsilon}$

(a) Execution of no statement is covered by E-NoStat$_c$, trivially succeeds, and therefore does not fail.

(b) Given that nothing changes, the postcondition follows immediately from the precondition.

$\boxed{s\ s'\ s_1 \ldots s_n}$

(a) Two or more statements are covered by E-Seq$_c$. Given the precondition of the execution step and the well-formedness rules T-Seq and N-Seq, we can assume by the induction hypothesis, part (a), that the first subexecution does not fail. If it succeeds, by the induction hypothesis, part (b), we may assume that the postcondition holds for $\mathcal{H}''$ and $\mathcal{L}''$, which is sufficient to assume that, by the same reasoning as for the first, the second subexecution does not fail.

(b) If the execution succeeds, we know that both of its subexecutions must have succeeded, so that by assuming the induction hypothesis, part (b), the postcondition holds for $\mathcal{H}'$ and $\mathcal{L}'$ (as it does for $\mathcal{H}''$ and $\mathcal{L}''$ above).

$\boxed{\texttt{if (}e\texttt{) \{}s_1 \ldots s_n\texttt{\} else \{}s'_1 \ldots s'_{n'}\texttt{\}}}$

(a) The applicable rules E-IfT$_c$ and E-IfF$_c$ share the first substep, evaluation of e. From the precondition of the step and the well-formedness rules T-If and N-If, we know that the precondition for the evaluation of e is satisfied, so that we can assume by the induction hypothesis and Lemma 4 that this first substep does not fail. If it succeeds, we can assume by Lemma 4 and the typing of values that v must be *true* or *false* so that the side condition of one of E-IfT$_c$ and E-IfF$_c$ succeeds, and also that ($:\mathcal{H}'\checkmark$) and ($:\mathcal{L}'\checkmark$) hold for $\mathcal{H}''$ and $\mathcal{L}$. This, together with T-If and N-If, suffices to assume that the precondition of the next substep, executing either $s_1 \ldots s_n$ or $s'_1 \ldots s'_{n'}$ (selected by v), is met, and hence by the induction hypothesis, part (a), that the second substep does not fail.

(b) If the execution step succeeds, its substeps must have succeeded; the postcondition of the step follows from the induction hypothesis, part (b), applied to the second substep.

`while (`e`) {`s_1 ... s_n`}`

(a) and (b) follow directly from the if-then-else case, to which the applicable rule E-While$_c$ forwards.

e_0`.`m`(`e_1`, ... ,`e_p`);`

(a) The applicable rule E-MethInvoc$_c$ forwards to method application, an evaluation step. From the precondition of the execution step and the well-formedness rules T-MethInvoc and N-MethInvoc we know that the precondition for the evaluation substep is satisfied, so that we can assume by the induction hypothesis and Lemma 4 that the evaluation, and hence the execution, does not fail.

(b) If the execution step succeeds, its subevaluation must have succeeded; ($:\mathcal{H}'\checkmark$) follows directly from Lemma 4 applied to the sub-step and ($:\mathcal{L}'\checkmark$) follows from ($:\mathcal{H}'\checkmark$) and Lemma 3 ($\mathcal{L}$ does not change).

`this.`f `= `e`;`

(a) Field assignment is handled by E-FldAssign$_c$. The first side condition, fetching the location of the receiver, does not fail for the same reasons as for the case of evaluating field access. Given the precondition of this execution step and the well-formedness rules T-FldAssign and N-FldAssign, the evaluation of the expression to be assigned does not fail by Lemma 4 and the induction hypothesis. If it succeeds the second side condition does not fail because of Lemma 1 and because we get ($:\mathcal{H}'\checkmark$) for $\mathcal{H}''$ from Lemma 4.

(b) If execution of the assignment succeeds, we may assume from Lemma 4 that ($:\mathcal{H}'\checkmark$) and ($:\mathcal{L}\checkmark$) hold for $\mathcal{H}''$ and $\mathcal{L}$; we thus only need to show that they hold for $\mathcal{H}'$ and $\mathcal{L}$, too. The update of $\mathcal{H}''$, $\mathcal{H}'$, is well-typed and well-numbered by ($:v\checkmark_\tau$) and ($:v\checkmark_\eta$) for v in conjunction with T-FldAssign and N-FldAssign, where the latter specify the required type and number for values of $C.f$. It follows that ($:\mathcal{H}'\checkmark$) holds for $\mathcal{H}'$; ($:\mathcal{L}'\checkmark$) for $\mathcal{H}'$ and $\mathcal{L}$ follows from Lemma 3.

x `= `e`;`

(a) Variable assignment is handled by E-VarAssign$_c$. Given the precondition of this execution step and the well-formedness rules T-VarAssign and N-VarAssign, the evaluation of the assigned expression does not fail by the the induction hypothesis and Lemma 4. If it succeeds, we may assume that, for the same reasons as for evaluating variable access, $x \propto dom(\mathcal{L})$, so that updating $\mathcal{L}$ does not fail.

(b) If the step succeeds, we may assume from Lemma 4 that ($:\mathcal{H}'\checkmark$) and ($:\mathcal{L}'\checkmark$) hold for $\mathcal{H}'$ and $\mathcal{L}$; we thus only need to show that ($:\mathcal{L}'\checkmark$) holds for $\mathcal{L}'$, too. From ($:\mathcal{L}\checkmark$) for $\mathcal{L}$, $\mathcal{L}[x] = \rho\, v'$, the well-formedness rules T-VarAssign and N-VarAssign (where $\mathcal{T}$ and $\mathcal{N}$ are supplied by $\mathcal{T}(\mathcal{L})$ and $\mathcal{N}(\mathcal{L})$), Lemma 4, and the fact that the range of x is not affected by the update, we know that $\mathcal{L}' = \mathcal{L}[x \mapsto v]$ is well-formed.

`return `e`;`

(a) and (b) analogous to variable assignment, using E-Return$_c$

This concludes all cases and, with it, the proof of Lemma 5. □

I can now come to the formulation and proof of the main theorem.

Theorem 1 (Safety of Num). *Given a well-formed program P and an expression e such that $\epsilon \vdash e : \tau$ and $\epsilon \vdash e \# \eta$ for some τ and η, if evaluation of $\langle \epsilon, \epsilon, e \rangle$ terminates, then either $\langle \epsilon, \epsilon, e \rangle \Rightarrow \uparrow X$ with X being a conceded exception, or $\langle \epsilon, \epsilon, e \rangle \Rightarrow \langle \mathcal{H}, v \rangle$ with $\mathcal{H} \vdash v : \tau$ and $\vdash v \# \eta$.*

Proof. That the evaluation of $\langle \epsilon, \epsilon, e \rangle$ terminates implies that there exists a d such that $\neg\big(\langle \epsilon, \epsilon, e \rangle \Rightarrow_d \uparrow \mathit{exhausted}\big)$. Because such an evaluation cannot have used the exhaustion axioms, there exists a corresponding derivation using the standard relation $\Rightarrow$ [18]. Therefore, we know that either $\langle \epsilon, \epsilon, e \rangle \Rightarrow \uparrow X$ or $\langle \epsilon, \epsilon, e \rangle \Rightarrow \langle \mathcal{H}, v \rangle$. In the first case, *fail* is ruled out as the value of X by Lemma 4 so that only conceded exceptions remain, as required; in the second case, Lemma 4 makes sure that $\mathcal{H} \vdash v : \tau$ and $\vdash v \# \eta$, as required. □

ACKNOWLEDGMENTS

Eric Hehner's bunches were brought to my attention by James Cordy following the presentation of my ideas at an IFIP 2.4 meeting held in 2016 in Dresden. Peter Mosses provided thorough feedback on an early version of this article, and suggested the term "containerless plurals", replacing my long-time favorite "containerless manies", whose ungrammaticality alluded to the introduction of plurals where we are used to seeing only singulars.

Central parts of this work were conceived during my private *Forschungsaufenthalt 20289* at the always inspiring Schloss Dagstuhl. I thank its hospitable and supportive staff for making this possible in the midst of a pandemic. During *Forschungsaufenthalt 21183*, Sebastian Erdweg acquainted me with mastering divergence through counting.

I am especially grateful to the anonymous reviewers who allowed me to write the article that I wanted to write, only gently shaping it by pointing me to problems and errors in my presentation and to important related work that I had been unaware of. Thank you for exercising these exemplary reviewing standards!

REFERENCES

[1] Nada Amin and Tiark Rompf. 2017. Type soundness proofs with definitional interpreters. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. Giuseppe Castagna and Andrew D. Gordon (Eds.), ACM, 666–679. DOI: http://dl.acm.org/citation.cfm?id=3009866

[2] Ken Arnold, James Gosling, and David Holmes. 2000. *The Java Programming Language, Third Edition*. Addison-Wesley.

[3] Stephanie Balzer and Thomas R. Gross. 2011. Verifying multi-object invariants with relationships. In *Proceedings of the 25th European Conference on Object-oriented Programming*, Mira Mezini (Ed.). Springer, 358–382. DOI: https://doi.org/10.1007/978-3-642-22655-7_17

[4] Gavin M. Bierman, Erik Meijer, and Wolfram Schulte. 2005. The essence of data access in Comega. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, Andrew P. Black (Ed.). Springer, 287–311. DOI: https://doi.org/10.1007/11531142_13

[5] Gavin M. Bierman, Erik Meijer, and Mads Torgersen. 2007. Lost in translation: Formalizing proposed extensions to c#. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr. (Eds.). ACM, 479–498. DOI: https://doi.org/10.1145/1297027.1297063

[6] Gavin M. Bierman, Erik Meijer, and Mads Torgersen. 2010. Adding dynamic types to C$^\#$. In *Proceedings of the 24th European Conference on Object-Oriented Programming*, Theo D'Hondt (Ed.). Springer, 76–100. DOI: https://doi.org/10.1007/978-3-642-14107-2_5

[7] Gavin M. Bierman, M. J. Parkinson, and A. M. Pitts. 2003. *MJ: An Imperative Core Calculus for Java and Java with Effects*. Technical Report UCAM-CL-TR-563. University of Cambridge, Computer Laboratory.

[8] Gavin M. Bierman and Alisdair Stuart Wren. 2005. First-class relationships in an object-oriented language. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, Andrew P. Black (Ed.). Springer, 262–286. DOI: https://doi.org/10.1007/11531142_12

[9] Bernd Braßel, Michael Hanus, and Frank Huch. 2004. Encapsulating non-determinism in functional logic computations. *J. Funct. Log. Program.* 2004 (2004), 1–28. Retrieved from http://danae.uni-muenster.de/lehre/kuchen/JFLP/articles/2004/S04-01/A2004-06/JFLP-A2004-06.pdf.

[10] Frederick P. Brooks, Jr. 1987. No silver bullet - essence and accidents of software engineering. *IEEE Computer* 20, 4 (1987), 10–19. DOI: https://doi.org/10.1109/MC.1987.1663532

[11] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, Jonathan Eifrig, Scott F. Smith, Valery Trifonov, Gary T. Leavens, and Benjamin C. Pierce. 1995. On binary methods. *TAPOS* 1, 3 (1995), 221–242.

[12] Jordi Cabot and Martin Gogolla. 2012. Object constraint language (OCL): A definitive guide. In *Proceedings of the 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems*, Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio (Eds.). Springer, 58–90. DOI: https://doi.org/10.1007/978-3-642-30982-3_3

[13] Georg Cantor. 1883. Ueber unendliche, lineare Punktmannichfaltigkeiten. *Mathematische Annalen* 21, 4 (1883), 545–591. DOI: https://doi.org/10.1007/BF01446819

[14] Arthur Charguéraud. 2013. Pretty-big-step semantics. In *Proceedings of the 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 41–60. DOI: https://doi.org/10.1007/978-3-642-37036-6_3

[15] Peter P. Chen. 1976. The entity-relationship model - toward a unified view of data. *ACM Transactions on Database Systems* 1, 1 (1976), 9–36. DOI: https://doi.org/10.1145/320434.320440

[16] Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2016. A calculus for variational programming. In *Proceedings of the 30th European Conference on Object-Oriented Programming*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 6:1–6:28. DOI: https://doi.org/10.4230/LIPIcs.ECOOP.2016.6

[17] Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. 2002. More dynamic object reclassification: Fickle$_{||}$. *ACM Transactions on Programming Languages and Systems* 24, 2 (2002), 153–191. DOI: https://doi.org/10.1145/514952.514955

[18] Erik Ernst, Klaus Ostermann, and William R. Cook. 2006. A virtual class calculus. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston*, J. Gregory Morrisett and Simon L. Peyton Jones (Eds.). ACM, 270–282. DOI: https://doi.org/10.1145/1111037.1111062

[19] Manuel Fähndrich and K. Rustan M. Leino. 2003. Declaring and checking non-null types in an object-oriented language. In *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, Ron Crocker and Guy L. Steele Jr. (Eds.). ACM, 302–312. DOI: https://doi.org/10.1145/949305.949332

[20] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. 1999. A programmer's reduction semantics for classes and mixins. In *Proceedings of the Formal Syntax and Semantics of Java (Lecture Notes in Computer Science, Vol. 1523)*, Jim Alves-Foss (Ed.). Springer, 241–269. DOI: https://doi.org/10.1007/3-540-48737-9_7

[21] Martin Fowler. 1999. *Refactoring - Improving the Design of Existing Code.* Addison-Wesley. Retrieved from http://martinfowler.com/books/refactoring.html.

[22] Martin Fowler. 2011. *Domain-Specific Languages.* Addison-Wesley. Retrieved form http://vig.pearsoned.com/store/product/1,1207,store-12521_isbn-0321712943,00.html.

[23] Jeremy Gibbons. 2017. APLicative programming with naperian functors. In *Proceedings of the 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017*, Hongseok Yang (Ed.). Springer, 556–583. DOI: https://doi.org/10.1007/978-3-662-54434-1_21

[24] Daco Harkes and Eelco Visser. 2014. Unifying and generalizing relations in role-based data modeling and navigation. In *Proceedings of the 7th International Conference on Software Language Engineering*, Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju (Eds.). Springer, 241–260. DOI: https://doi.org/10.1007/978-3-319-11245-9_14

[25] Daniël Corstiaan Harkes. 2019. *Declarative Specification of Information System Data Models and Business Logic.* Ph. D. Dissertation. Delft University of Technology, Netherlands. DOI: https://doi.org/10.4233/uuid:5e9805ca-95d0-451e-a8f0-55decb26c94a

[26] Eric C. R. Hehner. 1993. *A Practical Theory of Programming.* Springer-Verlag, New York. DOI: https://doi.org/10.1007/978-1-4419-8596-5

[27] Eric C. R. Hehner. 1981. Bunch theory: A simple set theory for computer science. *Information Processing Letters* 12, 1 (1981), 26–30. DOI: https://doi.org/10.1016/0020-0190(81)90071-5

[28] Husain Ibraheem and David A. Schmidt. 1997. Adapting big-step semantics to small-step style: Coinductive interpretations and "Higher-Order" derivations. *Electronic Notes in Theoretical Computer Science* 23, 3 (1997), 121. DOI: https://doi.org/10.1016/S1571-0661(05)80692-9

[29] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems* 23, 3 (2001), 396–450. DOI: https://doi.org/10.1145/503502.503505

[30] Daniel H. H. Ingalls. 1986. A simple technique for handling multiple polymorphism. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, Norman K. Meyrowitz (Ed.). ACM, 347–349. DOI: https://doi.org/10.1145/28697.28732

[31] Kenneth E. Iverson. 1962. *A Programming Language*. John Wiley & Sons, Inc.
[32] Daniel Jackson. 2006. *Software Abstractions - Logic, Language, and Analysis*. MIT Press. Retrieved from http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=10928.
[33] Gerwin Klein and Tobias Nipkow. 2006. A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.* 28, 4 (2006), 619–695. DOI: https://doi.org/10.1145/1146811
[34] Benjamin S. Lerner, Liam Elberty, Jincheng Li, and Shriram Krishnamurthi. 2013. Combining form and function: Static types for JQuery programs. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, Giuseppe Castagna (Ed.). Springer, 79–103. DOI: https://doi.org/10.1007/978-3-642-39038-8_4
[35] Stephen W. Liddle, David W. Embley, and Scott N. Woodfield. 1993. Cardinality constraints in semantic data models. *Data & and Knowledge Engineering* 11, 3 (1993), 235–270. DOI: https://doi.org/10.1016/0169-023X(93)90024-J
[36] Erik Meijer. 2011. The world according to LINQ. *Communications of the ACM* 54, 10 (2011), 45–51. DOI: https://doi.org/10.1145/2001269.2001285
[37] Erik Meijer, Wolfram Schulte, and Gavin M. Bierman. 2003. Unifying tables, objects and documents. In *Proceedings of Declarative Programming in the Context of OO Languages*. Retrieved from http://research.microsoft.com/apps/pubs/default.aspx?id=79586.
[38] Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. 2016. On essential configuration complexity: Measuring interactions in highly-configurable systems. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 483–494. DOI: https://doi.org/10.1145/2970276.2970322
[39] Microsoft. 2017. *C# Language Specification (5th Edition)*. Standard ECMA-334:2017. European Computer Manufacturers Association, Geneva, CH. Retrieved from https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-334.pdf.
[40] John C. Mitchell. 2003. *Concepts in Programming Languages*. Cambridge University Press.
[41] Joseph M. Morris and Alexander Bunkenburg. 2001. A theory of bunches. *Acta Informatica* 37, 8 (2001), 541–561. DOI: https://doi.org/10.1007/PL00013316
[42] Tobias Nipkow and David von Oheimb. 1998. Java$_{light}$ is type-safe - definitely. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, David B. MacQueen and Luca Cardelli (Eds.). ACM, 161–170. DOI: https://doi.org/10.1145/268946.268960
[43] James Noble. 2000. Basic relationship patterns. In *Proceedings of the Pattern Languages of Program Design*, Neil Harrison, Brian Foote, and Hans Rohnert (Eds.), Vol. 4. Addison-Wesley, 73–89.
[44] James Noble, Jan Vitek, and John Potter. 1998. Flexible alias protection. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, Eric Jul (Ed.). Springer, 158–185. DOI: https://doi.org/10.1007/BFb0054091
[45] OMG (Object Management Group). 2011. *Unified Modeling Language 2.4.1*. Specification. OMG (Object Management Group). Retrieved from http://www.omg.org/spec/UML/2.4.1/.
[46] OMG (Object Management Group). 2012. *Object Constraint Language 2.4*. Specification. OMG (Object Management Group). Retrieved from http://www.omg.org/spec/OCL/.
[47] Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press.
[48] James E. Rumbaugh. 1987. Relations as semantic constructs in an object-oriented language. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, Norman K. Meyrowitz (Ed.). ACM, 466–481. DOI: https://doi.org/10.1145/38765.38850
[49] James E. Rumbaugh. 1988. Controlling propagation of operations using attributes on relations. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, Norman K. Meyrowitz (Ed.). ACM, 285–296. DOI: https://doi.org/10.1145/62083.62109
[50] Craig Russell. 2008. Bridging the object-relational divide. *ACM Queue* 6, 3 (2008), 18–28. DOI: https://doi.org/10.1145/1394127.1394139
[51] Marco Servetto, Julian Mackay, Alex Potanin, and James Noble. 2013. The billion-dollar fix - safe modular circular initialisation with placeholders and placeholder types. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, Giuseppe Castagna (Ed.). Springer, 205–229. DOI: https://doi.org/10.1007/978-3-642-39038-8_9
[52] Jeremy Siek. 2013. Type Safety in Three Easy Lemmas. Blog Post. Retrieved from http://siek.blogspot.com/2013/05/type-safety-in-three-easy-lemmas.html.
[53] Justin Slepak, Olin Shivers, and Panagiotis Manolios. 2014. An array-oriented language with static rank polymorphism. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems*, Zhong Shao (Ed.). Springer, 27–46. DOI: https://doi.org/10.1007/978-3-642-54833-8_3
[54] Harald Søndergaard and Peter Sestoft. 1992. Non-determinism in functional languages. *Computer Journal* 35, 5 (1992), 514–523. DOI: https://doi.org/10.1093/comjnl/35.5.514

[55] Guy L. Steele, Jr. 2017. It's time for a new old language. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Vivek Sarkar and Lawrence Rauchwerger (Eds.). ACM, 1. Retrieved from https://www.youtube.com/watch?v=7HKbjYqqPPQ (abstract of the keynote; quote is from the video recording).

[56] Friedrich Steimann. 2013. Content over container: Object-oriented programming with multiplicities. In *Proceedings of the ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis*, Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld (Eds.). ACM, 173–186. DOI: https://doi.org/10.1145/2509578.2509582

[57] Friedrich Steimann. 2015. None, one, many - what's the difference, anyhow? In *Proceedings of the 1st Summit on Advances in Programming Languages*, Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 294–308. DOI: https://doi.org/10.4230/LIPIcs.SNAPL.2015.294

[58] Friedrich Steimann. 2021. The kingdoms of objects and values. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2021*, Elisa Baniassad (Ed.). ACM, 125–135. DOI: https://doi.org/10.1145/3486607.3486771

[59] Friedrich Steimann, Jesper Öqvist, and Görel Hedin. 2014. Multitudes of objects: first implementation and case study for java. *Journal of Object Technology* 13, 5 (2014), 1: 1–33. DOI: https://doi.org/10.5381/jot.2014.13.5.a1

[60] Alexander J. Summers and Peter Müller. 2011. Freedom before commitment: A lightweight type system for object initialisation. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011*, Cristina Videira Lopes and Kathleen Fisher (Eds.). ACM, 1013–1032. DOI: https://doi.org/10.1145/2048066.2048142

[61] Kim Topley. 2010. *JavaFX™ Developer's Guide*. Addison-Wesley Professional.

[62] Mads Torgersen, Erik Ernst, Christian Plesner Hansen, Peter von der Ahé, Gilad Bracha, and Neal M. Gafter. 2004. Adding wildcards to the java programming language. *Journal of Object Technology* 3, 11 (2004), 97–116. DOI: https://doi.org/10.5381/jot.2004.3.11.a5

[63] David M. Ungar and Sam S. Adams. 2010. Harnessing emergence for manycore programming: Early experience integrating ensembles, adverbs, and object-based inheritance. In *Proceedings of the Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, 19–26. DOI: https://doi.org/10.1145/1869542.1869546

[64] Maarten H. van Emden and Robert A. Kowalski. 1976. The semantics of predicate logic as a programming language. *Journal of the ACM* 23, 4 (1976), 733–742. DOI: https://doi.org/10.1145/321978.321991

[65] Philip Wadler. 1992. The essence of functional programming. In *Proceedings of the Conference Record of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 1992*, Ravi Sethi (Ed.). ACM Press, 1–14. DOI: https://doi.org/10.1145/143165.143169

Received July 2020; revised November 2021; accepted December 2021