None, One, Many – What's the Difference, Anyhow?

Friedrich Steimann

Lehrgebiet Programmiersysteme
Fernuniversität in Hagen, Germany
steimann@acm.org

Abstract

We observe that compared to natural and modelling languages, the differences in expression required to deal with no, one, or many objects in programming languages are particularly pronounced. We identify some problems inherent in type-based unifications of different numbers, and advocate a solution that builds on the introduction of multiplicity as a new grammatical category of programming languages.

1998 ACM Subject Classification D.3.3 Language Constructs and Features

Keywords and phrases objects, collections, relationships, pointers, multiplicity, null

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2015.294

1 Introduction

Programming languages like Smalltalk, Java, and C♯ abstract from pointers in that variables holding (heap) objects have reference semantics by default. However, the pointers (or references) become apparent, and need to be dealt with explicitly, in two not so rare cases: when a variable holds *no* object, and when a variable holds *many* objects. The first is usually represented by the null pointer; since its dereferencing causes the notorious null pointer exception, accessing variables that may hold `null` must be explicitly guarded, requiring stereotypical (and hence annoying) coding patterns. The second is usually implemented using a collection, which reifies pointers to many objects as one object; accessing the many objects then requires going through the collection, an indirection that is not unlike the explicit dereferencing of a pointer (and that also requires stereotypical code). It follows that each of the three cases, namely that a variable holds no, one, or many objects, must be handled differently.

By looking at the differences in handling comparable cases in other disciplines, notably natural language and modelling (Section 2), we find that the differences are not necessarily grounded in the nature of the matter. Indeed, as we discuss in some detail in Section 3 (using the programming languages Xen [11], Cω [1], and C♯ with LINQ [2] as examples), there have been previous attempts at smoothing out the differences of none, one, and many in object-oriented programming, but as we find, these attempts (which are mostly type-based) have been only partly successful. In Section 4, we therefore advocate our own solution based on the separation of type and multiplicity (previously studied in [19, 20]) which, given the new insights presented here, amounts to introducing the grammatical category *number* to programming. Section 5 then discusses some questions that may arise in the context of our solution. We conclude by reporting on how we have come to depart from our previous view on how number should be handled by programming languages (Section 6), and by outlining research questions for future work (Section 7).

1st Summit on Advances in Programming Languages (SNAPL'15).
Eds.: Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett; pp. 294–308

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2 The Differences from Different Perspectives

2.1 Linguistic Perspective

In the English language, the difference between *one* and *many* materializes in the grammatical category *number*, whose values are *singular* and *plural*. Number surfaces in the declension of nouns ("person" – "persons"), and also in the conjugation of verbs ("laugh" – "laughs"); in a sentence, the number of the verb representing the predicate must match the number of the noun representing the subject. However, singulars can be combined to form plurals. For instance, "Peter and Mary laugh." is a grammatically well-formed sentence.

Interestingly, in English the word "many" can only be used with nouns that refer to separate items, or individuals, such as "person", "car", etc. These nouns are called *countable*; as their name suggests, only countable nouns can be used with cardinals (e.g., "one person", "two cars"). To make *uncountable* nouns such as "information" or "rain" countable nevertheless (so that we can use them with "one" or "many"), they can be wrapped by countable nouns, as in "piece of information" or "occasion of rain".

Converting singulars to plurals, and uncountables to countables, is complemented by the possibility to cast plurals to singulars, using the means of reification: e.g., "Peter and Mary are a couple." is grammatically well-formed. Reifications of many individuals, like "couple", are usually themselves countable, meaning that there can be many instances of them, which can in turn be reified. It is characteristic for these kinds of reifications that they have properties of their own, i.e., properties that are not (derived from) properties of the individuals they reify. Neither nesting nor the attachment of properties are available for plurals; instead, plurals are unstructured ("flat"), and properties associated with plurals are lifted over the individual entities they represent: e.g., "Peter and Mary are happy." means that both are happy – a happy couple is something else.

There is however one property associated with plurals which is not lifted, and this is the *number* of individuals being denoted. Few would dissent that when determining this number (via counting), each individual is counted only once; e.g., for the sentence "Two people laughed." it is clear that the two are different entities. This appears to give plurals set semantics, and yet counting each individual only once is different from saying that each individual is contained only once, as the latter would require a container and hence a reification of the plural. Also note that countability does not imply order; in fact, there is nothing in a plural that would suggest an ordering – this is either an extrinsic property implied by properties of the individuals (such as their age, position in space, etc.), or an intrinsic property of the container (e.g., a sequence).

2.2 Modelling Perspective

Natural language is highly informal: Using it, misunderstandings are difficult to avoid and resolve. Philosophers have therefore long strived for "the perfect language" (see, e.g., [5]).

The language of mathematics can be considered perfect: it has been used with great success in modelling much of our world. Somewhat surprisingly, however, it was not before the end of the 19^{th} century that mathematicians formalized the notion of *many* as what is today known under the term "set". Interestingly, set is a reification, not a plural (a set of mathematical entities is itself a mathematical entity)[1], and indeed, there are sets of sets; at

[1] but note that mathematical entities are not considered to have identity; sets in particular are extensionally defined, i.e., two sets are the same (or identical) if their elements are the same

the same time, however, there are multitudes that cannot be reified as sets – for instance, what would naively be referred to as "the set of all sets" (which would need to contain itself) is commonly not considered to be a set.

While there are multitudes that cannot be reified as sets, sets are sufficient to represent many other kinds of collections that are in use in mathematics today (including vectors, tuples, and sequences). Sets also capture the notion of *none*, namely as the empty set, and hence can be used to express non-existence (for instance, that an equation has no solution). Regarding *one*, however, mathematics distinguishes between a singleton set and its (sole) member, as evidenced by the different operations available for each (unless of course the member is itself a set).

In software modelling, various languages are in use. Interestingly, in several of these languages, *many* appears as a plural (i.e., the many objects are not reified): for instance, entity-relationship diagrams [4] (and in its wake also the class diagrams of the UML [12]) use multiplicity annotations at the places of relationships (or ends of associations) to distinguish numbers. In fact, it is these modelling languages that make the least difference between *none*, *one*, and *many* – multiplicities are uniformly represented by (intervals of) cardinals[2], and changing a multiplicity never affects more than a single place. This ease of change is intriguing; it is not found in mathematics (where a change from scalar to vector or vice versa means a change of type) and not even in natural language (where, e.g., a change of number propagates from the predicate to its subject or vice versa).

Other modelling languages have embraced this smooth transition from *none* to *many*, and extended it to the expression of constraints and specifications. For instance, Alloy [8] does not distinguish between single objects and sets, but treats the former as singletons. Variables in Alloy are multiplicity-annotated, e.g. with *one* (for precisely one object), or with *set* (for any number of objects). A navigation expression $x.y$, where x and y are variables, always evaluates to a flat set, independently of the multiplicities of x and y. Navigation expressions of OCL [13] are evaluated similarly: even though OCL does not represent single objects as singletons, a navigation expression can mix multiplicities freely, and the resulting collection (if any) is always flat. The difference in type of an expression with multiplicity *one* (the type of the object) and with multiplicity *many* (a collection) that exists in OCL is mitigated by allowing collection operations (such as *forAll*) on single objects also.

2.3 Computing Perspective

In the pre-object-oriented era of programming, when data items did not have identity, the difference between *one* and *many* was basically that between a value and an array of values.[3] However, this distinction was already quite prominent, as an array represents a different data type than its values. Languages like APL eliminated this difference, by interpreting scalars as special arrays, and by lifting functions defined on scalars over arrays; however, the lifted functions may pose size constraints on their arguments, and may use padding to meet them (see, e.g., [15], for a recent account). While this approach is well-suited to remove boilerplate iterations from array computations, it seems less ideal for the handling of many objects (plurals) in object-oriented programs, which are mostly about updating object structures.

[2] Semantically, multiplicities as used in these languages can be interpreted as the cardinalities of sets, but these sets never surface, not even at the instance level, where many objects at a place of a relationship (at an association end) appear as just that: many objects (e.g., in a UML instance diagram).

[3] There were of course dynamic data structures and pointers (including the null pointer), but the notion of identity – a prerequisite of reifying many as one – had yet to emerge.

Table 1 Seven differences to be observed by the object-oriented programmer when implementing relationships to many, rather than one or no, objects using collections (see [19, 20] for details).

Manifestation of Difference in	Multiplicity	
	none or *one*	*many*

1: type of expression	Account a;
	a = new Account(); //OK
	... a.owner ... //OK
		Set<Account> as;
		as = new Account(); //type error
		... as.owner ... //type error

2: "no object"	a == null? ...	a.isEmpty()? ...
3: subtyping conditions	Account a = new SavAccount();	Set<? extends Account> as = new HashSet<SavAccount>();

4: encapsulation	Account getA() {
	return a;
	}
		Set<Account> getAs() {
		return as.clone();
		}

5: assignment semantics	Account backup = a;
	a = null; // oops!
	a = backup; // phew!
		Set<Account> backups = as;
		as.clear(); // oops!
		as = backups; // what??!

6: call semantics	swap(a, backup); //cannot work	sort(as); //no problem

7: meaning of final	final Account forLife =
	new Account();
	forLife = null; //compile error
		final Set<Account> allForLife =
		Arrays.asSet(forLife);
		allForLife.remove(forLife); //OK

With the advent of object-oriented programming languages, the differences between *one* and *many* became considerably more pronounced. At first glance, this may appear paradoxical, as in object-oriented programming, collections are objects in the same right as all others. Indeed, as mentioned in the introduction, with all objects residing on the heap, a relationship to no object can be represented as the null pointer, a relationship to one object as a pointer to that object, and a relationship to many objects as a pointer to a collection (i.e., a relationship to one object) reifying the multitude. However, the indirection introduced for implementing relationships to many objects does not only dominate the type of the expressions representing the relationship (the dominance of the container over the content noted in [19]), the reification of the relationship (in the form of the collection object) also allows its aliasing. The same kind of aliasing is not possible for a relationship to one object, unless this relationship is also reified (e.g., by using a singleton container type such as `Option<T>`).

In previous work, we have identified seven practically relevant differences between implementing relationships to no or one object on the one hand, and any number of (here referred to as *many*) objects on the other [19, 20]. The differences are summarized in Table 1; for reasons of space, we do not explain them further here, but point the reader to our earlier works. Note that all differences not only have to be fully understood by programmers (which may be a problem especially for beginners, or for people with a strong modelling background); they also present tough maintenance problems, namely when multiplicities change.

3 Type-based Reconciliation of None, One, and Many

There have of course been previous attempts of reconciling representations of no, one, and many objects in object-oriented programming (see, e.g., [19, 20] for overviews). However, except for the indexed instance variables of Smalltalk, which let objects possess an arbitrary number of pointers to other objects [7], the uniform solution seems to be that of going through

some kind of container. To highlight the problems this causes, we use the Xen/Cω/C$^\sharp$ with language-integrated querying (LINQ) line of programming languages [1, 2, 11] as a representative here. Note that the scope of these languages is bridging the gap between object-oriented programming and relational and XML-based representations of data, which is much wider, but here, we focus on their contributions to our problem, the reconciliation of *none*, *one*, and *many*.

3.1 The Streams of Xen

In Xen [11], occurrences of no, one, and many objects are unified as streams, i.e., ordered homogeneous collections of objects that can be iterated over. Streams are immutable, i.e., elements of a stream cannot be overwritten. Like other reifications of many objects, streams have identity; however, like plurals, they are always flat, meaning that there are no streams over streams. Hence, from a linguistic perspective, they are hybrids of reified multitudes and plurals (cf. Section 2.1).

Xen introduces three different types of streams: $T!$, a stream of one object of type T, $T?$, a stream of no or one object (also called an *optional*), and T^*, a stream of any number of objects. In addition, `null` is interpreted as the empty stream, resolving Item 2 in Table 1. The subtyping relationship of Xen (and with it assignment compatibility) observes the axioms

$$T! <: T \qquad T <: T? \qquad T? <: T* \tag{1}$$

[11], which suggests that literals and constructor invocations of type T have type $T!$ instead (otherwise, type $T!$ would have no instances). The insertion of the element type T in the hierarchy of stream types seems awkward[4], yet provides for convenient assignment compatibilities, partly resolving the difference noted under Item 1 in Table 1. Because streams are immutable, covariance of streams in their element types is safe, resolving Item 3; at the same time, immutability makes the remaining differences of Table 1 obsolete. However, we note that immutability is not generally a useful property for variables holding (one or many) objects; we will therefore drop it in our own approach (Section 4).

One of the most salient features of streams in Xen is that operations – including member access – defined for a stream's element type are lifted over the stream, resolving the remaining difference of Item 1. For instance, if `as` has type `Account*` and `owner` is a field of `Account`, `as.owner` evaluates to a stream of the owners of the accounts in `as`. At the type level, the flattening of streams that is required when accessing stream-typed members on stream-typed receivers is achieved in Xen by a type equivalence relation $\cong$ defined so that

$$\begin{array}{lll} T!! \cong T! & T!? \cong T? & T!* \cong T* \\ T?! \cong T? & T?? \cong T? & T?* \cong T* \\ T*! \cong T* & T*? \cong T* & T** \cong T* \end{array} \tag{2}$$

Member access on an empty stream (represented by the value `null`; cf. above) results in no member access rather than a null pointer exception; if the member is non-void, it returns an empty stream of the member type. Since optionals are special streams, they share this behaviour, gracefully handling the null case that makes other solutions (including the use of

[4] In particular, it means that stream types cannot be subtypes of *Object*, since otherwise, *Object* $<:$ *Object*$*$ $<:$ *Object*. Also, $T! <: T$ seems strange, since it allows a stream to be its own element (the same would hold for *Object*$*$, if it were a subtype of *Object*; note that neither recursion can be cured by flattening).

`Nullable<T>` in C♯; see below) so awkward. This kind of propagation of `null` is also known from other programming languages; however, the subsumption of member access on no object and on one object under the general member access on streams goes further, and in any case is very much in favour of unifying *none*, *one*, and *many* as we aspire for.

Unary and binary operators are lifted element-wise over streams in Xen [11]. For option streams on value types, particularly on Booleans and numbers, this means that operations can evaluate to "no value" (represented by `null`), effectively requiring introduction of a ternary logic and calculation with `null`. For streams with more elements, lifting of binary operators would either require that both streams have equal length (a dynamic multiplicity check), or mean that all elements of the left operand are combined with all elements of the right operand. In any case, one could argue that if binary operators receive special treatment, binary methods [3] should receive it as well. We will return to this at the end of Section 4.

3.2 The Streams of Cω

While the nature of Xen appears to be more that of a proposal, its successor language, Cω [1], comes with a formally specified type system which, compared to Xen, makes a number of simplifications. In particular, it drops the streams $T!$ containing precisely one element (whose full exploitation would require solving the problem of initializing circular structures; see, e.g., [14]); however, as we will see below, the multiplicity *one* of types $T!$ is actually useful in bypassing the problems with lifting operations on value types sketched above.

Except for the omission of $T!$, Cω adopts the subtyping relationship of Xen, and explicitly adds *Object* as a supertype of all stream types[5] [1]:

$$null <: T \qquad T <: T? \qquad T? <: T* \qquad T* <: Object \tag{3}$$

The type equivalence relation of the streams of Xen (Eq. 2) is retained in Cω, but remains implicit in the type rules, which code the flattening directly. These rules allow iterators to contain themselves; e.g., `int* i = {yield return i;}`, which lets iteration attempts over `i` recurse until stack overflow, is legal Cω code. Note that the same self-containment is not allowed for arrays and generic collections (unless their element type is `object`): e.g., `int[] i = new int[] {i}` is ill-typed (cf. Section 2.2; note how this type-based exclusion of self-containment, which is lifted for streams through flattening, is reminiscent of Russell's type theory).

The lifting of operators over streams, which we reviewed critically in Section 3.1, is not addressed in [1]; the Cω compiler appears to implement it only very selectively.

3.3 The Iterators of C♯, and Language-Integrated Querying (LINQ)

In C♯, the stream types $T*$ and $T?$ of Xen and Cω materialize as iterators (implementing the `IEnumerable<T>` interface) and nullables (instances of `struct Nullable<T> where T : struct`), respectively [2]. Unlike the streams of of Xen and Cω, however, iterators can be nested in C♯; their flattening can be enforced by using the `SelectMany` method (instead of `Select`) in LINQ expressions (see below). By contrast, nesting is illegal for `Nullable<T>`, even though its above definition would permit it: the declaration `Nullable<Nullable<int>> n`, for example, does not compile (another ad hoc type check required by coding multiplicity in the type).

[5] The Cω compiler, which can be obtained from `http://research.microsoft.com/comega`, refuses implicit conversion from `object` to `object*`, hence breaking the circularity noted in footnote 4.

Using the language-integrated querying (LINQ) facilities of C♯, member access on objects can be lifted over collections of objects. E.g., revisiting our running example, the expression `from a in as select a.owner`[6] yields an iterator over owners. If owners have a field `name`, `from a in as select a.owner.name` yields an iterator over names (or a null pointer exception if an account has no owner). On the other hand, the analogous query `from a in as select a.owners.name`, where `owners` is iterable, will not compile – instead, the query must be written as `from a in as from o in a.owners select o.name`, which is the sugared version of `as.SelectMany(a => a.owners).Select(o => o.name)`, and which differs from the *one* account case.

A related issue is the fact that in C♯, `Nullable<T>` is not a subtype of `IEnumerable<T>`, so that nullables cannot be the subject of language-integrated queries, losing much of the uniform access of *none*, *one*, and *many* that was granted by Xen and Cω. Instead, avoiding null pointer exceptions when accessing members on nullable receivers requires explicit tests for nullness and explicit retrieval of the value, exposing the wrapper nature of `Nullable<T>` to the programmer much like collections did before the introduction of LINQ.

3.4 Summary

To summarize, it appears that while Xen started off heading for a far-reaching reconciliation of *none*, *one*, and *many*, casting its respective contributions into the rules of a formal type system required certain concessions such as introducing type equivalences for the purpose of flattening and ad hoc type checks preventing circularity and ill-defined behaviour. Carrying over the contributions of Xen and Cω to C♯ and LINQ seems to have meant further abandonment of the reconciliation, to the extent that it has almost disappeared.

4 Separating Multiplicity from Type

To improve on this situation, we learn from natural language and complement the collections of programming (as reified multitudes) with (non-reified) plurals of countable entities. For this, we extend the grammars of programming languages by introducing *number* or, to use a more customary term, *multiplicity*, as a category that is largely orthogonal to *type*. In so-extended languages, expressions can evaluate to many, rather than just one or no, objects – they can be plural. The use and propagation of multiplicities is subject to specific rules that, like type rules, can be checked statically. For instance, multiplicities are not allowed for types whose instances do not have identity (value types); if multiplicity is required for these types nevertheless, the values need to be wrapped first (observe the analogy with natural language; cf. Section 2.1).

Following the example of Xen and several other (especially modelling) languages, we introduce three static multiplicity annotations, named *one* (for precisely one object), *option* (for no or one object), and *any* (for any number of, including many, objects). In addition, to cater for the absence of a multiplicity annotation, we introduce a pseudo-multiplicity *bare*, which, like *option*, means that there can be no or one object; its use will become clear below. Also, for completeness, we add *none* (for no object) as the multiplicity of `null` (which makes `null` the literal representation of "no object"). Like *bare*, *none* never appears literally in declarations. All non-*bare* expressions can be converted to iterables (or streams) so that they can be used in external and internal iterations.

[6] `as` is a keyword in C♯; however, we use it as an identifier here.

Table 2 Dependence of the multiplicity of a member access expression $r.m$ on the multiplicities of the receiver r and the member m.

Multiplicity of:	Member m			
Receiver r	*one*	*bare*	*option*	*any*
one	*one*	*bare*	*option*	*any*
bare	*one*	*bare*	*option*	*any*
option	*option*	N/A	*option*	*any*
any	*any*	N/A	*any*	*any*

The numbers of objects each multiplicity represents give rise to the subsumption hierarchy

$$none \leq bare \qquad one \leq bare \qquad bare \leq option \qquad option \leq any \tag{4}$$

(note the partial ordering required by consideration of *none* and *one*; also note the absence of a type T; cf. this with Eqs. 1 and 3). This hierarchy, which is largely orthogonal to the type hierarchy, says that expressions having a lesser multiplicity can occur where a greater multiplicity is required (a multiplicity upcast is always safe and hence remains implicit)[7]. In the reverse direction, explicit multiplicity downcasts are required which, except for casts from *option* to *bare*, may fail at runtime (e.g., when an *any* expression which evaluates to two objects is cast to *option*, or when an *option* expression evaluating to no object is cast to *one*). A downcast from *option* to *bare* by itself cannot cause a runtime multiplicity error; however, accessing a member on an *option* receiver cast to *bare* can cause a null pointer exception (which is why *bare* is considered "less" than *option*). Note that such a cast may be required, namely when a *bare* member is to be accessed on an *option* receiver (see below).

Table 2 shows how multiplicities propagate through member access expressions. Note how the table corresponds to the type equivalence relation of Xen (Eq. 2), extending it with rules for *bare*, disallowing the access of *bare* members on receivers having multiplicity *option* or *any*. Together with the rule that all members having value types must be declared *bare* (i.e., without an explicit multiplicity annotation; cf. above), this ensures that expressions having value types always remain *bare*. This restriction of the use of multiplicities avoids the problems of lifting (built-in) binary operations on value types over multitudes of values (cf. Section 3.1); it is further justified in Section 5.2.

To be able to update variables having non-*bare* multiplicity, we provide three assignment operations: `=`, letting the variable on the left-hand side hold (pointers to) the objects the expression on the right-hand side evaluates to; `+=`, letting the variable hold the objects of the expression, plus any objects it held before; and `-=`, letting the variable hold the objects it held before, minus the objects of the expression (the latter two are reserved for *any* variables). Note that since we do not use containers, `=` cannot create an alias for a container; therefore, updating the right-hand side of an assignment after the assignment cannot affect the variable on its left-hand side. Hence, all differences of Table 1 that can be reduced to differences between value semantics and reference semantics dissolve.

With multiplicity and type separated as suggested, all differences of Table 1 that are related to type also disappear. In particular, the following declarations and updates are all well-typed:

[7] Note that this displaces *Object* (with *bare* multiplicity) from the top of the assignment compatibility hierarchy, which is now occupied by *any Object*.

```
any SavAccount ss = new SavAccount();
any Account as = ss;
as += new Account();
any Owner os = as.owner;
```

Note that the last statement assigns the owners of the accounts held by `as` (a savings account and an ordinary account) to `os`, which must therefore be annotated with `any`; as in Xen and Cω, member access is applied to all objects held by `as`.

The use of multiplicities allows a number of simplifications of programs. Obviously, an iteration like

```
for (Account a : as) a.changeToEuro();
```

can be replaced by

```
as.changeToEuro();
```

if `as` has multiplicity `any`. Also, guards for not null can be dropped if the null case means "no operation" or "evaluate to null":

```
owner = a.owner;
```

completely replaces for

```
if (a != null) owner = a.owner;
else owner = null;
```

A pair of overloaded methods such as

```
void add(Account a) { as.add(a); }
void add(Collection<Account> as) { this.as.addAll(as); }
```

can be replaced by the single method

```
void add(any Acccount as) { this.as += as; }
```

More generally, changing the multiplicity of formal parameters to `any` where this makes sense can save loops in method invocations in which the many objects occur in the argument rather than the receiver position:

```
for (Account a : as) registry.add(a.owner);
```

can be replaced with the more fluent[8]

```
registry.add(as.owner);
```

The inherent asymmetry in method invocations with respect to multiplicity (while a method invoked on many receivers is dispatched to each of the receivers separately, many arguments passed in a single argument position are always passed together) prevents the application of binary methods [3] to pairs of many receivers and many arguments in an APL-like manner. For instance, invocation of `equals(.)` on many receivers passing many objects as the argument will match each receiver individually with all arguments jointly. However, while the problems of applying binary methods to many objects are largely the

[8] `http://martinfowler.com/bliki/FluentInterface.html`

same as for lifting binary operations over streams as discussed in Section 3.1, we note here that it is possible to combine all receiver objects with all argument objects, by extending the technique of double dispatching to multiplicity:

```
void pair(one Object arg) {
  // this and arg make a pair
}
void pair(any Object args) {
  args.pair(this);
}
```

(note that the multiplicity of `this` is `one`).

5 Discussion

5.1 More Multiplicity Annotations

While the static multiplicity annotations *option*, *one*, and *any* appear to be necessary for achieving what we strive for, more can certainly be added. For instance, Alloy offers *some*, meaning one or more [8]. More generally, since static multiplicities represent possible numbers of objects, integer intervals can be used to express them (e.g., $[0, 1]$ for *option*, $[1, 1]$ for *one*, $[0, \infty)$ for *any*, etc.). Interval-based multiplicities can then be propagated through expressions, but due to the imprecision of static data-flow and control-flow analyses, it is difficult to see how this would work without ubiquitous use of dynamic multiplicity checks and casts. For instance, subtracting objects from a $[1, \infty)$ variable (multiplicity *some*) using `-=` would require a dynamic check that its multiplicity remains greater than zero. Given that numeric multiplicity intervals account for only a small fraction of all desirable invariants regarding the number of objects (others are non-contiguous intervals, predicates such as *odd*, relative multiplicities [9], or arbitrary constraints relating multiplicities of different expressions), the value they add seems somewhat limited (yet may pay off in certain domains).

5.2 Why *bare*?

English and other natural languages suggest that there are things of which one cannot have many, namely those that are uncountable (cf. Section 2.1). However, values do not seem to fall into this category: in computing, we have two Boolean values and countable numbers of characters, integers, dates, etc. So, why do we require value-typed expressions to have multiplicity *bare*, denying programmers the possibility to have many values in non-reified form?

There are various reasons for this. One is that lifting operations defined for single values (scalars) over many values (vectors) in a way that makes sense for these values requires mechanisms like those of APL, which do not seem to carry over well to objects (cf. Sections 2.3 and 3.1). Another is that values typically do not have identity, which appears to be a prerequisite for countability in natural language (cf. Section 2.1). While this argument may appear weak, it gives rise to a third one.

In object-oriented programming, pointers to objects are typically used to express relationships between objects, and annotations like *option* and *any* as put forward here are meant to constrain the multiplicities of these relationships. However, objects typically do not relate to values: a person for instance does not relate to her name, height, or date of birth. Data models like the entity relationship model acknowledge this by storing values in attributes, which are distinct from relationships. Wrapping values in objects may remove the distinction

between values and objects technically, but conceptually, it remains intact: it is still unclear why an object should relate to stateless and propertyless data items. Hence, in the context of relationships, the distinction between objects and values is justified; given that natural languages distinguish between countables and uncountables, we should not be too worried about using different means for representing multitudes of objects and multitudes of values.

A last argument is technically motivated: *bare* introduces backward compatibility with code that does not use multiplicity annotations. In fact, in contexts where one object is expected, *bare* can by considered a dynamically checked variant of *one*, and the null pointer exception the result of a failed dynamic multiplicity check.

5.3 Multiplicity and Identity

Identity is a trait of single objects[9] – two objects are identical if they are in fact the same object. It follows that more than one object (a plural in non-reified form) cannot be identical to anything. For analogue reasons, it is meaningless to ask whether an object is identical to no object. Tests for identity (`==`) should therefore be reserved for arguments with multiplicity *one* (with `e == null` being shorthand for checking whether the dynamic multiplicity of expression `e` is *none*).

It is however reasonable to ask whether two expressions with multiplicity other than *one* evaluate to the same objects or, more specifically, to pairs of identical objects. This would be the case if mutually applied subtractive updates (i.e., `x1 -= x2` and `x2 -= x1`) both evaluate to no objects. Equality of two expressions (in the sense of `e1.equals(e2)`) is harder to define, since each expression by itself may evaluate to any number of equal (but not identical) objects.

6 Experience Gained from a Previous Case Study

Jesper Öqvist from Lund University implemented a variant of the multiplicities as described in Section 4 as an extension to the Java 7 programming language, which we evaluated in a case study [20]. The case study provided valuable insights, which in turn led to the writing of this paper.

The main differences between our original design and what has been proposed here are:

1. The original design and its implementation do not consider multiplicity *one*.
2. The original design and its implementation allow *any* annotations to be complemented with a collection type, which is used by the compiler to store the many objects internally.

The first proved to be a handicap for using multiplicities, since it required an explicit downcast of an *option* receiver to *bare* whenever a value-typed member was to be accessed on it (see Table 2), a constellation that occurred rather frequently in our case study. If the receiver would have had multiplicity *one* instead, no such cast would have been necessary. However, statically ensuring *one*-ness shares many of the problems of checking non-null annotations (see, e.g., [14]), which is why we left it to future work.

The second was a tribute to earlier reviewers' comments suggesting that in practice, the programmer would want to have some control over the nature of a multitude. However, from a theoretical stance, a non-reified multitude does not have a nature, which hence cannot be controlled; if anything, it appears that when counting many objects, each counts only once (and, as a corollary, iterating over many objects can yield each object only once). Any other

[9] In object-oriented programming, the term *object* is defined as an item that has identity.

property, such as sequence or order, would presuppose a container, suggesting that reification of the multitude (with all its associated problems) is more appropriate (cf. the discussion at the end of Section 2.1). Interestingly, the reverse dilemma exists in the relational database world: originally (i.e., in the pure relational calculus), many entities could not be reified as one entity, and all grouping and ordering had to be done in queries; only with the advent of object-relational extensions, entities could be wrapped in containers, and stored as attributes [6]. However, to our knowledge, these object-oriented extensions have not been embraced in practice, perhaps because with non-reified multitudes available, their reification appears unnecessary.

Concerning the utility of our multiplicities in practical programming, our case study showed that, while incurring no measurable computing overhead, multiplicities offer many opportunities for rewriting object-oriented code into more fluent style, avoiding the use of control structures (see [20] for examples). With hindsight, this should not have been surprising, given that we extended programming languages with a grammatical form that is quite useful in natural language: *the plural*.

7 Research Opportunities

The author's original motivation to conduct this work was to support his conception of object-relational programming, that is, to enhance object-oriented programming with relationships that are *not first-class*, but instead extend the object-oriented notion of a reference as a unidirectional pointer to no or a single object (which is likewise not first-class) to bidirectional to-many pointers.

7.1 Adding Bidirectionality

In object-oriented programming, a standard way of implementing bidirectional relationships using pointers is to identify "opposite" fields that implement the reverse direction of a pointer, a practice which is also adopted by object-relational mappers. For instance, to implement a bidirectional relationship between objects of classes `A` and `B`, the declaration of a field `B b` in class `A` would be annotated to indicate that field `A a` in class `B` implements its reverse direction. Updating field `a` or `b` would then be complemented under the hood with the necessary update of the opposite field. The to-many pointers advocated in this paper can readily be used to implement one half of non-first-class many-to-many relationships using fields; at the same time, extending multiplicities to other declared entities provides for straightforward integration of bidirectional relationships with arbitrary expressions, without having to resort to containers.

7.2 Connection with Roles

While the implementation of relationships using bidirectional to-many pointers is more or less a technical detail (yet one which raises many interesting questions!), the notion of *interfaces as roles* [17] is more central to the author's notion of object-relational programming. In the relational realm, roles are defined as the places of relationships, and every binary relationship comes with two roles, a role and its counter-role. By a suitable definition of role-playing [16], an object playing one role is necessarily related to one or more objects playing the counter-role. The behaviour associated with a role (as expressed by the methods offered by the corresponding interface) is then the behaviour available for the collaboration relying on the defining relationship [18]. Conceptually, an upcast of an object to a role it

plays (an interface it implements) uniquely identifies the objects related to it by playing the counter-role; technically, this can be realized by letting each role define access and update operations for an implicit bidirectional to-many pointer to the objects playing the counter-role. Hence, binary relationships and the collaborations between the participating objects can be specified by pairing interfaces (a role with its counter-role), and by letting classes declare to implement these interfaces (meaning that the classes' instances can play the associated roles). Conceptually simple as this may seem, it raises many research questions, such as specializing relationships via subtyping roles and the relationship of roles to traits (see [21] for a proposal of stateful traits that would lend itself to implementing roles as envisioned here).

7.3 Adding Swarm Behaviour

Section 4 ended with a brief example of overloading methods with different argument multiplicities. The example suggests that, in analogy to type-based dispatch, static method dispatching always selects the method with the most specific argument multiplicities. However, differing from type-based dispatch, we do no dispatch on the receiver's multiplicity (the inherent asymmetry noted above): invoking a method on many receivers is implicitly resolved by invoking it on each.

The notion of *swarm behaviour* put forward in [10] suggests that there is a different interpretation of invoking a method on many receivers: that the receivers are expected to act as a swarm. Multiplicity-based swarm behaviour could be implemented by static methods that have access to a pseudo-variable `these` whose static multiplicity is *any* (or *many*) and which evaluates to the objects the method was invoked on.[10] In fact, one could assume a default implementation of swarm behaviour, which dispatches a method invocation on many objects to the individual objects and which can be overridden when needed (just like the implicit default constructor can be overridden). Using multiplicities, no container would be required to reify the swarm in swarm methods (as in [10]); `these` is simply the plural of `this`.

Pushing the idea of dispatching on multiplicities further, one could even provide methods for multiplicity *none* (i.e., no receiver object), replacing for the Null Object pattern [22]. However, this would require *dynamic dispatching* on the receiver multiplicity.

8 Conclusion

The addition of multiplicity to programming languages as a new grammatical category appears similarly fundamental as the introduction of types. It may prove a better foundation for ridding programming of null pointer exceptions than current type-based approaches (using containers), and can generalize non-null annotations to handling more multiplicities than just *bare* and *one*. At the same time, the dropping of containers promises to solve a number of anomalies that force code for handling many objects to look very different from code for handling one or no object. However, as the author himself experienced during the writing of this paper, there exists a mental obstacle to adopting multiplicities as proposed here, and this is rooted in the entire dismissal of reification: sets in particular have become so fundamental in our thinking that it is difficult to even talk about occurrences of many objects without implicitly or explicitly reifying them into a set. It will be interesting to see whether programming without collections comes more natural, at least in places where the container is irrelevant.

[10] Here it would pay that Java allows static member access on object expressions.

Acknowledgements. The writing of this paper was encouraged by discussions led during Dagstuhl Seminar 14211; the Summit on Advances in Programming Languages was brought to the author's attention at Dagstuhl Seminar 14412.

References

1. Gavin M. Bierman, Erik Meijer, and Wolfram Schulte. The essence of data access in Cω. In Andrew P. Black, editor, *ECOOP 2005 – Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, volume 3586 of *Lecture Notes in Computer Science*, pages 287–311. Springer, 2005.
2. Gavin M. Bierman, Erik Meijer, and Mads Torgersen. Lost in translation: formalizing proposed extensions to C♯. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 479–498. ACM, 2007.
3. Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, Jonathan Eifrig, Scott F. Smith, Valery Trifonov, Gary T. Leavens, and Benjamin C. Pierce. On binary methods. *TAPOS*, 1(3):221–242, 1995.
4. Peter P. Chen. The entity-relationship model – Toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.
5. Umberto Eco. *The Search for the Perfect Language*. Wiley-Blackwell, 1997.
6. Andrew Eisenberg and Jim Melton. SQL: 1999, formerly known as SQL 3. *SIGMOD Record*, 28(1):131–138, 1999.
7. Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.
8. Daniel Jackson. *Software Abstractions – Logic, Language, and Analysis*. MIT Press, 2006.
9. Roman Knöll, Vaidas Gasiunas, and Mira Mezini. Naturalistic types. In Robert Hirschfeld and Eelco Visser, editors, *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2011, part of SPLASH'11, Portland, OR, USA, October 22-27, 2011*, pages 33–48. ACM, 2011.
10. Adrian Kuhn, David Erni, and Marcus Denker. Empowering collections with swarm behavior. *CoRR*, abs/1007.0159, 2010.
11. Erik Meijer, Wolfram Schulte, and Gavin M. Bierman. Unifying tables, objects and documents. In *Proceedings of Declarative Programming in the Context of OO Languages (DP-COOL 2003)*, August 2003.
12. OMG (Object Management Group). Unified Modeling Language 2.4.1. Specification, OMG (Object Management Group), 2011. `http://www.omg.org/spec/UML/2.4.1/`.
13. OMG (Object Management Group). Object Constraint Language 2.4. Specification, OMG (Object Management Group), 1 2012. `http://www.omg.org/spec/OCL/`.
14. Marco Servetto, Julian Mackay, Alex Potanin, and James Noble. The billion-dollar fix – safe modular circular initialisation with placeholders and placeholder types. In Giuseppe Castagna, editor, *ECOOP 2013 – Object-Oriented Programming – 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*, volume 7920 of *Lecture Notes in Computer Science*, pages 205–229. Springer, 2013.
15. Justin Slepak, Olin Shivers, and Panagiotis Manolios. An array-oriented language with static rank polymorphism. In Zhong Shao, editor, *Programming Languages and Systems – 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 27–46. Springer, 2014.

16 Friedrich Steimann. On the representation of roles in object-oriented and conceptual modelling. *Data Knowl. Eng.*, 35(1):83–106, 2000.

17 Friedrich Steimann. Role = interface – A merger of concepts. *Journal of Object-Oriented Programming*, 14:23–32, 2001.

18 Friedrich Steimann. Role + counter role = relationship + collaboration. In Stephen Nelson and Stephanie Balzer, editors, *RAOOL'08: Proceedings of the Workshop on Relationships and Associations in Object-Oriented Languages, co-located with OOPSLA 2008, Nashville, TN, USA*, 2008.

19 Friedrich Steimann. Content over container: Object-oriented programming with multiplicities. In Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld, editors, *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH'13, Indianapolis, IN, USA, October 26-31, 2013*, pages 173–186. ACM, 2013.

20 Friedrich Steimann, Jesper Öqvist, and Görel Hedin. Multitudes of objects: First implementation and case study for Java. *Journal of Object Technology*, 13(5):1: 1–33, 2014.

21 Toon Verwaest, Camillo Bruni, Mircea Lungu, and Oscar Nierstrasz. Flexible object layouts: enabling lightweight language extensions by intercepting slot access. In Cristina Videira Lopes and Kathleen Fisher, editors, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22–27, 2011*, pages 959–972. ACM, 2011.

22 Bobby Woolf. Null object. In Robert C. Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Program Design 3*, pages 5–18. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.