First-class Relationships in an Object-oriented Language

Gavin Bierman[1] and Alisdair Wren[2]

[1] Microsoft Research, Cambridge
`gmb@microsoft.com`
[2] University of Cambridge Computer Laboratory
`Alisdair.Wren@cl.cam.ac.uk`

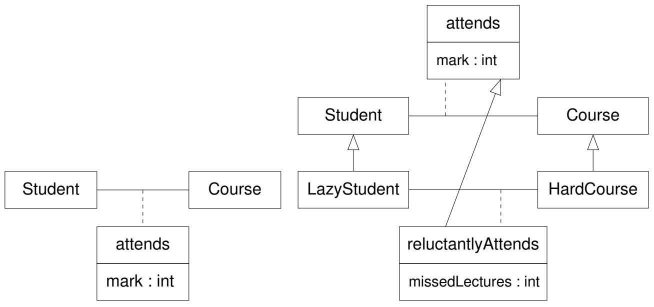
Abstract. In this paper we investigate the addition of first-class relationships to a prototypical object-oriented programming language (a "middleweight" fragment of Java). We provide language-level constructs to declare relationships between classes and to manipulate relationship instances. We allow relationships to have attributes and provide a novel notion of relationship inheritance. We formalize our language giving both the type system and operational semantics and prove certain key safety properties.

1 Introduction

Object-oriented programming languages, and object modelling techniques more generally, provide software engineers with useful abstractions to create large software systems. The grouping of objects into classes and those classes into hierarchies provides the software engineer with an extremely flexible way of representing real-world semantic notions directly in code.

However, whilst object-oriented languages easily represent real-world entities (e.g. students, lectures, buildings), the programmer is poorly served when trying to represent the many natural *relationships* between those entities (e.g. 'attends lecture', 'is taught in').

Relationships clearly can be represented in object-oriented languages—indeed patterns have been established for the purpose [10]—but this important abstraction can get lost in the implementation that is forced upon the programmer by the lack of first-class support. Different aspects of the relationship can be implemented by fields and methods of the participating classes, but this distributes information about the relationship across various classes. Alternatively, small classes can be defined to contain references to the two related objects along with any attributes of the relationship. In both cases, without great care the structure can become internally inconsistent, especially in the presence of aliasing. Furthermore, we argue that the application of standard class-based inheritance to these 'relationship classes' does not adequately capture the intuitive semantics of relationship inheritance, which must otherwise be encoded in standard Java. Such an encoding can only lead to further complexity and more opportunities for inconsistency.



(a) Association Class (b) Parallel Hierarchy

Fig. 1. Relationships represented as UML *association classes*

The importance of relationships is clearly reflected by their prominence in almost all modelling languages: from (Extended) Entity-Relationship Diagrams (ER-diagrams) [5] to Unified Modelling Language (UML) [9]. In Figure 1 we give some examples of relationships expressed in UML (we use these as running examples throughout this paper).

We argue that such important abstractions deserve first-class support from programming languages. We are the not the first to do so; Rumbaugh also pointed out the importance of first-class language support for relationships [13]. Noble and Grundy also proposed that relationships should persist from the modelling to the implementation stage of program development [11]. Albano et al. propose a similar extension to a language for managing object-oriented databases (OODB) [1], but do so in a much richer data model and do not give a full description of their language.

In contrast to these works, our approach is more formal. We believe that such a formal, mathematical approach is essential to set a firm foundation for researchers, users and implementors of advanced programming languages. To that end, our main contribution is a precise description of how Java (or any other class-based, strongly-typed, object-oriented language) can be extended to support first-class relationships. Our tool is a small core language, RelJ, which is a subset of Java (much like Middleweight Java [4]) with suitable extensions for the support of relationships. RelJ provides means to define relationships between objects, to specify attributes associated with those relationships, and to create hierarchies of relationships. RelJ is intended to capture the essence of these extensions to Java, yet is small enough to formalize completely. Other features could be added to RelJ to make it a more complete language, but these would not impact on the extensions for relationships.

The remainder of the paper is organized as follows. In Sect. 2 we introduce our calculus and give a grammar. The type system of RelJ is defined in Sect. 3, where the formal notion of subtyping is discussed and well-typed RelJ programs are characterized. Section 4 gives the dynamics of RelJ with a small-step operational semantics. We outline a proof of type soundness for RelJ in Sect. 5. Section 6 describes an extension to RelJ which allows the addition of UML-style multiplicity restrictions to relationships. Finally, in Sect. 7, we conclude and consider further and related work.

2 The RelJ Calculus

As mentioned earlier, the core of RelJ is a subset of Java, similar to other fragments of Java-like languages [4, 7, 8]. The fragment we use consists of simple class declarations that contain a number of field declarations and method declarations. The exact form of the class declarations will be made more precise later.

2.1 Relationship Model

The main feature of RelJ is its support for first-class relationships. In addition to class declarations, therefore, a RelJ program consists of a number of relationship declarations, which are written:

$$\texttt{relationship}\ r\ \texttt{extends}\ r'\ \texttt{(}n\texttt{,}\ n'\texttt{)}\ \{\ \mathsf{FieldDecl}^*\ \mathsf{MethDecl}^*\ \}$$

This defines a relationship, r, with a number of type/field name pairs, $\mathsf{FieldDecl}^*$ and method declarations, $\mathsf{MethDecl}^*$. The relationship is between n and n' where n, n' range over classes *and* relationships. This provides a means for relationship instances to participate in further relationships. This feature is known as *aggregation* in ER-modelling [14]. An example is shown in Fig. 2: the `Recommends` relationship specifies that a `Tutor` may recommend a `Student` to attend a particular `Course` by relating an instance of `Tutor` to an instance of `Attends`, the relationship that specifies which students attend which courses. Relationships are directed (one-way) and many-to-many—more on this in Sect. 6.

We relate two objects, o_1 and o_2, with a relationship, r, by creating an instance of r, which then exists *between* o_1 and o_2, and stores the values for r's fields. Relationship instances are first-class runtime objects in RelJ and so can, for example, be stored in variables and fields. This immediately introduces design issues relating to the removal of relationship instances and consequent creation (or not) of dangling pointers: these are discussed later.

We also support relationship inheritance, which is denoted idiomatically in UML as inheritance between association classes (Fig. 1b). To the best of our knowledge, our support for this inheritance is novel and, as we will detail later, is significantly different from the standard class-based inheritance model.

2.2 Class Inheritance vs Relationship Inheritance

While class inheritance in RelJ is identical to that in Java, RelJ's relationship inheritance is based on a restricted form of delegation, as found in languages

```
class Student {
   String name;
}
class LazyStudent extends Student {
   int    hoursOfSleep;
}
class Course {
   String title;
}
class Tutor {
   String name;
}
relationship Attends (Student, Course) {
   int mark;
}
relationship ReluctantlyAttends extends Attends
                                (LazyStudent, Course) {
   int missedLectures;
}
relationship CompulsorilyAttends extends Attends
                                 (Student, Course) {
   String reason;
}
relationship Recommends (Tutor, Attends) {
   String reason;
}
...
alice = new LazyStudent();
programming = new Course();
typeSystems = new Course();
Attends.add(alice, programming);              // Alice attends Programming
ReluctantlyAttends.add(alice, typeSystems);
                                // Alice reluctantly attends Type Systems
for (Course c : alice.Attends) {
  print "Attends: " + c.title;
};                                         // Prints:
                                           //    Attends: Programming
                                           //    Attends: Type Systems
```

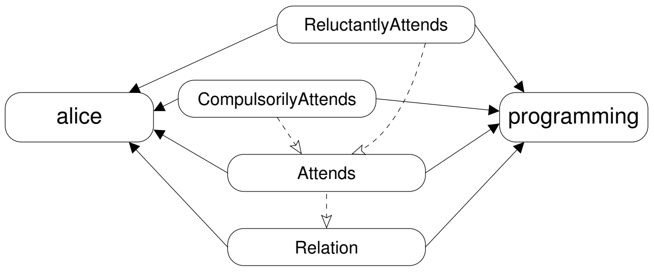


Fig. 2. Example RelJ code and possible instantiation

such as Self [16] and, more recently, δ [2]. Consider the RelJ code for a simple example, adapted from Pooley and Stevens [15], which is shown in Fig. 2.

When `alice` and `programming` are placed in the `Attends` relationship, an instance of `Attends` is created between those objects. Subsequently, when `alice` and `programming` are further placed in `ReluctantlyAttends`, an instance of `ReluctantlyAttends` is created between `alice` and `programming`, but contains *only* the `missedLectures` field. If that `ReluctantlyAttends` instance receives a field look-up request for `mark`, it passes—*delegates*—the request to the `Attends` instance—the *super-instance*—that exists between those same objects.

To ensure all instances are 'complete', specifically that they have all the fields one would expect by inheritance, we impose the following invariant:

Invariant 1 *Consider a relationship r_2 which* `extends` *r_1. For every instance of relationship r_2 between objects o_1 and o_2, there is an instance of r_1, also between o_1 and o_2, to which it delegates requests for r_1's fields.*

By this invariant, if `alice` and `programming` were placed in the `ReluctantlyAttends` relationship without first having been placed in the `Attends` relationship, then an `Attends` instance would be implicitly created between them.

Invariant 2 *For every relationship r and pair of objects o_1 and o_2, there is at most one instance of r between o_1 and o_2.*

According to this second invariant, if `alice` and `programming` were later placed in the `CompulsorilyAttends` relationship, then its instance and that of `ReluctantlyAttends` would share a common super-instance: the `Attends` instance between `alice` and `programming`. This situation is shown at the bottom of Fig. 2, with the dotted lines indicating delegation of field lookups.

The motivation for such a mechanism is based on what one might intuitively expect from relationships: Clearly, if Alice reluctantly attends a course, then she also attends it and will receive a mark, thus we require sub-relationships to be included in their super-relationship, giving rise to Invariant 1. Also, if Alice is both compulsorily and reluctantly attending some course, the mark will be the same regardless of whether one views her attendance as reluctant, compulsory or without any annotation. Thus, for each pair of related objects, there should be only one instance of each relationship so that relationship properties are consistent, hence Invariant 2.

RelJ also allows the removal of relationship instances. For example, we could extend the code of Fig. 2 to remove the fact that `Alice` attends `programming`:

```
...
Attends.rem(alice, programming);  // Remove Alice attends Programming
for (Course c : alice.Attends){
  print "Attends: " + c.title;    // Prints:
}                                 //   Attends: Type Systems
```

In fact, both the relationship addition and removal operations are *statement expressions*. When used as an expression, `add` returns the relationship instance that was created: this provides a convenient short-cut for setting the new instance's fields. For regularity, `rem` returns the instance that was removed, or `null` if the relationship did not exist before the attempted removal.

We return now to the issue raised earlier concerning relationship instance removal. Consider the following code:

```
bob = new Student();
bob.name = "Bob";
databases = new Course();
databases.title = "DB 101";

bobdb = Attends.add(bob, databases);    // Add bob to databases
bobdb.mark = 99;
for (Course cs : bob.Attends) {
  print cs.title;
};                                      // Prints DB 101

print bobdb.mark;                       // Prints 99

Attends.rem(bob, databases);            // Remove bob from databases

for (Course cs : bob.Attends) {
  print cs.title;
};                                      // Prints nothing
```

The second iteration shows that the relationship between `bob` and `databases` has been correctly removed. We must then choose the fate of the reference to the `Attends`-instance stored in `bobdb`: what happens if we append the statement `print bobdb.mark;`?

There are clearly a number of options: either the instance is removed, in which case we would expect a runtime error; or the runtime maintains some liveness information so that an access to the variable `bobdb` would generate a specific relationship exception; or finally, we could choose not to remove the relationship instance at all, in which case the code would print `99`. We have chosen the third option. Thus, in RelJ, the relationship instance itself is not removed upon deletion, but rather is treated like any other runtime value and is removed by garbage collection. More experience in relationship programming is needed before we can determine if this is the correct design decision.

2.3 Language Definition

We give the grammar for RelJ programs and types in Fig. 3.

The Java types used in RelJ are class names and a single primitive type, `boolean` (the inclusion of further primitive types does not impact on the formalization). As discussed, we provide relationship names as types. To allow relationship processing RelJ has a (generic) set type `set<`n`>`, that denotes a set of values of type n. This set type is not a *reference* type, but is a *primitive*

$$
\begin{array}{rcll}
p \in \mathsf{Program} & ::= & \mathsf{ClassDecl}^* \ \mathsf{RelDecl}^* & \\
\mathsf{ClassDecl} & ::= & \texttt{class}\ c\ \texttt{extends}\ c' & \\
 & & \quad \{\ \mathsf{FieldDecl}^*\ \mathsf{MethDecl}^*\ \} & \\
\mathsf{RelDecl} & ::= & \texttt{relationship}\ r\ \texttt{extends}\ r'\ \texttt{(}n\texttt{,}\ n'\texttt{)} & \\
 & & \quad \{\ \mathsf{FieldDecl}^*\ \mathsf{MethDecl}^*\ \} & \\
n \in \mathsf{NominalType} & ::= & c \mid r & \\
t \in \mathsf{Type} & ::= & \texttt{boolean} \mid n \mid \texttt{set<}n\texttt{>} & \\
\mathsf{FieldDecl} & ::= & t\ f\texttt{;} & \\
\mathsf{MethDecl} & ::= & t\ m\texttt{(}t'\ x\texttt{)}\ mb & \\
mb \in \mathsf{MethBody} & ::= & \{\ s\ \texttt{return}\ e\texttt{;}\ \} & \\
v \in \mathsf{Value} & ::= & \texttt{true} \mid \texttt{false} \mid \texttt{null} \mid \texttt{empty} & \\
l \in \mathsf{LValue} & ::= & x \mid & \\
 & & e\texttt{.}f & \text{field access} \\
e \in \mathsf{Expression} & ::= & v \mid & \text{value} \\
 & & l \mid & \text{l-value} \\
 & & e_1\ \texttt{==}\ e_2 \mid & \text{equality test} \\
 & & e_1\ \texttt{+}\ e_2 \mid e_1\ \texttt{-}\ e_2 \mid & \text{set addition/removal} \\
 & & e\texttt{.}r \mid e\texttt{:}r \mid & \text{relationship access} \\
 & & e\texttt{.from} \mid & \text{relationship source} \\
 & & e\texttt{.to} \mid & \text{relationship destination} \\
 & & se & \text{statement expression} \\
se \in \mathsf{StatementExp} & ::= & \texttt{new}\ c\texttt{()} \mid & \text{instantiation} \\
 & & l\ \texttt{=}\ e \mid & \text{assignment} \\
 & & r\texttt{.add(}e\texttt{,}e'\texttt{)} \mid r\texttt{.rem(}e\texttt{,}e'\texttt{)} \mid & \text{relationship addition/removal} \\
 & & e\texttt{.}m\texttt{(}e'\texttt{)} & \text{method call} \\
s \in \mathsf{Statement} & ::= & \epsilon \mid & \text{empty statement} \\
 & & se\texttt{;}\ s_1 \mid & \text{expression} \\
 & & \texttt{if (}e\texttt{)}\ \{s_1\}\ \texttt{else}\ \{s_2\}\texttt{;}\ s_3 \mid & \text{conditional} \\
 & & \texttt{for (}n\ x\ \texttt{:}\ e\texttt{)}\ \{s_1\}\texttt{;}\ s_2 & \text{set iteration}
\end{array}
$$

Fig. 3. The grammar of RelJ types and programs

(value) type, much like the generic literal types used by the ODMG [12].[3] RelJ does not support nested sets—sets of sets are not permitted. RelJ offers a `for` iterator over set values (we adopt the same syntax as Java 5.0 for iterating over collections). We also provide operators for explicitly adding an element to a set (`+`), and for removing an element (`-`).

For simplicity, we require some regularity in the class (and relationship) declarations of RelJ programs: (1) we insist that all class declarations include the supertype; (2) we write out the receiver of field access or method invocation in full; (3) all methods take just one argument; (4) all method declarations end with a `return` statement; and (5) we assume that in a RelJ program exactly one class supports a `main` method. To be concise, we do not consider constructor methods; field initialization, other than the provision of type-appropriate initial values, is performed explicitly.

The metavariable c ranges over the set of class names, ClassName; r ranges over the set of relationship names, RelName; n ranges over both ClassName and RelName; f ranges over the set of field names, FldName; m ranges over the set of method names, MethName; and x ranges over the set of variable names, VarName, which we assume contains the element `this`, which cannot be on the left-hand side of an assignment. Metavariables may not take the undefined value.

As usual for such language formalizations, we assume that given a RelJ program, P, the class and relationship declarations give rise to class and relationship tables that are denoted by $\mathcal{C}_P$ and $\mathcal{R}_P$, respectively [6]. (We will drop the subscript when it is unambiguous.) A class (relationship) table is then a map from a class (relationship) name to a class (relationship) definition. Signatures for these maps are to be found in Fig. 4.

$$
\begin{aligned}
\mathcal{C} \in \mathsf{ClassTable} &: \mathsf{ClassName} \rightarrow \mathsf{ClassName} \times \mathsf{FieldMap} \times \mathsf{MethMap} \\
\mathcal{R} \in \mathsf{RelTable} &: \mathsf{RelName} \rightarrow \mathsf{RelName} \times \mathsf{NominalType} \times \mathsf{NominalType} \times \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{FieldMap} \times \mathsf{MethMap} \\
\mathcal{F} \in \mathsf{FieldMap} &: \mathsf{FldName} \rightarrow \mathsf{Type} \\
\mathcal{M} \in \mathsf{MethMap} &: \mathsf{MethName} \rightarrow \mathsf{VarName} \times \mathsf{LocalMap} \times \mathsf{Type} \times \mathsf{Type} \times \mathsf{MethBody} \\
\mathcal{L} \in \mathsf{LocalMap} &: \mathsf{VarName} \rightarrow \mathsf{Type}
\end{aligned}
$$

Fig. 4. Signatures of class and relationship tables

A class definition is a tuple, $(c, \mathcal{F}, \mathcal{M})$, where c is the superclass; $\mathcal{F}$ is a map from field names to field types; and $\mathcal{M}$ is a map from method names to method definitions. Method definitions are tuples $(x, \mathcal{L}, t_1, t_2, mb)$ where x is the parameter; $\mathcal{L}$ is a map from local variable names to their types; t_1 is the parameter type; t_2 is the return type; and mb is the method body. For brevity, we write $\mathcal{F}_c$ and $\mathcal{M}_c$ for the field and method definition maps of class c.

Relationship definitions are tuples $(r', n, n', \mathcal{F}, \mathcal{M})$ where r' is the super-relationship; n and n' are the types between which the relationship is formed

[3] Having sets as a generic value type allows us to soundly support covariance—this is discussed in more detail in Sect. 3.

(the *source* and *destination* respectively); and $\mathcal{F}$, $\mathcal{M}$ are the field map and method map respectively, as found in class definitions. As for classes, we write $\mathcal{F}_r$ for r's field definition map and $\mathcal{M}_r$ for r's method map.

In summary, RelJ offers the following operations to manipulate relationships: $e\texttt{.}r$ finds the objects related to the result of e through relationship r; $e\texttt{:}r$ finds the instances of r that exist between the result of e and the objects to which it is related; and the pseudo-fields `from` and `to` are made available on relationship instances, and return the source and destination objects between which the instance exists (or existed). These are further described in the following sections.

3 Type System

We provide `Object` for the root of the class hierarchy as usual, and `Relation` as its counterpart in the relationship hierarchy, and assume appropriate entries in $\mathcal{C}$ and $\mathcal{R}$ respectively. We define the usual subtyping relation $P \vdash t \leq t'$ where t is a subtype of t', directly populated with the information about immediate super-types provided by $\mathcal{C}$ and $\mathcal{R}$, then closed under transitivity and reflexivity. P is omitted where the context makes it unambiguous.

We leave the less important typing rules to Appendix A, but two rules worth particular note are shown here:

$$\text{(STCov)}\ \frac{\vdash n_1 \leq n_2}{\vdash \texttt{set<}n_1\texttt{>} \leq \texttt{set<}n_2\texttt{>}} \qquad \text{(STObject)}\ \frac{}{\vdash \texttt{Relation} \leq \texttt{Object}}$$

STCov makes set types covariant with their contained type. If $\texttt{set<}-\texttt{>}$ were a reference type, then this kind of covariance would be unsound. However, $\texttt{set<}-\texttt{>}$ is a value type, thus such values are not referenced or mutated, only copied.

To unify the relationship and class hierarchies—desirable in the absence of generics—we take `Relation` as a subtype of `Object` in rule STObject.[4]

While $\mathcal{F}_c$ and $\mathcal{M}_c$ give us the fields and methods declared directly in c, we define $\mathcal{FD}_c$ and $\mathcal{MD}_c$ to provide us with all the fields and methods available for c's instances, including those inherited from its superclasses, so that their types might be checked in the later type rules:

$$\mathcal{FD}_c(f) = \begin{cases} \mathcal{F}_c(f) & \text{if } f \in \mathsf{dom}(\mathcal{F}_{P,c}) \text{ or } c = \texttt{Object} \\ \mathcal{FD}_{c'}(f) & \text{if } f \notin \mathsf{dom}(\mathcal{F}_{P,c}) \text{ and } \mathcal{C}(c) = (c', \_, \_) \end{cases}$$

$\mathcal{MD}$ is defined similarly for class methods, as are $\mathcal{FD}$ and $\mathcal{MD}$ for relationships.

We type expressions and statements in the presence of a typing environment, Γ, which assigns types to variable names. Selected typing judgements for RelJ expressions are given below:

$$\text{(TSRelObj)}\ \frac{\Gamma \vdash e\colon n_1 \quad \mathcal{R}(r) = (\_, n_2, n_3, \_, \_) \quad \vdash n_1 \leq n_2}{\Gamma \vdash e\texttt{.}r\colon \texttt{set<}n_3\texttt{>}} \qquad \text{(TSRelInst)}\ \frac{\Gamma \vdash e\colon n_1 \quad \mathcal{R}(r) = (\_, n_2, \_, \_, \_) \quad \vdash n_1 \leq n_2}{\Gamma \vdash e\texttt{:}r\colon \texttt{set<}r\texttt{>}}$$

[4] If we added generics to RelJ it would be possible to remove this typing rule.

TSRelObj types the lookup of objects related through r to the result of e. As our relationships are implicitly many-to-many, the result of this lookup is a set of r's destination type, n_3. The relationship instances that sit between the result of e and the result of e.r are accessed through e:r. The result of such a lookup is a set of r-instances, as specified in TSRelInst. There is a bias here between the source and destination of a relationship: the relationship instances may only be accessed from the source object. It is not difficult to extend the language so that access from the destination objects is also possible.

$$\text{(TSFrom)}\ \frac{\Gamma \vdash e\colon r \quad \mathcal{R}(r) = (\_, n, \_, \_, \_)}{\Gamma \vdash e\texttt{.from}\colon n} \qquad \text{(TSTo)}\ \frac{\Gamma \vdash e\colon r \quad \mathcal{R}(r) = (\_, \_, n, \_, \_)}{\Gamma \vdash e\texttt{.to}\colon n}$$

Given an r-instance, the objects between which it exists (or between which it once existed) can be accessed with the `from` and `to` properties. TSFrom and TSTo assign types according to the relationship's declaration—therefore, these are typed covariantly with the relationship type, but this is sound as they are immutable for all instances of such a relationship.

$$\text{(TSRelAdd)}\ \frac{\mathcal{R}(r) = (\_, n_1, n_2, \_, \_) \quad \Gamma \vdash e_1\colon n_3 \quad \Gamma \vdash e_2\colon n_4 \quad \vdash n_3 \leq n_1 \quad \vdash n_4 \leq n_2}{\Gamma \vdash r\texttt{.add(}e_1\texttt{,}e_2\texttt{)}\colon r} \qquad \text{(TSRelRem)}\ \frac{\mathcal{R}(r) = (\_, n_1, n_2, \_, \_) \quad \Gamma \vdash e_1\colon n_3 \quad \Gamma \vdash e_2\colon n_4 \quad \vdash n_3 \leq n_1 \quad \vdash n_4 \leq n_2}{\Gamma \vdash r\texttt{.rem(}e_1\texttt{,}e_2\texttt{)}\colon r}$$

Finally, TSRelAdd and TSRelRem specify typing of the operators that relate and unrelate objects. In both cases, e_1 and e_2 must be of the source and destination type, respectively, of relationship r. The result of either operation will be an instance of r; that which was created or removed. A removal may evaluate to `null` where the results of e_1 and e_2 were unrelated by r.

The type-checking relation for statements is of the form $\Gamma \vdash s$, the rules for which are largely routine. We show some examples, however:

$$\text{(TSExp)}\ \frac{\Gamma \vdash se\colon t \quad \Gamma \vdash s}{\Gamma \vdash se\texttt{;}\ s} \qquad \text{(TSFor)}\ \frac{\Gamma \vdash e\colon \texttt{set<}n_1\texttt{>} \quad \Gamma[x \mapsto n_2] \vdash s_1 \quad \vdash n_1 \leq n_2 \quad \Gamma \vdash s_2}{\Gamma \vdash \texttt{for (}n_2\ x\ \texttt{:}\ e\texttt{) \{}s_1\texttt{\};}\ s_2}\ x \notin \mathsf{dom}(\Gamma)$$

TSExp allows type-correct statement expressions to be used as statements, while TSFor checks that the `for` construct is only asked to iterate over a set of object references. Note that, to be consistent with the Java 5.0 syntax, we require an explicit type for the iterating variable, although there is no reason why this type could not be inferred. We also require that the iteration variable is not already in scope.

The set $\mathsf{validTypes}_P$ specifies the types that may be assigned to fields and variables:

$$\mathsf{validTypes}_P = \{\texttt{boolean}\} \cup \mathsf{dom}(\mathcal{C}_P) \cup \mathsf{dom}(\mathcal{R}_P) \cup \{\texttt{set<}n\texttt{>} \mid n \in \mathsf{dom}(\mathcal{C}_P) \cup \mathsf{dom}(\mathcal{R}_P)\}$$

In the following two rules, we check fields and methods in the presence of their enclosing class or relationship:

$$\frac{\begin{array}{ll} & \mathcal{C}(n) = (n', \_, \_) \ \vee\ \mathcal{R}(n) = (n', \_, \_, \_, \_) \\ ^{1.} & f \notin \mathsf{dom}(\mathcal{FD}_{n'}) \\ ^{2.} & \mathcal{F}_n(f) \in \mathsf{validTypes}_P \\ ^{3.} & \mathcal{R}(f) = (\_, n_1, n_2, \_) \Rightarrow\ \not\vdash n \leq n_1 \end{array}}{P, n \vdash f} \ \text{(TSField)}$$

TSField checks that f is a good field for class or relationship n by verifying (1) that f is not defined in any super-type of n; (2) that f's type is valid in a well-typed program and (3) that there is no relationship with the same name as f that might make references to f ambiguous.

$$\frac{\begin{array}{ll} & \mathcal{C}_P(n) = (n', \_, \mathcal{M}_n) \vee \mathcal{R}_P(n) = (n', \_, \_, \_, \mathcal{M}_n) \\ & \mathcal{M}_n(m) = (x, \mathcal{L}, t_1, t_2, \{\ s\ \mathtt{return}\ e;\ \}) \\ ^{1.} & t_1 \in \mathsf{validTypes}_P \\ ^{2.} & \mathtt{this}, x \notin \mathsf{dom}(\mathcal{L}) \\ ^{3.} & \{x \mapsto t_1, \mathtt{this} \mapsto n\} \cup \mathcal{L} \vdash s \\ ^{4.} & \{x \mapsto t_1, \mathtt{this} \mapsto n\} \cup \mathcal{L} \vdash e\colon t_2' \\ ^{5.} & \vdash t_2' \leq t_2 \\ ^{6.} & \mathcal{MD}_{n'}(m) = (\_, \_, t_3, t_4, \_) \Rightarrow\ \vdash t_3 \leq t_1\ \wedge\ \vdash t_2 \leq t_4 \end{array}}{P, n \vdash m} \ \text{(TSMethod)}$$

TSMethod checks (1) that the input type of method m in class/relationship n is valid; (2) that the parameter name and `this` do not clash with any local variables; (3) that the method body is well-typed when the parameter, `this` and the local variables are assigned the types specified in the class' method table; (4, 5) that the `return` expression has a subtype of the method's declared return type; and (6) that the input type of this method is a supertype of any previous declaration of m in a super-type of c, and that the return type of m is a subtype of any previous method declaration: that is, that this definition of m may be used anywhere a supertype's version of m can be used. We then specify the validity of classes and relationships:

$$\frac{\begin{array}{c} \mathcal{C}(c) = (c' \neq c, \mathcal{F}, \mathcal{M}) \\ P \vdash c' \\ \forall f \in \mathsf{dom}(\mathcal{F}) : P, c \vdash f \\ \forall m \in \mathsf{dom}(\mathcal{M}) : P, c \vdash m \end{array}}{P \vdash c} \ \text{(TSClass)}$$

$$\frac{\begin{array}{ll} & \mathcal{R}_P(r) = (r' \neq r, n_1, n_2, \mathcal{F}, \mathcal{M}) \\ & r' \in \mathsf{validTypes}_P \\ ^{1.} & \mathcal{R}_P(r') = (\_, n_1', n_2', \_, \_) \\ ^{2.} & \vdash n_1 \leq n_1' \\ ^{3.} & \vdash n_2 \leq n_2' \\ & \forall f \in \mathsf{dom}(\mathcal{F}) : P, r \vdash f \\ & \forall m \in \mathsf{dom}(\mathcal{M}) : P, r \vdash m \end{array}}{P \vdash r} \ \text{(TSRelationship)}$$

TSClass specifies that a class type is well-formed if its superclass is well-formed, and if all of its methods and fields are well-typed. TSRelationship imposes many of the same restrictions as TSClass, with the addition of conditions 1–3, which check the types related by r's super-relationship are supertypes of those that r relates.

4 Semantics

We specify evaluation rules for a small-step semantics. We use evaluation contexts to specify evaluation order [17], and use variable renaming to avoid the need for an explicit frame stack [7].

The meta-variables used in the semantics range over addresses, values, errors, objects and stores as follows:

$$\begin{aligned}
\iota &\in \mathsf{Address} \\
\iota^{\texttt{null}} &\in \mathsf{Address} \cup \{\texttt{null}\} \\
u &\in \mathsf{DynValue} = \{\texttt{null}, \texttt{true}, \texttt{false}\} \cup \mathsf{Address} \cup \mathcal{P}(\mathsf{Address}) \\
w &\in \mathsf{Error} ::= \mathsf{NullPtrError} \mid \mathcal{E}_{\mathrm{e}}[w] \mid \mathcal{E}_{\mathrm{s}}[w] \mid \{\ w\ \texttt{return}\ e;\ \} \\
o &\in \mathsf{Object} \\
\sigma &: \mathsf{Address} \rightarrow \mathsf{Object} \\
\rho &: (\mathsf{Address} \times \mathsf{Address} \times \mathsf{RelName}) \rightarrow \mathsf{Address} \\
\lambda &: \mathsf{VarName} \rightarrow \mathsf{DynValue}
\end{aligned}$$

Objects, ranged over by o, are either class instances or relationship instances. We write class instances as an annotated pair, $\langle\!\langle c \| f_1 : v_1, \ldots, f_i : v_i \rangle\!\rangle$, containing a mapping from field names to values, and the object's dynamic type, c. Relationship instances are written as an annotated 5-tuple, $\langle\!\langle r, \iota^{\texttt{null}}, \iota_1, \iota_2 \| f_1 : v_1, \ldots, f_i : v_i \rangle\!\rangle$, containing the familiar field value map and dynamic type, as well as the object addresses the instance relates, ι_1 and ι_2, and a reference to the relationship instance's *super-instance*, $\iota^{\texttt{null}}$; specifically, the instance of r's super-relationship which relates the same object addresses ι_1 and ι_2. Where $r = \texttt{Relation}$, there is no super-relationship and this reference is `null`. For both types of object, we take $o(f)$ and $\mathsf{dom}(o)$ as if they were applied to o's field value map.

Dynamic values (as opposed to syntactic value literals), ranged over by u, are either addresses, ranged over by ι, sets of addresses, or `true`, `false` or `null`. A small-step semantics means that expressions may at times be only partially evaluated, so we include these run-time values and partially-evaluated method bodies in language expressions by extending $\mathsf{Expression}$ as follows:

$e \in \mathsf{DynExpression} ::=$	
$u \mid$	dynamic values
$mb \mid$	method body
$\ldots$	terms from $\mathsf{Expression}$ grammar

$\mathsf{DynLValue}$ and $\mathsf{DynStatement}$ are generated from LValue and $\mathsf{Statement}$ in the obvious way, and e, l and s will range over these new definitions from this point onward.

A store, σ, is a map from addresses to objects, while local variables are given values by a locals store, λ. A relationship store, ρ maps relationship tuples to addresses such that $\rho(r, \iota_1, \iota_2)$ indicates the address of the instance of r which exists between ι_1 and ι_2.

$\mathcal{E}_e \in \mathsf{ExpContext} ::=$	
$\bullet$	hole
$\mid \mathcal{E}_e\texttt{.}f$	field lookup
$\mid \mathcal{E}_e \texttt{ == } e \mid u \texttt{ == } \mathcal{E}_e$	equality test
$\mid \mathcal{E}_e \texttt{ + } e \mid u \texttt{ + } \mathcal{E}_e$	set addition
$\mid \mathcal{E}_e \texttt{ - } e \mid u \texttt{ - } \mathcal{E}_e$	set removal
$\mid \mathcal{E}_e\texttt{.}r \mid \mathcal{E}_e\texttt{:}r$	relationship access
$\mid \mathcal{E}_e\texttt{.from} \mid \mathcal{E}_e\texttt{.to}$	relationship from/to
$\mid \texttt{\{ } \mathcal{E}\ \texttt{return } e\texttt{; \}} \mid \texttt{\{ return } \mathcal{E}_e\texttt{; \}}$	method body
$\mid \mathcal{E}_e\texttt{.}f \texttt{ = } e \mid x \texttt{ = } \mathcal{E}_e \mid u\texttt{.}f \texttt{ = } \mathcal{E}_e$	assignment
$\mid \mathcal{E}_e\texttt{.}m\texttt{(}e'\texttt{)} \mid u\texttt{.}m\texttt{(}\mathcal{E}_e\texttt{)}$	method call
$\mid r\texttt{.add(}\mathcal{E}_e\texttt{,}e'\texttt{)} \mid r\texttt{.add(}u\texttt{,}\mathcal{E}_e\texttt{)}$	relationship addition
$\mid r\texttt{.rem(}\mathcal{E}_e\texttt{,}e'\texttt{)} \mid r\texttt{.rem(}u\texttt{,}\mathcal{E}_e\texttt{)}$	relationship removal
$\mathcal{E}_s \in \mathsf{StatContext} ::=$	
$\mathcal{E}_e\texttt{; } s$	expression
$\mid \texttt{for (}n\ x \texttt{ : } \mathcal{E}_e\texttt{) \{}s_1\texttt{\}; } s_2$	set iteration
$\mid \texttt{if (}\mathcal{E}_e\texttt{) \{}s_1\texttt{\} else \{}s_2\texttt{\}; } s_3$	conditional

Fig. 5. Grammar for evaluation contexts

During execution, the store and its constituent objects are modified by updating the relevant map. Update of some map f is written $f[a \mapsto b]$ such that $f[a \mapsto b](a) = b$ and $f[a \mapsto b](c) = f(a)$ where $a \neq c$. Such substitutions are commonly applied to stores ($\sigma[\iota \mapsto o]$) and to objects ($o[f \mapsto v]$).

Substitution of variables in program syntax uses the standard notation, $e[x'/x]$, for the replacement of all variables x in e with x', and similarly with statements, $s[x'/x]$.

Figure 5 gives the evaluation contexts for RelJ expressions and statements. All contexts $\mathcal{E}$ contain a hole, denoted $\bullet$, which indicates the position of the sub-expression to be evaluated first—in this case the left-most, inner-most. An expression may be placed in a context's hole position by substitution, denoted $\mathcal{E}_e[e]$. Notice that we no longer distinguish between those expressions that may or may not be used in statement position.

A *configuration* in the semantics is a 5-tuple of typing environment, heap, relationship store, locals map, and a statement: $\langle \Gamma, \sigma, \rho, \lambda, s\rangle$. An *error configuration* is a configuration $\langle \Gamma, \sigma, \rho, \lambda, w\rangle$, with an error in place of a statement. Γ is included for the proof of type soundness.

Expression execution proceeds when a sub-expression in hole position may be reduced, as specified by OSContextE. We elide the similar rule for expressions in statement context:

$$(\text{OSContextE})\ \frac{\langle \Gamma, \sigma, \rho, \lambda, e\rangle \overset{P}{\rightsquigarrow} \langle \Gamma', \sigma', \rho', \lambda', e'\rangle}{\langle \Gamma, \sigma, \rho, \lambda, \mathcal{E}_e[e]\rangle \overset{P}{\rightsquigarrow} \langle \Gamma', \sigma', \rho', \lambda', \mathcal{E}_e[e']\rangle}$$

We also execute statements inside partially-executed method bodies:

$$(\text{OSInBody})\ \frac{\langle \Gamma, \sigma, \rho, \lambda, s\rangle \overset{P}{\rightsquigarrow} \langle \Gamma', \sigma', \rho', \lambda', s'\rangle}{\langle \Gamma, \sigma, \rho, \lambda, \texttt{\{ } s \texttt{ return } e\texttt{; \}}\rangle \overset{P}{\rightsquigarrow} \langle \Gamma', \sigma', \rho', \lambda', \texttt{\{ } s' \texttt{ return } e\texttt{; \}}\rangle}$$

$$\mathsf{newPart}_P(r, \iota^{\texttt{null}}, \iota_1, \iota_2) = \langle\!\langle r, \iota^{\texttt{null}}, \iota_1, \iota_2 \| f_1 : \mathsf{initial}_P(\mathcal{F}_{P,r}(f_1)), \ldots, f_i : \mathsf{initial}_P(\mathcal{F}_{P,r}(f_i))\rangle\!\rangle$$
$$\text{where } \{f_1, f_2, \ldots, f_i\} = \mathsf{dom}(\mathcal{F}_{P,r})$$

$$\mathsf{addRel}_P(r, \iota_1, \iota_2, \sigma_1, \rho_1) = \begin{cases} (\sigma_1, \rho_1) & \text{if } \rho(r, \iota_1, \iota_2) = \iota'' \\ (\sigma_1[\iota \mapsto \mathsf{newPart}_P(r, \texttt{null}, \iota_1, \iota_2)], \rho_1[(r, \iota_1, \iota_2) \mapsto \iota]) & \\ & \text{if } r = \texttt{Relation} \\ (\sigma_3, \rho_3) & \text{otherwise} \end{cases}$$
$$\begin{aligned} &\text{where } \iota \notin \mathsf{dom}(\sigma_1) \text{ or } \mathsf{dom}(\sigma_2) \\ &\quad r \neq \texttt{Relation} \Rightarrow \mathcal{R}_P(r) = (r', \_, \_, \_) \\ &\quad (\sigma_2, \rho_2) = \mathsf{addRel}_P(r', \iota_1, \iota_2, \sigma_1, \rho_1) \\ &\quad \sigma_3 = \sigma_2[\iota \mapsto \mathsf{newPart}_P(r, \rho_2(r', \iota_1, \iota_2), \iota_1, \iota_2)] \\ &\quad \rho_3 = \rho_2[(r, \iota_1, \iota_2) \mapsto \iota] \end{aligned}$$

$$\mathsf{remRel}_P(r, \iota_1, \iota_2, \rho) = \rho \setminus \{((r', \iota_1, \iota_2) \mapsto \iota) \mid \vdash r' \leq r\}$$

$$\mathsf{fldUpd}(\sigma, f, \iota, u) = \begin{cases} \sigma[\iota \mapsto \sigma(\iota)[f \mapsto u]] & \text{if } f \in \mathsf{dom}(\sigma(\iota)) \\ \mathsf{fldUpd}(\sigma, f, \iota', u) & \text{if } \sigma(\iota) = \langle\!\langle r, \iota', \_, \_ \| \ldots \rangle\!\rangle \end{cases}$$

Fig. 6. Definitions of auxiliary functions for creating relationship instances (newPart), for putting objects in relationships (addRel) and for removing objects from relationships (remRel). fldUpd demonstrates delegation of field updates to super-relationship instances.

It remains now to define the base cases for the operational semantics. We begin with RelJ's two relationship operations on an object address, ι: firstly, the objects related to ι by relationship r may be accessed using e.r; secondly, the instances of r that relate those objects to ι may be accessed with e:r so that relationship attributes may be read or modified:

OSRelObj: $\langle \Gamma, \sigma, \rho, \lambda, \iota.r \rangle \overset{P}{\leadsto} \langle \Gamma, \sigma, \rho, \lambda, \{\iota' \mid \exists \iota'' : \rho(r, \iota, \iota') = \iota''\}\rangle$

OSRelObjN: $\langle \Gamma, \sigma, \rho, \lambda, \texttt{null}.r \rangle \overset{P}{\leadsto} \langle \Gamma, \sigma, \rho, \lambda, \mathsf{NullPtrError}\rangle$

OSRelInst: $\langle \Gamma, \sigma, \rho, \lambda, \iota\texttt{:}r \rangle \overset{P}{\leadsto} \langle \Gamma, \sigma, \rho, \lambda, \{\iota'' \mid \exists \iota' : \rho(r, \iota, \iota') = \iota''\}\rangle$

OSRelObj and OSRelObjN give the semantics for obtaining the objects related to ι through r. Notice that the result is not just a matter of looking-up the result in a table; the objects are found by querying ρ. If `null` is the target of the lookup, a null-pointer error occurs. Similar rules are left for the appendix.

The pseudo-fields `from` and `to` provide access to the objects between which a relationship instance exists, returning the source and destination objects respectively:

OSFrom: $\langle \Gamma, \sigma, \rho, \lambda, \iota.\texttt{from} \rangle \overset{P}{\leadsto} \langle \Gamma, \sigma, \rho, \lambda, \iota' \rangle$ where $\sigma(\iota) = \langle\!\langle \_, \_, \iota', \_ \| \_ \rangle\!\rangle$

OSTo: $\langle \Gamma, \sigma, \rho, \lambda, \iota.\texttt{to} \rangle \overset{P}{\leadsto} \langle \Gamma, \sigma, \rho, \lambda, \iota' \rangle$ where $\sigma(\iota) = \langle\!\langle \_, \_, \_, \iota' \| \_ \rangle\!\rangle$

OSRelAdd and OSRelRem give semantics to the relationship addition and removal operators `add` and `rem` respectively, and are based entirely on addRel and remRel from Fig. 6:

OSRelAdd: $\langle \Gamma, \sigma_1, \rho_1, \lambda, r.\texttt{add}(\iota_1,\iota_2) \rangle \overset{P}{\leadsto} \langle \Gamma, \sigma_2, \rho_2, \lambda, \iota_3 \rangle$
where $(\sigma_2, \rho_2) = \mathsf{addRel}_P(r, \iota_1, \iota_2, \sigma_1, \rho_1)$ and $\iota_3 = \rho_2(r, \iota_1, \iota_2)$

OSRelRem1: $\langle \Gamma, \sigma, \rho_1, \lambda, r\texttt{.rem(}\iota_1\texttt{,}\iota_2\texttt{)}\rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho_2, \lambda, \rho_1(r, \iota_1, \iota_2)\rangle$
where $(r, \iota_1, \iota_2) \in \mathsf{dom}(\rho_1)$ and $\rho_2 = \mathsf{remRel}_P(r, \iota_1, \iota_2, \rho_1)$

OSRelRem2: $\langle \Gamma, \sigma, \rho, \lambda, r\texttt{.rem(}\iota_1\texttt{,}\iota_2\texttt{)}\rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \texttt{null}\rangle$
where $(r, \iota_1, \iota_2) \notin \mathsf{dom}(\rho)$

addRel adds an instance of r between ι_1 and ι_2 if such an instance does not already exist. With a recursive call, it also ensures that instances of r's super-relationships exist between ι_1 and ι_2, ensuring Invariant 1 is maintained.

remRel removes an instance of r from between ι_1 and ι_2, but does *not* alter the heap, only the relationship store, ρ. Again, to maintain Invariant 1, all instances of sub-relationships to r are similarly removed from between ι_1 and ι_2.

In the case of a relationship addition in expression context, a reference is returned to the relationship instance that was added. Relationship removal evaluates to the instance that was removed, if any. Where no such instance exists, `null` is returned.

Field update is performed with an auxiliary function fldUpd, also found in Fig. 6, which demonstrates the delegation of field lookup to super-relationship instances:

OSFldAss: $\langle \Gamma, \sigma, \rho, \lambda, \iota\texttt{.}f \texttt{ = } u\rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \mathsf{fldUpd}(\sigma, \iota, f, u), \rho, \lambda, u\rangle$

We conclude our discussion of the operational semantics with the two circumstances in which variables are scoped—method call, and the `for` iterator.

The semantics for method call is given in OSCall. Access to the formal parameter, x, local variables, $x_{1..i}$, and `this` must be scoped within the body of m, so we freshen these syntactic names to x', $x'_{1..i}$ and $x'_{\texttt{this}}$ in the style of Drossopoulou et al. [7].

OSCall: $\langle \Gamma_1, \sigma, \rho, \lambda_1, \iota.m(u)\rangle \overset{P}{\rightsquigarrow} \langle \Gamma_2, \sigma, \rho, \lambda_2, \texttt{\{ } s_2 \texttt{ return } e_2\texttt{; \}}\rangle$
where
$\sigma(\iota) = \langle\!\langle n \| \ldots \rangle\!\rangle$ or $\sigma(\iota) = \langle\!\langle n, \_, \_, \_ \| \ldots \rangle\!\rangle$
$\mathcal{MD}_{P,n}(m) = (x, \mathcal{L}, t_1, \_, s_1 \texttt{ return } e_1\texttt{;})$
$\mathsf{dom}(\mathcal{L}) = \{x_1, \ldots, x_i\}$
$x', x'_{\texttt{this}}, x'_1, \ldots, x'_i \notin \mathsf{dom}(\lambda_1)$
$\Gamma_2 = \Gamma_1[x' \mapsto t_1][x'_{\texttt{this}} \mapsto n][x'_{1..i} \mapsto \mathcal{L}(x_{1..i})]$
$\lambda_2 = \lambda_1[x' \mapsto u][x'_{\texttt{this}} \mapsto \iota][x'_{1..i} \mapsto \mathsf{initial}(\Gamma_2(x'_{1..i}))]$
$s_2 = s_1[x'/x][x'_{1..i}/x'_{1..i}][x'_{\texttt{this}}/\texttt{this}]$
$e_2 = e_1[x'/x][x'_{1..i}/x_{1..i}][x'_{\texttt{this}}/\texttt{this}]$

We extend the typing environment, Γ_2, with new local variable type bindings for the fresh names (as well as those for the formal parameter and `this`), and include appropriate initial values in the locals store, λ_2. Finally, the old syntactic names are updated in the method body, s, and `return` expression, e, by substitution.

A similar strategy is used to avoid binding clashes for the `for` iterator:

OSFor1: $\langle \Gamma, \sigma, \rho, \lambda, \texttt{for (}n\ x \texttt{ : } \emptyset\texttt{) \{}s_1\texttt{\}; } s_2\rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, s_2\rangle$

OSFor2: $\langle \Gamma_1, \sigma, \rho, \lambda_1, \texttt{for (}n\ x \texttt{ : } u\texttt{) \{}s_1\texttt{\}; } s_2\rangle \overset{P}{\rightsquigarrow}$
$\langle \Gamma_2, \sigma, \rho, \lambda_2, s_3 \texttt{ for (}n\ x \texttt{ : } (u \setminus \iota)\texttt{) \{}s_1\texttt{\}; } s_2\rangle$
where
$\iota \in u, x \neq x' \notin \mathsf{dom}(\lambda_1)$
$\Gamma_2 = \Gamma_1[x' \mapsto x], \lambda_2 = \lambda_1[x' \mapsto \iota], s_3 = s_1[x'/x]$

Iteration of the empty set evaluates immediately to 'skip', while iteration over the non-empty set picks an element from the set, assigns this to the iterator variable, and unfolds the statement block, in which the bound iterator variable is freshened. We do not specify the order in which the elements of u are bound to x.

5 Soundness

In this section we outline proofs of two key safety properties: that no type-correct program will get 'stuck'—except in a well-defined error state—and that types are preserved during program execution.

Firstly, however, we define some well-formedness properties of stores and values, so that we can check type preservation through subject reduction.

Value Typing and Well-formedness

We redefine our typing relation to include the store, σ, so that values may be typed—particularly important for showing subject-reduction. Typings of `true` and `false` with `boolean`, and of `null` with any valid nominal type are elided.

Firstly, an address has a type, n, if the object at that address in the store has a dynamic type (written $\mathsf{dynType}(\sigma(\iota))$) subordinate to n. This condition is then mapped over the members of a set of addresses in DTSet:

$$\text{(DTAddr)}\quad \frac{\vdash \mathsf{dynType}(\sigma(\iota)) \leq n}{P, \Gamma, \sigma \vdash \iota : n} \qquad \text{(DTSet)}\quad \frac{P \vdash n \quad \forall j \in 1..i : P, \Gamma, \sigma \vdash \iota_j : n}{P, \Gamma, \sigma \vdash \{\iota_1, \ldots, \iota_i\} : \texttt{set<}n\texttt{>}}$$

We also provide a typing rule for the method body construction introduced in Fig. 5:

$$\text{(DTMethBody)}\quad \frac{P, \Gamma, \sigma \vdash s \quad P, \Gamma, \sigma \vdash e : t}{P, \Gamma, \sigma \vdash \{\ s\ \texttt{return}\ e\texttt{;}\ \} : t}$$

We make use of a 'well-formed object' relation, $P, \sigma \vdash o \diamond_{\mathsf{inst}}$, when o is a well-formed object in some store, the rules for which follow:

$$\text{(WFField)}\quad \frac{\mathsf{dynType}(o) = n \quad \mathcal{FD}_{P,n}(f) = t \quad P, \emptyset, \sigma \vdash o(f) : t}{P, \sigma, o \vdash f \diamond_{\mathsf{fld}}}$$

WFField checks that the field f stores a value of appropriate type for its definition in class or relationship n, according the dynamic typing relation given above. This relation is mapped across the fields of classes and relationships in the following rules:

$$\text{(WFObject1)}\quad \frac{}{P, \sigma \vdash \langle\!\langle \texttt{Object} \| \rangle\!\rangle \diamond_{\mathsf{inst}}} \qquad \text{(WFRelInst1)}\quad \frac{\iota_1, \iota_2 \in \mathsf{dom}(\sigma)}{P, \sigma \vdash \langle\!\langle \texttt{Relation}, \texttt{null}, \iota_1, \iota_2 \| \rangle\!\rangle \diamond_{\mathsf{inst}}}$$

$$\frac{\{f_1,\ldots,f_i\} = \mathsf{dom}(\mathcal{FD}_{P,c}) \quad \forall j \in 1..i : P,\sigma,o \vdash f_j \diamond_{\mathsf{fld}}}{P,\sigma \vdash \langle\!\langle c \| f_1 : v_1,\ldots,f_i : v_i\rangle\!\rangle \diamond_{\mathsf{inst}}}\ (\text{WFO}\textsc{bject}2)$$

$$\frac{\begin{array}{c}\mathcal{R}_P(r) = (\mathsf{dynType}(\sigma(\iota)), n_1, n_2, \mathcal{F}, \_) \\ \{f_1,\ldots,f_i\} = \mathsf{dom}(\mathcal{F}) \\ \forall j \in 1..i : P,\sigma,o \vdash f_j \diamond_{\mathsf{fld}} \\ \vdash \mathsf{dynType}(\sigma(\iota_1)) \leq n_1 \\ \vdash \mathsf{dynType}(\sigma(\iota_2)) \leq n_2\end{array}}{P,\sigma \vdash \langle\!\langle r,\iota,\iota_1,\iota_2 \| f_1 : v_1,\ldots,f_i : v_i\rangle\!\rangle \diamond_{\mathsf{inst}}}\ (\text{WFR}\textsc{el}\text{I}\textsc{nst}2)$$

WFObject1 and WFRelInst1 specify that instances of `Object` and `Relation`, respectively, are valid. WFObject2, requires that all fields are well-formed and that the class instance has precisely those fields that were declared or inherited. WFRelInst2, checks that only those fields *immediately* declared in r are present in the relationship instance; that those fields are well-formed; that the super-instance, at ι, is present, and has a dynamic type equal to r's supertype; and that the r-instance sits between two instances of appropriate type according to r's definition.

We check that the relationships are properly specified in ρ according to the following two rules:

$$\frac{\sigma(\rho(\texttt{Relation},\iota_1,\iota_2)) = \langle\!\langle \texttt{Relation}, \texttt{null}, \iota_1, \iota_2 \| \rangle\!\rangle}{P,\sigma,\rho \vdash (\texttt{Relation},\iota_1,\iota_2) \diamond_{\mathsf{rel}}}\ (\text{WFR}\textsc{elation}1)$$

$$\frac{\begin{array}{c}\mathcal{R}_P(r) = (r', \_, \_, \_, \_) \\ (r',\iota_1,\iota_2) \in \mathsf{dom}(\rho) \\ \sigma(\rho(r,\iota_1,\iota_2)) = \langle\!\langle r, \rho(r',\iota_1,\iota_2), \iota_1, \iota_2 \| \ldots \rangle\!\rangle\end{array}}{P,\sigma,\rho \vdash (r,\iota_1,\iota_2) \diamond_{\mathsf{rel}}}\ (\text{WFR}\textsc{elation}2)$$

WFRelation2 ensures that the r-instance between ι_1 and ι_2 has a super-instance that also sits between ι_1 and ι_2. WFRelation1 acts as a base-case for `Relation`, instances of which do not take a super-instance.

We then map the conditions for well-formed instances, relations and local variables over the heap, σ, the relationship heap, ρ, and the locals map, λ:

$$\frac{\forall \iota \in \mathsf{dom}(\sigma) : P,\sigma \vdash \sigma(\iota) \diamond_{\mathsf{inst}}}{P \vdash \sigma \diamond_{\mathsf{heap}}}\ (\text{WFH}\textsc{eap}) \qquad \frac{\forall (r,\iota_1,\iota_2) \in \mathsf{dom}(\rho) : P,\sigma,\rho \vdash (r,\iota_1,\iota_2) \diamond_{\mathsf{rel}}}{P,\sigma \vdash \rho \diamond_{\mathsf{relheap}}}\ (\text{WFR}\textsc{el}\text{H}\textsc{eap})$$

$$\frac{\forall x \in \mathsf{dom}(\mathit{\Gamma}) : P,\mathit{\Gamma},\sigma \vdash \lambda(x) : \mathit{\Gamma}(x)}{P,\mathit{\Gamma},\sigma \vdash \lambda \diamond_{\mathsf{locals}}}\ (\text{WFL}\textsc{ocals})$$

We consider a configuration $\langle \mathit{\Gamma},\sigma,\rho,\lambda,s\rangle$ to be well-formed when σ, ρ and λ are well-formed, and where s is type-correct. Error configurations, $\langle \mathit{\Gamma},\sigma,\rho,\lambda,w\rangle$, are well-formed under similar conditions.

Safety

Type safety is shown by a subject reduction theorem, central to which is the idea that context substitution respects types:

Lemma 1 (Substitution). *For expressions e_1 and e_2, which are typed t_1 and t_2 respectively, where t_2 is a subtype of t_1 and where $\mathcal{E}_{\mathrm{e}}[e_1]$ is typed t_3, then $\mathcal{E}_{\mathrm{e}}[e_2]$ has a subtype of t_3.*

The proof follows by induction on the structure of the typing derivation. Next, we show type preservation, which follows naturally from the previous lemma, and by induction on the structure of the derivation of execution:

Theorem 1 (Subject Reduction). *In a well-typed program, P, where $\langle \Gamma_1, \sigma_1, \rho_1, \lambda_1, s_1 \rangle$ executes to a new configuration $\langle \Gamma_2, \sigma_2, \rho_2, \lambda_2, s_2 \rangle$, that configuration will be well-formed. Furthermore, $\Gamma_1 \subseteq \Gamma_2$ and all objects in σ_1 retain their dynamic type in σ_2.*

Similarly where the original configuration executes to an error configuration.

Finally, we show that a well-typed program may always perform an execution step:

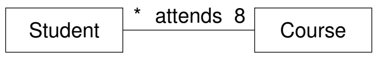
Theorem 2 (Progress). *For all well-typed programs, P, all well-formed configurations $\langle \Gamma_1, \sigma_1, \rho_1, \lambda_1, s_1 \rangle$ execute to either:*

i. *an error configuration $\langle \Gamma_2, \sigma_2, \rho_2, \lambda_2, w \rangle$, or*
ii. *a new statement configuration $\langle \Gamma_2, \sigma_2, \rho_2, \lambda_2, s_2 \rangle$*

By Theorems 1 and 2, any well-typed program can make a step to a new well-formed configuration: well-typed programs do not go wrong.

6 Restricting Multiplicities

In UML, associations can be annotated with *multiplicities*, which restrict the number of instances that may take part in any given relation. For example, it could be that every student attends exactly eight courses, but that a course may have any number of students:



More exotic multiplicities can include ranges ('1..7'), and comma-separated ranges ('1..7, 10..*'). There are a number of ways in which such restrictions could be expressed in RelJ. We describe below both a flexible, but dynamically checked approach, as well as a more restricted, statically checked approach.

6.1 Dynamic Approach

The use of a run-time check at every relationship addition would allow us to represent most of the possible multiplicities that can be expressed in UML. When, say, too many courses are added to the `Attends` relationship, an exception could be raised:

```
relationship Attends (many Student, 2 Course) { int mark; }
...
Attends.add(alice, programming);
Attends.add(alice, semantics);
Attends.add(alice, types);       // Exception!
```

We deviate from UML slightly: an association annotated at one end with '2' would always have exactly two associated instances. Instead, we interpret our `2` annotation on `Course` as '0..2' in UML notation: that is, courses start without any students.

6.2 Static Approach

Our preference, however, is for a static approach to the expression of multiplicities. While less flexible, we need not generate constraint-checking code for relationship additions, and we provide more robust guarantees that the multiplicity constraints are satisfied. Rather than give the formal details, we shall give an overview of this extension to RelJ.

We only allow `one` and `many` annotations. The former is equivalent to '0..1' in UML, the latter to '0..*':

```
relationship Attends (many Student, many Course);
relationship Failed (many PassedStudent, one Course);
```

In the declarations above, we see that students' course attendance is unrestricted, but that a `PassedStudent` may have failed at most one course.

We further restrict relationship inheritance so that a many-to-one relationship may only inherit from a many-to-one or many-to-many relationship. We impose similar restrictions on many-to-many and one-to-many relationship definitions. We then add to the invariants of Sect. 2.

Invariant 3 *For a relationship r, declared "`relationship` r `(`n_1`,` n_2`)`", where n_1 is annotated with `one`, there is at most one n_1-instance related through r to every n_2-instance. The converse is true where n_2 is annotated with `one`.*

There is a tension between Invariants 1 and 3. Consider the following relationship definitions, where a course can only be taught by a single lecturer, and where lecturers enjoy teaching hard courses, but teach them slowly:

```
relationship Teaches (one Lecturer, many Course);
relationship ExcitedlyTeaches extends Teaches
                             (one Lecturer, many HardCourse);
relationship SlowlyTeaches extends Teaches
                             (one Lecturer, many HardCourse);

charlie = new Lecturer();
deirdre = new Lecturer();
advancedWidgets = new HardCourse();
```

Suppose that `charlie ExcitedlyTeaches advancedWidgets`, then by Invariant 1, `charlie` also `Teaches advancedWidgets`.

Now suppose that `deirdre` is to slowly teach `advancedWidgets`:

```
SlowlyTeaches.add(deirdre, advancedWidgets);
```

By Invariant 1, `deirdre` must also be related to `advancedWidgets` via `Teaches`. However, by Invariant 3, `charlie` and `deirdre` cannot *both* `Teach advancedWidgets`. In our formalised semantics, we remove `charlie` from `Teaches` with `advancedWidgets`: the `add` becomes an assignment, rather than an addition, in this case. Furthermore, by Invariant 1, `charlie` cannot be in `ExcitedlyTeaches` with `advancedWidgets` once he has been removed from `Teaches`—therefore, he is also removed from `ExcitedlyTeaches`.

This behaviour, where not only sub-relationships of r are altered by a change to r's contents, but possibly also the contents of parents and siblings of r, might seem unexpected. At the same time, they make sense when examining examples, and provide a means for avoiding run-time checks.

7 Conclusion

In this paper, we have presented RelJ, a core fragment of Java that offers first-class support for first-class relationships. Unlike other work, we have formally specified our language; giving mathematical definitions of its type system and operational semantics. Given such definitions we are able prove an important correctness property of our language.

7.1 Related Work

Modelling languages like UML [9] and ER-diagrams [5] provide associations and relationships as core abstractions. Several database systems, for example object databases adhering to the ODMG standard [12], also provide relationships as primitives. Unfortunately, programming languages provide no first-class access to such primitives, so weak APIs must be used instead.

As we mentioned earlier, Rumbaugh [13] was the first to point out that relationships have an important rôle to play in general object-oriented languages, and gave an informal description of a language based on Smalltalk. However, the matter of relationship inheritance was mentioned only as an analogue to class inheritance, and there was no formal treatment of this or the language as a whole.

Noble has presented some patterns for programming with relationships [10]. In fact, many of these patterns could be used in translating RelJ programs to 'pure' Java. Noble and Grundy also suggested that relationships should be made explicit in object-oriented programs [11]. Again, neither work provides any concrete details of language support for relationships.

After completing the first draft of this work we discovered the paper by Albano, Ghelli and Orsini [1], which describes a language based on associations (relationships) for use in an object-oriented database environment. Their data model is quite different from ours; for example, they treat classes as containers, or extents [12]. Thus values can inhabit multiple classes, and classes also support multiple inheritance. In fact, classes turn out to be unary associations, which is the core abstraction in Albano et al.'s model.

Their model also provides a rich range of constraints; for example, surjectivity and cardinality constraints for associations, and disjointness constraints on classes. These are compiled to the appropriate runtime checks. (They take advantage of the underlying database infrastructure and utilize triggers and transactions.) Finally, they give no formal description of the language.

Our work, in contrast, takes as its starting point the Java object model and hence much of the complexity of Albano et al.'s model is simply not available. However, a notion of 'container' can be easily coded up. First assume a class `Singleton` and a single object of that class, called `default`. We can then define containers for the `Person` and `Student` classes of Fig. 2 as follows (where we assume a super-relationship `Extent` between `Singleton` and `Object` classes).

```
relationship Persons extends Extent
                    (Singleton, Person) {
}
relationship Students extends Persons
                    (Singleton, Student) {
}
```

So to place `Tom` in the `Persons` container we simply write `Persons.add(default, Tom)`. Similarly `Students.add(default, Jerry)` would add the object `Jerry` to the `Students` container, and by delegation also in the `Persons` container. The expression `default.Persons` would return the current contents of the `Persons` container. (Syntactic sugar could easily be added to make this code a little more compact.)

Interest in relationships is not restricted to modelling and programming languages. In the timeframe of the next generation of Microsoft Windows, codenamed 'Longhorn', the Windows storage subsystem will be replaced with a new system called *WinFS*. WinFS provides a database-like file store, the core of which is a collection of *items*, like objects, which represent data such as images, Outlook contacts, and user-defined items. The other key component of the WinFS data model is relationships, which are defined between items. WinFS thus represents a move away from the traditional tree-based file system hierarchy to an arbitrary graph-based file system, where the key abstraction is the relationship. At the time of writing, details of the API for WinFS are scarce, but it is clear that a language such as RelJ would provide a more direct programming framework, where various compile-time checks and optimizations would be possible. When the details of WinFS are finalized and made public, it would be interesting to compare various systems routines written in a language such as RelJ with those written using the APIs.

7.2 Further work

Clearly RelJ is just a first step in providing comprehensive first-class support of relationships in an object-oriented language. There are several features available in modelling languages, such as UML, that cannot currently be expressed in RelJ; notably, we only support relationships that are one-way. We hope to add relationships that may be traversed in both directions safely, as well as further investigating multiplicities.

In this paper we have not given details of how RelJ can be implemented. To support it directly in the runtime would require considerable extension of the JVM. The design and evaluation of such an extension is interesting future work. As an alternative, we have informally specified a systematic translation of RelJ into 'pure' Java. In the future, we plan to formalize this translation and prove it correct.

Another direction we wish to consider is extending RelJ with more query-like facilities (in a style similar to Cω [3]). For example, one might add a simple filter facility, e.g. the expression `alice.Attends[it.title.matches("*101")]` would return the beginners' courses that `alice` is currently attending. (The subexpression in square brackets is a simple boolean-valued expression, where `it` is bound to each element of the relationship in turn.)

Finally, we conclude by recording our hope that our language may provide a first step in the process of principled unification of modelling languages (UML, ER-diagrams), programming languages (Java, C$^\sharp$), and data query and specification languages (SQL, schema design).

Acknowledgments

Much of this work was completed whilst Bierman was at the University of Cambridge Computer Laboratory and supported by EU grants Appsem-II and EC FET-GC project IST-2001-33234 Pepito. Wren is currently supported by an EPSRC studentship. We are grateful to Sophia Drossopoulou and her group for useful comments on this work, as well as to Matthew Fairbairn, Giorgio Ghelli, Alan Mycroft, James Noble, Matthew Parkinson, Andrew Pitts, Peter Sewell and the attendees of FOOL 2005.

References

1. A. Albano, G. Ghelli, and R. Orsini. A relationship mechanism for a strongly typed object-oriented database programming language. In *Proceedings of VLDB*, 1991.
2. C. Anderson and S. Drossopoulou. δ: An imperative object-based calculus with delegation. In *Proceedings of USE*, 2002.
3. G. Bierman, E. Meijer, and W. Schulte. The essence of Cω. In *Proceedings of ECOOP*, 2005.
4. G. Bierman, M. Parkinson, and A. Pitts. MJ: A core imperative calculus for Java and Java with effects. Technical Report 563, University of Cambridge Computer Laboratory, 2003.
5. P. P.-S. Chen. The entity-relationship model – toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.
6. S. Drossopoulou. An abstract model of Java dynamic linking and loading. In *Proceedings of Types in Compilation (TIC)*, 2000.
7. S. Drossopoulou, T. Valkevych, and S. Eisenbach. Java type soundness revisited, September 2000.
8. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proceedings of POPL*, pages 171–183, 1998.
9. I. Jacobson, G. Booch, and J. Rumbaugh. *The unified software development process.* Addison-Wesley, 1999.

10. J. Noble. Basic relationship patterns. In *Pattern Languages of Program Design, vol. 4*. Addison Wesley, 1999.
11. J. Noble and J. Grundy. Explicit relationships in object-oriented development. In *Proceedings of TOOLS*, 1995.
12. R.G.G. Cattell et al. *The Object Data Standard: ODMG 3.0.* Morgan Kaufmann, 2000.
13. J. Rumbaugh. Relations as semantic constructs in an object-oriented language. In *Proceedings of OOPSLA*, pages 466–481, 1987.
14. J. Smith and D. Smith. Database abstractions: Aggregation and generalizations. *ACM Transactions on Database Systems*, 2(2):105–133, 1977.
15. P. Stevens and R. Pooley. *Using UML: software engineering with objects and components*. Addison-Wesley, 1999.
16. D. Ungar and R. B. Smith. Self: The power of simplicity. In *Proceedings of OOPSLA*, pages 227–242. ACM Press, 1987.
17. A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

A Details of Type System and Semantics

This appendix contains the details of the semantics not covered in the main body of the paper.

A.1 Typing Rules

In addition to the subtyping rules given in Sect. 3, the following rules populate the subtyping relation with the immediate supertypes provided by the language syntax, and give the reflexive, transitive closure:

$$\begin{array}{cccc}
(\text{STR}\textsc{ef}) & (\text{STT}\textsc{rans}) & (\text{STC}\textsc{lass}) & (\text{STR}\textsc{el}) \\
\dfrac{P \vdash t}{\vdash t \leq t} & \dfrac{\begin{array}{c}\vdash t_1 \leq t_2 \\ \vdash t_2 \leq t_3\end{array}}{\vdash t_1 \leq t_3} & \dfrac{\mathcal{C}(c_1) = (c_2, \_, \_)}{\vdash c_1 \leq c_2} & \dfrac{\mathcal{R}(r_1) = (r_2, \_, \_, \_, \_)}{\vdash r_1 \leq r_2}
\end{array}$$

The typing rules for the RelJ statements and expressions not typed in Sect. 3 are shown in Fig. 7.

We omit the typing of literal values `true`, `false`, `null` and `empty`, which are typed in the obvious way – `boolean`, n and `set<`n`>` respectively. Variables are typed by TSVar simply by look-up in the typing environment. Note that TSVar covers the type of `this` by its inclusion in VarName. New class-instance allocation is typed in the obvious way. The equality test is valid as long as both expressions are addresses. (Similar rules are required for e_1 and e_2 as `set<`$-$`>` or `boolean` types, but these are obvious and omitted.) Field look-up is typed from the field table of the receiver's static type. Rules TSVarAdd to TSFldSub demonstrate object addition and removal from set values. In all cases, the right-hand operand must be the address of an object with a type subordinate to the set's static type. The entire expression takes the right-hand operand's type. Variables and fields may be assigned values subordinate to the left-hand side's declared type. Method call is typed directly from the method look-up table. The

$$\text{(TSVar)}\ \frac{\Gamma(x) = t}{\Gamma \vdash x\colon t} \qquad \text{(TSNew)}\ \frac{P \vdash c}{\Gamma \vdash \texttt{new}\ c\texttt{()}\colon c} \qquad \text{(TSEq)}\ \frac{\begin{array}{c}\Gamma \vdash e_1\colon n\\ \Gamma \vdash e_2\colon n'\end{array}}{\Gamma \vdash e_1\ \texttt{==}\ e_2\colon \texttt{boolean}} \qquad \text{(TSFld)}\ \frac{\begin{array}{c}\Gamma \vdash e\colon n\\ \mathcal{FD}_n(f) = t\end{array}}{\Gamma \vdash e\texttt{.}f\colon t}$$

$$\text{(TSAdd)}\ \frac{\begin{array}{c}\Gamma \vdash e_1\colon \texttt{set<}n_1\texttt{>}\\ \Gamma \vdash e_2\colon n_2\\ \vdash n_1 \leq n_3\\ \vdash n_2 \leq n_3\end{array}}{\Gamma \vdash e_1\ \texttt{+}\ e_2\colon \texttt{set<}n_3\texttt{>}} \qquad \text{(TSSub)}\ \frac{\begin{array}{c}\Gamma \vdash e_1\colon \texttt{set<}n_1\texttt{>}\\ \Gamma \vdash e_2\colon n_2\\ \vdash n_1 \leq n_3\\ \vdash n_2 \leq n_3\end{array}}{\Gamma \vdash e_1\ \texttt{-}\ e_2\colon \texttt{set<}n_3\texttt{>}} \qquad \text{(TSAss)}\ \frac{\begin{array}{c}x \neq \texttt{this}\\ \Gamma \vdash x\colon t_1\\ \Gamma \vdash e\colon t_2\\ \vdash t_2 \leq t_1\end{array}}{\Gamma \vdash x\ \texttt{=}\ e\colon t_2}$$

$$\text{(TSFldAss)}\ \frac{\begin{array}{c}\Gamma \vdash e_1\colon n\\ \Gamma \vdash e_2\colon t_1\\ \mathcal{FD}_n(f) = t_2\\ \vdash t_1 \leq t_2\end{array}}{\Gamma \vdash e_1\texttt{.}f\ \texttt{=}\ e_2\colon t_1} \qquad \text{(TSCall)}\ \frac{\begin{array}{c}\Gamma \vdash e_1\colon n\\ \Gamma \vdash e_2\colon t_1\\ \mathcal{MD}_n(m) = (x, \mathcal{L}, t_2, t_3, \_)\\ \vdash t_1 \leq t_2\end{array}}{\Gamma \vdash e_1\texttt{.}m\texttt{(}e_2\texttt{)}\colon t_3} \qquad \text{(TSCond)}\ \frac{\begin{array}{c}\Gamma \vdash e\colon \texttt{boolean}\\ \Gamma \vdash s_1\\ \Gamma \vdash s_2\\ \Gamma \vdash s_3\end{array}}{\Gamma \vdash \texttt{if (}e\texttt{)}\ \{s_1\}\ \texttt{else}\ \{s_2\}\texttt{;}\ s_3}$$

$$\text{(TSSkip)}\ \frac{}{\Gamma \vdash \epsilon}$$

Fig. 7. The remaining type rules of RelJ

`for` statement was typed in the body of the paper. The conditional's typing-checking is standard, recalling that we do not assign types to statements. All statements require that their continuation statement is also well-typed, and we explicitly type the empty statement (ϵ), which is usually omitted in program text.

Finally, a program is well-typed if all of its classes and relationships are well-typed, if classes and relationships are disjoint, and if the subtyping relationship is antisymmetric:

$$\text{(TSProgram)}\ \frac{\begin{array}{c}\forall n \in \mathsf{dom}(\mathcal{C}_P) \cup \mathsf{dom}(\mathcal{R}_P) : P \vdash n\\ \forall n_1, n_2 : P \vdash n_1 \leq n_2 \wedge P \vdash n_2 \leq n_1 \Rightarrow n_1 = n_2\end{array}}{\vdash P}$$

A.2 Operational Semantics

First, we give full definitions of new, which returns an initialised class instance; initial, which returns an appropriate initial value for a variable of type t; dynType, which returns the dynamic type of an address in the store; and of fld, which returns the value of field f in the object at ι in store σ, delegating the field lookup to the superinstance as appropriate.

$$\mathsf{new}_P(c) = \begin{cases} \langle\!\langle \texttt{Object} \| \rangle\!\rangle & \text{if } c = \texttt{Object} \\ \langle\!\langle c \| f_1 : \mathsf{initial}_P(\mathcal{FD}_{P,c}(f_1)), \ldots, f_i : \mathsf{initial}_P(\mathcal{F}_{P,c}(f_i)) \rangle\!\rangle & \text{otherwise} \end{cases}$$

$$\text{where } \{f_1, f_2, \ldots, f_i\} = \mathsf{dom}(\mathcal{FD}_{P,c})$$

$$\mathsf{initial}_P(t) = \begin{cases} \texttt{null} & \text{if } t = n' \\ \texttt{false} & \text{if } t = \texttt{boolean} \\ \emptyset & \text{if } t = \texttt{set<}n\texttt{>} \end{cases}$$

$$\mathsf{dynType}(o) = n \text{ where } o = \langle\!\langle n \| \ldots \rangle\!\rangle \ \vee \ o = \langle\!\langle n, \_, \_, \_ \| \ldots \rangle\!\rangle$$

$$\mathsf{fld}(\sigma, f, \iota) = \begin{cases} \sigma(\iota)(f) & \text{if } f \in \mathsf{dom}(\sigma(\iota)) \text{ or} \\ \mathsf{fld}(\sigma, f, \iota') & \text{if } f \notin \mathsf{dom}(\sigma(\iota)) \ \wedge \ \sigma(\iota) = \langle\!\langle r, \iota', \_, \_ \| \ldots \rangle\!\rangle \end{cases}$$

The remaining rules of the operation semantics are then as follows:

OSEmpty:	$\langle \Gamma, \sigma, \rho, \lambda, \texttt{empty} \rangle \overset{P}{\leadsto} \langle \Gamma, \sigma, \rho, \lambda, \emptyset \rangle$
OSVar:	$\langle \Gamma, \sigma, \rho, \lambda, x \rangle \overset{P}{\leadsto} \langle \Gamma, \sigma, \rho, \lambda, \lambda(x) \rangle$
OSFldN:	$\langle \Gamma, \sigma, \rho, \lambda, \texttt{null}.f \rangle \langle \Gamma, \sigma, \rho, \lambda, \mathsf{NullPtrError} \rangle$
OSFld:	$\langle \Gamma, \sigma, \rho, \lambda, \iota.f \rangle \overset{P}{\leadsto} \langle \Gamma, \sigma, \rho, \lambda, \mathsf{fld}(\sigma, \iota, f) \rangle$
OSRelInstN:	$\langle \Gamma, \sigma, \rho, \lambda, \texttt{null:}r \rangle \overset{P}{\leadsto} \langle \Gamma, \sigma, \rho, \lambda, \mathsf{NullPtrError} \rangle$
OSEq:	$\langle \Gamma, \sigma, \rho, \lambda, u \texttt{ == } u \rangle \overset{P}{\leadsto} \langle \Gamma, \sigma, \rho, \lambda, \texttt{true} \rangle$
OSNeq:	$\langle \Gamma, \sigma, \rho, \lambda, u \texttt{ == } u' \rangle \overset{P}{\leadsto} \langle \Gamma, \sigma, \rho, \lambda, \texttt{false} \rangle$ where $u \neq u'$
OSNew:	$\langle \Gamma, \sigma, \rho, \lambda, \texttt{new } c\texttt{()} \rangle \overset{P}{\leadsto} \langle \Gamma, \sigma[\iota \mapsto \mathsf{new}_P(c)], \rho, \lambda, \iota \rangle$ where $\iota \notin \mathsf{dom}(\sigma)$
OSBody:	$\langle \Gamma, \sigma, \rho, \lambda, \{ \texttt{ return } u\texttt{; } \} \rangle \overset{P}{\leadsto} \langle \Gamma, \sigma, \rho, \lambda, u \rangle$
OSAdd:	$\langle \Gamma, \sigma, \rho, \lambda, u \texttt{ + } \iota \rangle \overset{P}{\leadsto} \langle \Gamma, \sigma, \rho, \lambda, u \cup \{\iota\} \rangle$
OSAddN:	$\langle \Gamma, \sigma, \rho, \lambda, u \texttt{ + null} \rangle \overset{P}{\leadsto} \langle \Gamma, \sigma, \rho, \lambda, \mathsf{NullPtrError} \rangle$
OSSub:	$\langle \Gamma, \sigma, \rho, \lambda, u \texttt{ - } \iota \rangle \overset{P}{\leadsto} \langle \Gamma, \sigma, \rho, \lambda, u \setminus \{\iota\} \rangle$
OSSubN:	$\langle \Gamma, \sigma, \rho, \lambda, u \texttt{ - null} \rangle \overset{P}{\leadsto} \langle \Gamma, \sigma, \rho, \lambda, \mathsf{NullPtrError} \rangle$
OSVarAss:	$\langle \Gamma, \sigma, \rho, \lambda, x \texttt{ = } u \rangle \overset{P}{\leadsto} \langle \Gamma, \sigma, \rho, \lambda[x \mapsto u], u \rangle$
OSFldAssN:	$\langle \Gamma, \sigma, \rho, \lambda, \texttt{null}.f \texttt{ = } u \rangle \overset{P}{\leadsto} \langle \Gamma, \sigma, \rho, \lambda, \mathsf{NullPtrError} \rangle$
OSRelAddN:	$\langle \Gamma, \sigma, \rho, \lambda, r\texttt{.add(}\iota_1^{\texttt{null}}\texttt{,}\iota_2^{\texttt{null}}\texttt{)} \rangle \overset{P}{\leadsto} \langle \Gamma, \sigma, \rho, \lambda, \mathsf{NullPtrError} \rangle$ where $\iota_1^{\texttt{null}} = \texttt{null}$ or $\iota_2^{\texttt{null}} = \texttt{null}$
OSRelRemN:	$\langle \Gamma, \sigma, \rho, \lambda, r\texttt{.rem(}\iota_1^{\texttt{null}}\texttt{,}\iota_2^{\texttt{null}}\texttt{)} \rangle \overset{P}{\leadsto} \langle \Gamma, \sigma, \rho, \lambda, \mathsf{NullPtrError} \rangle$ where $\iota_1^{\texttt{null}} = \texttt{null}$ or $\iota_2^{\texttt{null}} = \texttt{null}$
OSCallN:	$\langle \Gamma, \sigma, \rho, \lambda, \texttt{null}.m\texttt{(}u\texttt{)} \rangle \overset{P}{\leadsto} \langle \Gamma, \sigma, \rho, \lambda, \mathsf{NullPtrError} \rangle$
OSStat:	$\langle \Gamma, \sigma, \rho, \lambda, u\texttt{; } s \rangle \overset{P}{\leadsto} \langle \Gamma, \sigma, \rho, \lambda, s \rangle$
OSCondT:	$\langle \Gamma, \sigma, \rho, \lambda, \texttt{if (true) } \{s_1\} \texttt{ else } \{s_2\}\texttt{; } s_3 \rangle \overset{P}{\leadsto} \langle \Gamma, \sigma, \rho, \lambda, s_1 \ s_3 \rangle$
OSCondF:	$\langle \Gamma, \sigma, \rho, \lambda, \texttt{if (false) } \{s_1\} \texttt{ else } \{s_2\}\texttt{; } s_3 \rangle \overset{P}{\leadsto} \langle \Gamma, \sigma, \rho, \lambda, s_2 \ s_3 \rangle$