First-class relationships in an object-oriented language

Gavin Bierman
Microsoft Research, Cambridge
gmb@microsoft.com

Alisdair Wren
University of Cambridge Computer Laboratory
aw345@cl.cam.ac.uk

Abstract

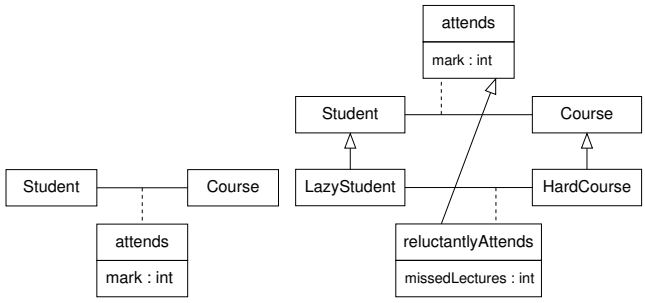
In this paper we investigate the addition of first-class relationships to a prototypical object-oriented programming language (a "middleweight" fragment of Java). We provide language-level constructs to declare relationships between classes and to manipulate relationship instances. We allow relationships 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 [8]—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

Figure 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) [4] to Unified Modelling Language (UML) [7]. 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 [11]. Noble and Grundy also proposed that relationships should persist from the modelling to the implementation stage of program development [9]. 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.

FOOL 2005 15 January 2005, Long Beach, California

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 we describe precisely 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 [3]) 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 §2 we introduce our calculus and give a grammar. The type system of RelJ is defined in §3, where the formal notion of subtyping is discussed and well-typed RelJ programs are characterized. §4 gives the dynamics of RelJ with a small-step operational semantics. We outline a proof of type soundness for RelJ in §5. §6 describes an extension to RelJ which allows the addition of UML-style multiplicity restrictions to relationships. Finally, in §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 [3, 5, 6]. 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.

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:

```
relationship r extends r′
        from n to n′ { FieldDecl* }
```

This defines a new relationship, r, with a number of type/field name pairs, FieldDecl*. To simplify the presentation, and to save space, we do not allow relationships to have methods (though these can be easily added in the obvious way). 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 E/R modelling [12]. An example is shown in Figure 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.

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 of dangling pointers: these are discussed later.

We also support relationship inheritance, which is denoted idiomatically as inheritance between association classes (Figure 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.

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 such as Self [14] and, more recently, δ [2]. Consider the RelJ code for a simple example, adapted from Pooley and Stevens [13], which is shown in Figure 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

```
class Student {
   String name;
}
class LazyStudent extends Student {
   int    hoursOfSleep;
}
class Course {
   String title;
}
class Tutor {
   String name;
}
relationship Attends
        from Student to Course {
   int mark;
}
relationship ReluctantlyAttends
        extends Attends
        from LazyStudent to Course {
   int missedLectures;
}
relationship CompulsorilyAttends
        extends Attends
        from Student to Course {
   String reason;
}
relationship Recommends
        from Tutor to Attends {
   String reason;
}
...

alice = new LazyStudent();
programming = new Course();
typeSystems = new Course();
alice.Attends += programming;
      // Alice attends Programming
alice.ReluctantlyAttends += typeSystems;
      // Alice reluctantly attends Type Systems
for (Course c : alice.Attends) {
  print "Attends: " + c.title;
};
      // Prints:
      //    Attends: Programming
      //    Attends: Type Systems
```

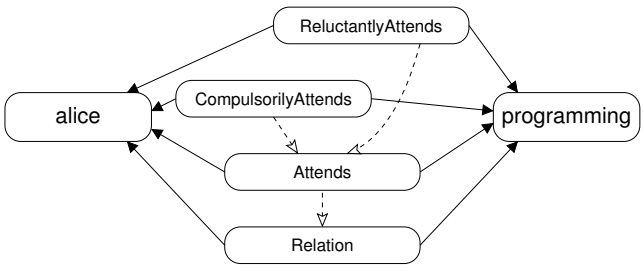


Figure 2. Example RelJ code and possible instantiation

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 Figure 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 Figure 2 to remove the fact that `Alice` attends `programming`:

```
...
alice.Attends -= programming;
      // Remove Alice attending 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, `+=` returns the relationship instance that was created: this provides a short-cut for setting the new instance's fields. For regularity, `-=` returns the value of the expression on the right, in common with other assignments.

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 = (bob.Attends += databases);
bobdb.mark = 99;
for (Course cs : bob.Attends) {
  print cs.title;
};
       // Prints DB 101
print bobdb.mark;
       // Prints 99
bob.Attends -= 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 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 opted for the third case. 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.

Language Definition

We give the grammar for RelJ programs and types in Figure 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 generic class type, but is a generic value type, much like the generic literal types used by the ODMG [10].[1] 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 offers 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. We use the notation $\overline{s}$ to denote a possibly empty sequence of statements. 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. (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 Figure 4.

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 $(r', n, n', \mathcal{F})$ where r' is the super-relationship; n and n' are the types between which the relationship is formed (the *source* and *destination* respectively); and $\mathcal{F}$ is a field map, as found in class definitions. As for classes, we write $\mathcal{F}_r$ for r's field definition map.

In summary, RelJ offers the following operations to manipulate relationships: $e.r$ finds the objects related to the result of e through relationship r; $e:r$ 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:

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

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

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

$$
\begin{array}{rcll}
p \in \mathsf{Program} & ::= & \mathsf{ClassDecl}^*\ \mathsf{RelDecl}^* & \\
\mathsf{ClassDecl} & ::= & \texttt{class}\ c\ \texttt{extends}\ c'\ \{\ \mathsf{FieldDecl}^*\ \mathsf{MethDecl}^*\ \} & \\
\mathsf{RelDecl} & ::= & \texttt{relationship}\ r\ \texttt{extends}\ r' & \\
 & & \quad \texttt{from}\ n\ \texttt{to}\ n'\ \{\ \mathsf{FieldDecl}^*\ \} & \\
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} & ::= & \{\ \overline{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 \mid & \text{field access} \\
 & & e\texttt{.}r & \text{related object access} \\
e \in \mathsf{Expression} & ::= & v \mid & \text{value} \\
 & & l \mid & \text{l-value} \\
 & & e_1\ \texttt{==}\ e_2 \mid & \text{equality test} \\
 & & 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} \\
 & & x\ \texttt{=}\ e \mid e\texttt{.}f\ \texttt{=}\ e' \mid & \text{assignment} \\
 & & e\texttt{.}m\texttt{(}e'\texttt{)} \mid & \text{method call} \\
 & & l\ \texttt{+=}\ e \mid & \text{set addition} \\
 & & l\ \texttt{-=}\ e & \text{set removal} \\
s \in \mathsf{Statement} & ::= & \texttt{;} \mid & \text{skip} \\
 & & se\texttt{;} \mid & \text{expression} \\
 & & \texttt{if (}e\texttt{) \{}\overline{s_1}\texttt{\} else \{}\overline{s_2}\texttt{\};} \mid & \text{conditional} \\
 & & \texttt{for (}n\ x\ \texttt{:}\ e\texttt{) \{}\overline{s}\texttt{\};} & \text{set iteration}
\end{array}
$$

Figure 3. The grammar of RelJ types and programs

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

Figure 4. Signatures of class and relationship tables

To unify the relationship and class hierarchies—desirable in the absence of generics—we take `Relation` as a subtype of `Object` in rule STOBJECT.[2]

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:

$$\mathcal{FD}_{P,c}(f) = \begin{cases} \mathcal{F}_{P,c}(f) & \text{if } f \in \mathsf{dom}(\mathcal{F}_{P,c}) \\ \mathcal{FD}_{c'}(f) & \text{if } f \notin \mathsf{dom}(\mathcal{F}_{P,c}) \\ & \quad \text{and } \mathcal{C}_P(c) = (c', \_, \_) \\ \bot & \text{otherwise} \end{cases}$$

$$\mathcal{MD}_{P,c}(m) = \begin{cases} \mathcal{M}_{P,c}(m) & \text{if } f \in \mathsf{dom}(\mathcal{M}_{P,c}) \\ \mathcal{MD}_{P,c'}(m) & \text{if } f \notin \mathsf{dom}(\mathcal{M}_{P,c}) \\ & \quad \text{and } \mathcal{C}_P(c) = (c', \_, \_) \\ \bot & \text{otherwise} \end{cases}$$

$\mathcal{FD}$ can be defined similarly for relationships, using $\mathcal{R}$ in place of $\mathcal{C}$.

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:

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

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_2. 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.

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

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.

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

$$\text{(TSRELADD)}\ \frac{\mathcal{R}(r) = (\_, n_1, n_2, \_) \quad \Gamma \vdash e_1 : n_3 \quad \Gamma \vdash e_2 : n_4 \quad \vdash n_3 \leq n_1 \quad \vdash n_4 \leq n_2}{\Gamma \vdash e_1.r\ \texttt{+=}\ e_2 : r} \qquad \text{(TSRELSUB)}\ \frac{\mathcal{R}(r) = (\_, n_1, n_2, \_) \quad \Gamma \vdash e_1 : n_3 \quad \Gamma \vdash e_2 : n_4 \quad \vdash n_3 \leq n_1 \quad \vdash n_4 \leq n_2}{\Gamma \vdash e_1.r\ \texttt{-=}\ e_2 : n_4}$$

Finally, TSRELADD and TSRELSUB 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. For TSRELADD, the result will be an instance of r: the instance that was created. For TSRELSUB, the result of e_2 is returned.

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:

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

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 we require an explicit type for the iterating variable to be consistent with the Java 5.0 syntax; although there is no reason why this type couldn't be inferred. For simplicity, 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:

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

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

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

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.

(TSMethod)

$$\frac{\begin{array}{c}\mathcal{C}_P(c) = (c', \_, \mathcal{M}_c) \\ \mathcal{M}_c(m) = (x, \mathcal{L}, t_1, t_2, \{\ \overline{s}\ \texttt{return}\ e;\ \}) \\ 1.\quad t_1 \in \mathsf{validTypes}_P \\ 2.\quad \texttt{this}, x \notin \mathsf{dom}(\mathcal{L}) \\ 3.\quad \{x \mapsto t_1, \texttt{this} \mapsto c\} \cup \mathcal{L} \vdash \overline{s} \\ 4.\quad \{x \mapsto t_1, \texttt{this} \mapsto c\} \cup \mathcal{L} \vdash e : t_2' \\ 5.\quad \vdash t_2' \leq t_2 \\ 6.\quad \mathcal{MD}_{c'}(m) = (\_, \_, t_3, t_4, \_) \Rightarrow \vdash t_3 \leq t_1 \ \wedge\ \vdash t_2 \leq t_4\end{array}}{P, c \vdash m}$$

TSMethod checks (1) that the input type of method m in class c 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 superclass 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 superclass' version of m can be used. We then specify the validity of classes and relationships:

(TSClass)

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

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. Relationships are similarly checked:

(TSRelationship)

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

TSRelationship imposes many of the same restrictions as TSClass, except without method checking, and 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 [15], and use variable renaming to avoid the need for an explicit frame stack [5].

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

$$\begin{array}{rcl}\iota & \in & \mathsf{Address} \\ \iota^{\texttt{null}} & \in & \mathsf{Address} \cup \{\texttt{null}\} \\ u & \in & \mathsf{DynValue} = \{\texttt{null}, \texttt{true}, \texttt{false}\} \\ & & \quad \cup\ \mathsf{Address} \cup \mathcal{P}(\mathsf{Address}) \\ w & \in & \mathsf{Error} = \{\mathsf{NullPtrError}\} \\ o & \in & \mathsf{Object} \\ \sigma & : & \mathsf{Address} \to \mathsf{Object} \\ \rho & : & (\mathsf{Address} \times \mathsf{Address} \times \mathsf{RelName}) \to \mathsf{Address} \\ \lambda & : & \mathsf{VarName} \to \mathsf{DynValue}\end{array}$$

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 Expression as follows:

$$\begin{array}{lll}e \in \mathsf{DynExpression} ::= & & \\ \quad u \mid & & \text{dynamic values} \\ \quad mb \mid & & \text{method body} \\ \quad \ldots & & \text{terms from Expression grammar}\end{array}$$

DynLValue and DynStatement are generated from LValue and 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.

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 ($\sigma[\iota \mapsto o_{new}]$) and to objects ($o[f \mapsto v_{new}]$).

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, $s[x'/x]$, and statement sequences, $\overline{s}[x'/x]$.

$\mathcal{E}_l \in$ LValContext ::=

	$\mathcal{E}_e\,.\,f$	field
$\mid$	$\mathcal{E}_e\,.r$	relation

$\mathcal{E}_e \in$ ExpContext ::=

	$\bullet$	hole
$\mid$	$\mathcal{E}_l$	l-value
$\mid$	$\mathcal{E}_e$ `==` $e \mid u$ `==` $\mathcal{E}_e$	equality test
$\mid$	$\mathcal{E}_e$`:`r	relationship access
$\mid$	$\mathcal{E}_e$`.from` $\mid \mathcal{E}_e$`.to`	relationship from/to

$\mathcal{E}_e, \mathcal{E}_{se} \in$ SExpContext ::=

	`{` $\mathcal{E}$ `return` e`;` `}`	method body
$\mid$	`{ return` $\mathcal{E}_e$`; }`	method body
$\mid$	$\mathcal{E}_e\,.\,f$ `=` $e \mid x$ `=` $\mathcal{E}_e \mid u.f$ `=` $\mathcal{E}_e$	assignment
$\mid$	$\mathcal{E}_e\,.\,m(e') \mid u\,.\,m(\mathcal{E}_e)$	method call
$\mid$	$\mathcal{E}_l$ `+=` $e \mid x$ `+=` $\mathcal{E}_e$	set addition
$\mid$	$u.f$ `+=` $\mathcal{E}_e \mid u.r$ `+=` $\mathcal{E}_e$	
$\mid$	$\mathcal{E}_l$ `-=` $e \mid x$ `-=` $\mathcal{E}_e$	set removal
$\mid$	$u.f$ `-=` $\mathcal{E}_e \mid u.r$ `-=` $\mathcal{E}_e$	

$\mathcal{E}_s \in$ StatContext ::=

	$\mathcal{E}_{se}$`;`	expression
$\mid$	`for (`$n\ x$ `:` $\mathcal{E}_e$`) {`$\overline{s}$`};`	set iteration
$\mid$	`if (`$\mathcal{E}_e$`) {`$\overline{s_1}$`} else {`$\overline{s_2}$`};`	conditional

$\mathcal{E} \in$ Context ::=

	$\mathcal{E}_s\ \overline{s}$	statement sequence

Figure 5. Grammar for evaluation contexts

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. It can be shown that all (non-value) expressions and statements may be decomposed into a context with a (strictly smaller) expression in hole position.

A *configuration* in the semantics is a 5-tuple of typing environment, heap, relationship store, locals map, and a statement sequence: $\langle \Gamma, \sigma, \rho, \lambda, \overline{s} \rangle$. An *error configuration* is a configuration $\langle \Gamma, \sigma, \rho, \lambda, w \rangle$, with an error in place of a statement sequence. 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, below, which also lifts to statement contexts:

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

We also define OSError, which propagates an error raised by a sub-expression upward through the syntax:

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

It remains now to define the base cases for the operational semantics. Firstly, OSSeq1 strips 'skip' statements from the front of statement sequences, while OSSeq2 executes the statement at the head of a statement sequence:

$$(\text{OSSeq1})\ \frac{}{\langle \Gamma, \sigma, \rho, \lambda, \texttt{;}s \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, s \rangle}$$

$$(\text{OSSeq2})\ \frac{\langle \Gamma_1, \sigma_1, \rho_1, \lambda_1, s_1 \rangle \overset{P}{\rightsquigarrow} \langle \Gamma_2, \sigma_2, \rho_2, \lambda_2, s_2 \rangle}{\langle \Gamma_1, \sigma_1, \rho_1, \lambda_1, s_1\ \overline{s} \rangle \overset{P}{\rightsquigarrow} \langle \Gamma_2, \sigma_2, \rho_2, \lambda_2, s_2\ \overline{s} \rangle}$$

RelJ provides two relationship operations on an expression, e, returning 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 $e\texttt{:}r$ so that relationship attributes may read or modified:

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

$$(\text{OSRelObjN})\quad \langle \Gamma, \sigma, \rho, \lambda, \texttt{null}\,.\,r \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \mathsf{NullPtrError} \rangle$$

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

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:

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

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

OSRelAdd and OSRelSub give semantics to the relationship addition and removal operators `+=` and `-=` respectively, and are based entirely on addRel and delRel from Figure 6:

$$(\text{OSRelAdd})\quad \langle \Gamma, \sigma_1, \rho_1, \lambda, \iota_1\,.\,r\ \texttt{+=}\ \iota_2 \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma_2, \rho_2, \lambda, \iota_3 \rangle \quad \text{where } (\sigma_2, \rho_2) = \mathsf{addRel}_P(r, \iota_1, \iota_2, \sigma_1, \rho_1),\ \iota_3 = \rho_2(r, \iota_1, \iota_2)$$

$$(\text{OSRelSub})\quad \langle \Gamma, \sigma, \rho_1, \lambda, \iota_1\,.\,r\ \texttt{-=}\ \iota_2 \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho_2, \lambda, \iota_2 \rangle \quad \text{where } \rho_2 = \mathsf{delRel}_P(r, \iota_1, \iota_2, \rho_1)$$

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.

delRel 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 simply evaluates to the

$$\mathsf{newPart}_P(r, \iota^{\texttt{null}}, \iota_1, \iota_2) = \langle\!\langle r, \iota^{\texttt{null}}, \iota_1, \iota_2 \| f_1 : v_1,\ f_2 : v_2, \ldots,\ f_i : v_i \rangle\!\rangle$$
$$\text{where } \{f_1,\ f_2, \ldots,\ f_i\} = \mathsf{dom}(\mathcal{F}_{P,c})$$
$$v_i = \mathsf{initial}_P(\mathcal{F}_{P,r}(f_i))$$

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

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

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

Figure 6. Definitions of auxiliary functions for creating relationship instances (newPart, in which $\iota^{\texttt{null}}$ ranges over addresses and the undefined value), for putting objects in relationships (addRel) and for removing objects from relationships (delRel). fldUpd demonstrates delegation of field updates to super-relationship instances.

right-hand side of the assignment, in common with other assignment operators.

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

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

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:

(OSCALL) $\langle \Gamma_1, \sigma, \rho, \lambda_1, \iota.m(u) \rangle \overset{P}{\rightsquigarrow} \langle \Gamma_2, \sigma, \rho, \lambda_2, \{\ \overline{s_2}\ \texttt{return}\ e_2\texttt{;}\ \} \rangle$

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

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. [5]. 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, $\overline{s}$, and `return` expression, e, by substitution.

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

(OSFOR1) $\langle \Gamma, \sigma, \rho, \lambda, \texttt{for}\ (n\ x : \emptyset)\ \{\overline{s}\}\texttt{;} \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \texttt{;} \rangle$

(OSFOR2) $\langle \Gamma_1, \sigma, \rho, \lambda_1, \texttt{for}\ (n\ x : u)\ \{\overline{s_1}\}\texttt{;} \rangle \overset{P}{\rightsquigarrow} \langle \Gamma_2, \sigma, \rho, \lambda_2, \overline{s_2}\ \texttt{for}\ (n\ x : (u \setminus \iota))\ \{\overline{s}\}\texttt{;} \rangle$

where
$u \in \mathcal{P}(\mathsf{Address}), \iota \in u$
$x' \notin \mathsf{dom}(\lambda_1)$
$\Gamma_2 = \Gamma_1[x' \mapsto x]$
$\lambda_2 = \lambda_1[x' \mapsto \iota]$
$\overline{s_2} = \overline{s_1}[x'/x]$

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:

$$\text{(DTADDR)}\ \frac{\vdash \mathsf{dynType}(\sigma(\iota)) \leq n}{P, \Gamma, \sigma \vdash \iota : n}$$

$$\text{(DTSET)}\ \frac{\forall j \in 1..i : P, \Gamma, \sigma \vdash \iota_j : n}{P, \Gamma, \sigma \vdash \{\iota_1, \ldots, \iota_i\} : \texttt{set<n>}}$$

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

$$\text{(DTMETHBODY)}\ \frac{\begin{array}{c} P, \Gamma, \sigma \vdash \overline{s} \\ P, \Gamma, \sigma \vdash e : t \end{array}}{P, \Gamma, \sigma \vdash \{\ \overline{s}\ \texttt{return}\ e;\ \} : t}$$

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:

$$\text{(WFFIELD)}\ \frac{\begin{array}{c} \mathsf{dynType}(o) = n \\ \mathcal{FD}_{P,n}(f) = t \\ P, \emptyset, \sigma \vdash o(f) : t \end{array}}{P, \sigma, o \vdash f \diamond_{\mathsf{fld}}}$$

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:

$$\text{(WFOBJECT1)}\ \frac{}{P, \sigma \vdash \langle\!\langle \texttt{Object} \| \rangle\!\rangle \diamond_{\mathsf{inst}}}$$

$$\text{(WFOBJECT2)}\ \frac{\begin{array}{c} \{f_1, \ldots, f_i\} = \mathsf{dom}(\mathcal{FD}_{P,c}) \\ \forall j \in 1..i : P, \sigma, o \vdash f_j \diamond_{\mathsf{fld}} \end{array}}{P, \sigma \vdash \langle\!\langle c \| f_1 : v_1, \ldots, f_i : v_i \rangle\!\rangle \diamond_{\mathsf{inst}}}$$

$$\text{(WFRELINST1)}\ \frac{\iota_1, \iota_2 \in \mathsf{dom}(\sigma)}{P, \sigma \vdash \langle\!\langle \texttt{Relation}, \texttt{null}, \iota_1, \iota_2 \| \rangle\!\rangle \diamond_{\mathsf{inst}}}$$

$$\text{(WFRELINST2)}\ \frac{\begin{array}{c} \{f_1, \ldots, f_i\} = \mathsf{dom}(\mathcal{F}_{P,r}) \\ \forall j \in 1..i : P, \sigma, o \vdash f_j \diamond_{\mathsf{fld}} \\ \mathcal{R}_P(r) = (\mathsf{dynType}(\sigma(\iota)), n_1, n_2, \_) \\ \vdash \mathsf{dynType}(\sigma(\iota_1)) \leq n_1 \\ \vdash \mathsf{dynType}(\sigma(\iota_2)) \leq n_2 \end{array}}{P, \sigma \vdash \langle\!\langle r, \iota, \iota_1, \iota_2 \| f_1 : v_1, \ldots, f_i : v_i \rangle\!\rangle \diamond_{\mathsf{inst}}}$$

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:

$$\text{(WFRELATION1)}\ \frac{\sigma(\rho(\texttt{Relation}, \iota_1, \iota_2)) = \langle\!\langle \texttt{Relation}, \texttt{null}, \iota_1, \iota_2 \| \rangle\!\rangle}{P, \sigma, \rho \vdash (\texttt{Relation}, \iota_1, \iota_2) \diamond_{\mathsf{rel}}}$$

$$\text{(WFRELATION2)}\ \frac{\begin{array}{c} \mathcal{R}_P(r) = (r', \_, \_, \_) \\ (r', \iota_1, \iota_2) \in \mathsf{dom}(\rho) \\ \sigma(\rho(r, \iota_1, \iota_2)) = \langle\!\langle r, \rho(r', \iota_1, \iota_2), \iota_1, \iota_2 \| \ldots \rangle\!\rangle \end{array}}{P, \sigma, \rho \vdash (r, \iota_1, \iota_2) \diamond_{\mathsf{rel}}}$$

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:

$$\text{(WFHEAP)}\ \frac{\forall \iota \in \mathsf{dom}(\sigma) : P, \sigma \vdash \sigma(\iota) \diamond_{\mathsf{inst}}}{P \vdash \sigma \diamond_{\mathsf{heap}}}$$

$$\text{(WFRELHEAP)}\ \frac{\forall (r, \iota_1, \iota_2) \in \mathsf{dom}(\rho) : P, \sigma, \rho \vdash (r, \iota_1, \iota_2) \diamond_{\mathsf{rel}}}{P, \sigma \vdash \rho \diamond_{\mathsf{relheap}}}$$

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

We consider a configuration $\langle \Gamma, \sigma, \rho, \lambda, \overline{s} \rangle$ to be well-formed when σ, ρ and λ are well-formed, and where $\overline{s}$ is type-correct. Error configurations, $\langle \Gamma, \sigma, \rho, \lambda, w \rangle$, 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 $\mathcal{E}[e_1]$ is typed t_3, then $\mathcal{E}[e_2]$ 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 2 (Subject Reduction). *In a well-typed program, P, where $\langle \Gamma_1, \sigma_1, \rho_1, \lambda_1, \overline{s_1} \rangle$ executes to a new configuration $\langle \Gamma_2, \sigma_2, \rho_2, \lambda_2, \overline{s_2} \rangle$, 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 3 (Progress). *For all well-typed programs, P, all well-formed configurations $\langle \Gamma_1, \sigma_1, \rho_1, \lambda_1, \overline{s_1} \rangle$ execute to either:*

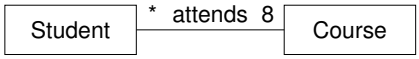
i. an error configuration $\langle \Gamma_2, \sigma_2, \rho_2, \lambda_2, w \rangle$, or

ii. a new statement configuration $\langle \Gamma_2, \sigma_2, \rho_2, \lambda_2, \overline{s_2} \rangle$

By Theorems 2 and 3, 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. Here, we describe both a flexible, but dynamically checked approach, as well as a more restricted, statically checked approach:

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
       from many Student to 2 Course {
   int mark;
}
...
alice.Attends += Programming;
alice.Attends += Semantics;
alice.Attends += 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.

Static approach

Our preference, however, is for a static approach to the expression of multiplicities. While less flexible, we need not generate wrapper 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
       from many Student to many Course;
relationship Failed
       from many PassedStudent to 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 §2.

INVARIANT 3. *For a relationship r, declared "`relationship r from` n_1 `to` 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
       from one Lecturer to many Course;
relationship ExcitedlyTeaches extends Teaches
       from one Lecturer to many HardCourse;
relationship SlowlyTeaches extends Teaches
       from one Lecturer to 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`:

```
deirdre.SlowlyTeaches += 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 `+=` 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 relation-

ships. 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.

Related Work

Modelling languages like UML [7] and ER Diagrams [4] provide associations and relationships as core abstractions. Several database systems, for example object databases adhering to the ODMG standard [10], also provide relationships as primitive. Unfortunately the language access to such primitives is compromised by the lack of first-class support in the language, and so is limited to weak API access.

As we mentioned earlier, Rumbaugh [11] 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 no formal treatment for this or the language as a whole was provided.

Noble has presented some patterns for programming with relationships [8]. In fact, many of these patterns could be used in translating RelJ programs in 'pure' Java. Noble and Grundy also suggested that relationships should be made explicit in object-oriented programs [9]. Again neither works provide any concrete details of language support for relationships.

After completing this work we discovered the paper by Albano, Ghelli and Orsini [1], which specifies a language for use in an object-oriented database environment. They offer many of the same constructions as well as a richer set of available constraints, but build on a data model that is much richer than that of Java and similar languages. They give no formal description of the language. A more detailed comparison between our approaches is future work.

Interest in relationships is not restricted to modelling and programming languages. In the timeframe of the next generation of Microsoft Windows, code-named '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.

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 be currently 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 extensions to the JVM. The design and evaluation of such extensions is interesting future work. In the short term, one can systematically 'compile' RelJ programs into 'pure' Java. In future work we plan to specify such a compilation formally, and consider techniques for verfying the correctness of such a compilation.

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, Alan Mycroft, Matthew Parkinson, Andrew Pitts and Peter Sewell. We would also like to thank the anonymous referees for their efforts.

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, 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.

[4] P. P.-S. Chen. The entity-relationship model – toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[5] S. Drossopoulou, T. Valkevych, and S. Eisenbach. Java type soundness revisited, September 2000.

[6] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proceedings of POPL*, pages 171–183, 1998.

[7] I. Jacobson, G. Booch, and J. Rumbaugh. *The unified software development process*. Addison-Wesley, 1999.

[8] J. Noble. Basic relationship patterns. In *Proceedings of EuroPLOP*, 1997.

[9] J. Noble and J. Grundy. Explicit relationships in object-oriented development. In *Proceedings of TOOLS*, 1995.

[10] R.G.G. Cattell et al. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann, 2000.

[11] J. Rumbaugh. Relations as semantic constructs in an object-oriented language. In *Proceedings of OOPSLA*, pages 466–481, 1987.

[12] J. Smith and D. Smith. Database abstractions: Aggregation and generalizations. *ACM Transactions on Database Systems*, 2(2):105–133, 1977.

[13] P. Stevens and R. Pooley. *Using UML: software engineering with objects and components*. Addison-Wesley, 1999.

[14] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Proceedings of OOPSLA*, pages 227–242. ACM Press, 1987.

[15] 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 Section 3, the following rules populate the subtyping relation with the immediate supertypes provided by the language syntax, and give the reflexive, transitive closure:

$$\frac{}{\vdash t \leq t}\ (\textsc{StRef}) \qquad \frac{\vdash t_1 \leq t_2 \quad \vdash t_2 \leq t_3}{\vdash t_1 \leq t_3}\ (\textsc{StTrans}) \qquad \frac{\mathcal{C}(c_1) = (c_2, \_, \_)}{\vdash c_1 \leq c_2}\ (\textsc{StClass})$$

$$\frac{\mathcal{R}(r_1) = (r_2, \_, \_, \_)}{\vdash r_1 \leq r_2}\ (\textsc{StRel})$$

Next, we provide rules for typing RelJ expressions.

$$\frac{}{\Gamma \vdash \texttt{true}: \texttt{boolean} \quad \Gamma \vdash \texttt{false}: \texttt{boolean} \quad \Gamma \vdash \texttt{null}: n \quad \Gamma \vdash \texttt{empty}: \texttt{set<}n\texttt{>}}\ (\textsc{TsBoolNull}) \qquad \frac{\Gamma(x) = t}{\Gamma \vdash x : t}\ (\textsc{TsVar})$$

Literal values are typed with rules TSBool and TSNull. 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.

$$\frac{}{\Gamma \vdash \texttt{new } c\texttt{()}: c}\ (\textsc{TsNew})$$

$$\frac{\Gamma \vdash e_1 : n \quad \Gamma \vdash e_2 : n'}{\Gamma \vdash e_1 \texttt{ == } e_2 : \texttt{boolean}}\ (\textsc{TsEq}) \qquad \frac{\Gamma \vdash e : n \quad \mathcal{FD}_n(f) = t}{\Gamma \vdash e\,.\,f : t}\ (\textsc{TsFld})$$

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:

$$\frac{\Gamma \vdash e_1 : n_1 \quad \Gamma(x) = \texttt{set<}n_2\texttt{>} \quad \vdash n_1 \leq n_2}{\Gamma \vdash x \texttt{ += } e_1 : n_1}\ (\textsc{TsVarAdd}) \qquad \frac{\Gamma \vdash e_1 : n_1 \quad \Gamma(x) = \texttt{set<}n_2\texttt{>} \quad \vdash n_1 \leq n_2}{\Gamma \vdash x \texttt{ -= } e_1 : n_1}\ (\textsc{TsVarSub})$$

$$\frac{\Gamma \vdash e_1 : n_1 \quad \Gamma \vdash e_2 : n_2 \quad \mathcal{FD}_{n_1}(f) = \texttt{set<}n_3\texttt{>} \quad \vdash n_2 \leq n_3}{\Gamma \vdash e_1\,.\,f \texttt{ += } e_2 : n_2}\ (\textsc{TsFldAdd}) \qquad \frac{\Gamma \vdash e_1 : n_1 \quad \Gamma \vdash e_2 : n_2 \quad \mathcal{FD}_{n_1}(f) = \texttt{set<}n_3\texttt{>} \quad \vdash n_2 \leq n_3}{\Gamma \vdash e_1\,.\,f \texttt{ -= } e_2 : n_2}\ (\textsc{TsFldSub})$$

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: unlike the use of `+=`/`-=` on relationships, no new instances are created.

$$\frac{\Gamma \vdash x : t_1 \quad \Gamma \vdash e : t_2 \quad \vdash t_2 \leq t_1}{\Gamma \vdash x \texttt{ = } e : t_2}\ x \neq \texttt{this}\ (\textsc{TsAss}) \qquad \frac{\Gamma \vdash e_1 : n_1 \quad \Gamma \vdash e_2 : t_1 \quad \mathcal{FD}_{n_1}(f) = t_2 \quad \vdash t_1 \leq t_2}{\Gamma \vdash e_1\,.\,f \texttt{ = } e_2 : t_1}\ (\textsc{TsFldAss})$$

Variables and fields may be assigned, but note that no such rule exists for relationships. As usual, values of type subordinate to a field's type may be assigned to such a field.

$$\frac{\Gamma \vdash e_1 : n_1 \quad \Gamma \vdash e_2 : t_1 \quad \mathcal{M}_{n_1}(m) = (x, \mathcal{L}, t_2, t_3, \_) \quad \vdash t_1 \leq t_2}{\Gamma \vdash e_1\,.\,m(e_2) : t_3}\ (\textsc{TsCall})$$

Method call is typed directly from the method look-up table.

The `for` statement was typed in the body of the paper. The 'skip' statement is always well-typed. The conditional's typing-checking is standard, recalling that we do not assign types to statements or statement sequences.

$$\frac{}{\Gamma \vdash ;}\ (\textsc{TsSkip}) \qquad \frac{\Gamma \vdash e : \texttt{boolean} \quad \Gamma \vdash \overline{s_1} \quad \Gamma \vdash \overline{s_2}}{\Gamma \vdash \texttt{if (}e\texttt{) \{}\overline{s_1}\texttt{\} else \{}\overline{s_2}\texttt{\};}}\ (\textsc{TsCond})$$

We now provide types for statement sequences in the obvious way, where ϵ denotes the empty sequence (usually omitted):

$$\frac{}{\Gamma \vdash \epsilon}\ (\text{TSSEQ1}) \qquad \frac{\Gamma \vdash s_1 \quad \Gamma \vdash \overline{s_2}}{\Gamma \vdash s_1 \overline{s_2}}\ (\text{TSSEQ2})$$

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:

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

A.2 Operational Semantics

First, we give full definitions of 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.

$$\mathsf{new}_P(c) = \begin{cases} \langle\!\langle \texttt{Object} \| \rangle\!\rangle & \text{if } c = \texttt{Object} \\ \langle\!\langle c \| f_1 : v_1,\ f_2 : v_2,\ \ldots,\ f_i : v_i \rangle\!\rangle & \text{otherwise} \end{cases}$$
where
$\{f_1,\ f_2, \ldots,\ f_i\} = \mathsf{dom}(\mathcal{FD}_{P,c})$
$v_i = \mathsf{initial}_P(\mathcal{FD}_{P,c}(f_i))$

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

$$\mathsf{dynType}(o) = \begin{cases} c & \text{if } o = \langle\!\langle c \| \ldots \rangle\!\rangle \\ r & \text{if } o = \langle\!\langle r, \_, \_, \_ \| \ldots \rangle\!\rangle \\ \bot & \text{otherwise} \end{cases}$$

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

We then give the remaining rules of the operational semantics, covering the routine aspects of the Java-like core, as well as rules for raising null-pointer errors:

(OSEMPTY) $\langle \Gamma, \sigma, \rho, \lambda, \texttt{empty} \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \emptyset \rangle$

(OSVAR) $\langle \Gamma, \sigma, \rho, \lambda, x \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \lambda(x) \rangle$

(OSFLDN) $\langle \Gamma, \sigma, \rho, \lambda, \texttt{null}.\ f \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \mathsf{NullPtrError} \rangle$

(OSFLD) $\langle \Gamma, \sigma, \rho, \lambda, \iota.\ f \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \mathsf{fld}(\sigma, \iota,\ f) \rangle$

(OSRELINSTN) $\langle \Gamma, \sigma, \rho, \lambda, \texttt{null:}r \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \mathsf{NullPtrError} \rangle$

(OSEQ) $\langle \Gamma, \sigma, \rho, \lambda, u\ \texttt{==}\ u \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \texttt{true} \rangle$

(OSNEQ) $\langle \Gamma, \sigma, \rho, \lambda, u\ \texttt{==}\ u' \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \texttt{false} \rangle$
where
$u \neq u'$

(OSNEW) $\langle \Gamma, \sigma, \rho, \lambda, \texttt{new}\ c\texttt{()} \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma[\iota \mapsto \mathsf{new}_P(c)], \rho, \lambda, \iota \rangle$
where
$\iota \notin \mathsf{dom}(\sigma)$

(OSBODY) $\langle \Gamma, \sigma, \rho, \lambda, \{\ \texttt{return}\ u;\ \} \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, u \rangle$

(OSASS) $\langle \Gamma, \sigma, \rho, \lambda, x\ \texttt{=}\ u \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda[x \mapsto u], u \rangle$

(OSADD) $\langle \Gamma, \sigma, \rho, \lambda, x\ \texttt{+=}\ u \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda[x \mapsto \lambda(x) \cup u], u \rangle$

(OSSUB) $\langle \Gamma, \sigma, \rho, \lambda, x\ \texttt{-=}\ u \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda[x \mapsto \lambda(x) \setminus u], u \rangle$

(OSFLDASSN) $\langle \Gamma, \sigma, \rho, \lambda, \texttt{null}.\ f\ \texttt{=}\ u \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \mathsf{NullPtrError} \rangle$

(OSFLDADDN) $\langle \Gamma, \sigma, \rho, \lambda, \texttt{null}.\ f\ \texttt{+=}\ u \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \mathsf{NullPtrError} \rangle$

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

(OSFLDSUBN) $\langle \Gamma, \sigma, \rho, \lambda, \texttt{null}.\ f\ \texttt{-=}\ u \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \mathsf{NullPtrError} \rangle$

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

(OSRELADDN) $\langle \Gamma, \sigma, \rho, \lambda, \iota_1^{\texttt{null}}.r\ \texttt{+=}\ \iota_2^{\texttt{null}} \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \mathsf{NullPtrError} \rangle$
where
$\iota_1^{\texttt{null}} = \texttt{null}$ or $\iota_2^{\texttt{null}} = \texttt{null}$

(OSRELSUBN) $\langle \Gamma, \sigma, \rho, \lambda, \iota_1^{\texttt{null}}.r\ \texttt{-=}\ \iota_2^{\texttt{null}} \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \mathsf{NullPtrError} \rangle$
where
$\iota_1^{\texttt{null}} = \texttt{null}$ or $\iota_2^{\texttt{null}} = \texttt{null}$

(OSCALLN) $\langle \Gamma, \sigma, \rho, \lambda, \texttt{null.}m\texttt{(}u\texttt{)} \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \mathsf{NullPtrError} \rangle$

(OSSTAT) $\langle \Gamma, \sigma, \rho, \lambda, u\texttt{;} \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \texttt{;} \rangle$

(OSCONDT) $\langle \Gamma, \sigma, \rho, \lambda, \texttt{if (true) \{}\overline{s_1}\texttt{\} else \{}\overline{s_2}\texttt{\};} \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \overline{s_1} \rangle$

(OSCONDF) $\langle \Gamma, \sigma, \rho, \lambda, \texttt{if (false) \{}\overline{s_1}\texttt{\} else \{}\overline{s_2}\texttt{\};} \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \overline{s_2} \rangle$