Technical Report

UCAM-CL-TR-642
ISSN 1476-2986

Number 642

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

First-class relationships in an object-oriented language

Gavin Bierman, Alisdair Wren

August 2005

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

ISSN 1476-2986

First-class relationships in an object-oriented language*

Gavin Bierman
Microsoft Research, Cambridge
gmb@microsoft.com

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

August 5, 2005

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.

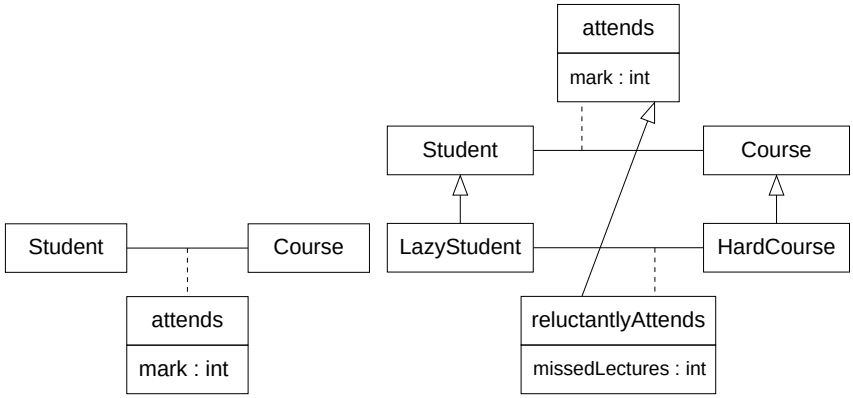
However, whilst object-oriented languages easily represent real-world entities (e.g. students, lectures, buildings), the programmer is poorly served when trying to represent the many natural *relationships* between those entities (e.g. 'attends lecture', 'is taught in').

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.

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

*This is an extended version of a paper presented at ECOOP 2005



(a) Association Class (b) Parallel Hierarchy

Figure 1: Relationships represented as UML *association classes*

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 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

```
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
```

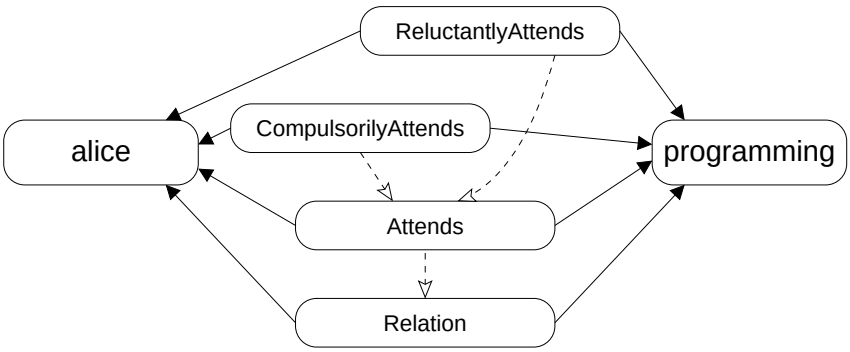


Figure 2: Example RelJ code and possible instantiation

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* (value) type, much like the generic literal types used by the ODMG [12].[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 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 (which does not include `from` or `to`); 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.

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 (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: $e.r$ 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

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

$$
\begin{array}{rll}
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 \qquad\qquad \text{field access} \\
e \in \mathsf{Expression} & ::= & v \mid \qquad\qquad \text{value} \\
 & & l \mid \qquad\qquad \text{l-value} \\
 & & e_1\ \texttt{==}\ e_2 \mid \qquad\qquad \text{equality test} \\
 & & e_1\ \texttt{+}\ e_2 \mid e_1\ \texttt{-}\ e_2 \mid \qquad\qquad \text{set addition/removal} \\
 & & e\texttt{.}r \mid e\texttt{:}r \mid \qquad\qquad \text{relationship access} \\
 & & e\texttt{.from} \mid \qquad\qquad \text{relationship source} \\
 & & e\texttt{.to} \mid \qquad\qquad \text{relationship destination} \\
 & & se \qquad\qquad \text{statement expression} \\
se \in \mathsf{StatementExp} & ::= & \texttt{new}\ c\texttt{()} \mid \qquad\qquad \text{instantiation} \\
 & & l\ \texttt{=}\ e \mid \qquad\qquad \text{assignment} \\
 & & r\texttt{.add(}e\texttt{,}e'\texttt{)} \mid r\texttt{.rem(}e\texttt{,}e'\texttt{)} \mid \qquad\qquad \text{relationship addition/removal} \\
 & & e\texttt{.}m\texttt{(}e'\texttt{)} \qquad\qquad \text{method call} \\
s \in \mathsf{Statement} & ::= & \epsilon \mid \qquad\qquad \text{empty statement} \\
 & & se\texttt{;}\ s_1 \mid \qquad\qquad \text{expression} \\
 & & \texttt{if (}e\texttt{) \{}s_1\texttt{\} else \{}s_2\texttt{\};}\ s_3 \mid \qquad\qquad \text{conditional} \\
 & & \texttt{for (}n\ x\ \texttt{:}\ e\texttt{) \{}s_1\texttt{\};}\ s_2 \qquad\qquad \text{set iteration} \\
R \in \mathsf{Term} & ::= & s \mid e \qquad\qquad \mathsf{RelJ}\ \text{terms}
\end{array}
$$

Figure 3: The grammar of RelJ types and programs

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

Figure 4: Signatures of class and relationship tables

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:

$$\begin{aligned} \mathcal{C}_P(\texttt{Object}) &= (\texttt{Object}, \emptyset, \emptyset) \\ \mathcal{R}_P(\texttt{Relation}) &= (\texttt{Relation}, \texttt{Object}, \texttt{Object}, \emptyset, \emptyset) \end{aligned}$$

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:

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

STCov makes set types covariant with their contained type. If `set<->` were a reference type, then this kind of covariance would be unsound. However, `set<->` 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.[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:

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

$\mathcal{MD}$ 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, $\Gamma \in \mathsf{TypeEnv}$, which assigns types to variable names. Selected typing judgements for RelJ expressions are given below:

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

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 $e\texttt{:}r$. 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.

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

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

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.

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

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

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:

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

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:

$$\frac{\begin{array}{ll} & \mathcal{C}(n) = (n', \_, \_) \ \vee\ \mathcal{R}(n) = (n', \_, \_, \_, \_) \\ 1. & f \notin \mathsf{dom}(\mathcal{FD}_{n'}) \\ 2. & \mathcal{F}_n(f) \in \mathsf{validTypes}_P \\ 3. & \mathcal{R}(f) = (\_, n_1, n_2, \_) \Rightarrow\ \nexists\, n \leq n_1 \end{array}}{P, n \vdash f}\ (\textsc{WTField})$$

WTField 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.

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

WTMethod 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:

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

WTClass 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. WTRelationship imposes many of the same restrictions as WTClass, with the addition of conditions 1–3, which check the types related by r's super-relationship are supertypes of those that r relates.

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:

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

We only consider programs that are well-formed with respect to the above rule.

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:

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

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, $\langle\!\langle r, \iota^{\mathtt{null}}, \iota_1, \iota_2 \| f_1 : v_1, \ldots, f_i : v_i \rangle\!\rangle$, 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*, $\iota^{\mathtt{null}}$; specifically, the instance of r's super-relationship which relates the same object addresses ι_1 and ι_2. Where $r = \mathtt{Relation}$, 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:

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

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 ($\sigma[\iota \mapsto o]$) 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 $\mathcal{E}_{\mathrm{e}}[e]$. 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 sound-

$\mathcal{E}_\mathrm{e} \in \mathsf{ExpContext} ::=$

$\bullet$	hole
$\mid \mathcal{E}_\mathrm{e}\,.\,f$	field lookup
$\mid \mathcal{E}_\mathrm{e}$ `==` $e \mid u$ `==` $\mathcal{E}_\mathrm{e}$	equality test
$\mid \mathcal{E}_\mathrm{e}$ `+` $e \mid u$ `+` $\mathcal{E}_\mathrm{e}$	set addition
$\mid \mathcal{E}_\mathrm{e}$ `-` $e \mid u$ `-` $\mathcal{E}_\mathrm{e}$	set removal
$\mid \mathcal{E}_\mathrm{e}\,.\,r \mid \mathcal{E}_\mathrm{e}\texttt{:}r$	relationship access
$\mid \mathcal{E}_\mathrm{e}$`.from` $\mid \mathcal{E}_\mathrm{e}$`.to`	relationship from/to
$\mid$ `{` $\mathcal{E}$ `return` e`;` `}` $\mid$ `{ return` $\mathcal{E}_\mathrm{e}$`;` `}`	method body
$\mid \mathcal{E}_\mathrm{e}\,.\,f$ `=` $e \mid x$ `=` $\mathcal{E}_\mathrm{e} \mid u.f$ `=` $\mathcal{E}_\mathrm{e}$	assignment
$\mid \mathcal{E}_\mathrm{e}\,.\,m(e') \mid u\,.\,m(\mathcal{E}_\mathrm{e})$	method call
$\mid r$`.add(`$\mathcal{E}_\mathrm{e}$`,`e'`)` $\mid r$`.add(`u`,`$\mathcal{E}_\mathrm{e}$`)`	relationship addition
$\mid r$`.rem(`$\mathcal{E}_\mathrm{e}$`,`e'`)` $\mid r$`.rem(`u`,`$\mathcal{E}_\mathrm{e}$`)`	relationship removal

$\mathcal{E}_\mathrm{s} \in \mathsf{StatContext} ::=$

$\mathcal{E}_\mathrm{e}$`;` s	expression
$\mid$ `for (`$n\ x$ `:` $\mathcal{E}_\mathrm{e}$`) {`s_1`};` s_2	set iteration
$\mid$ `if (`$\mathcal{E}_\mathrm{e}$`) {`s_1`} else {`s_2`};` s_3	conditional

Figure 5: Grammar for evaluation contexts

ness.

$$
\begin{aligned}
R \in \mathsf{DynTerm} &= e \mid s \\
C \in \mathsf{Config} &= \mathsf{TypeEnv} \times \mathsf{Heap} \times \mathsf{RelHeap} \times \mathsf{Locals} \times \mathsf{DynTerm} \\
C_\mathrm{E} \in \mathsf{ErrorConfig} &= \mathsf{TypeEnv} \times \mathsf{Heap} \times \mathsf{RelHeap} \times \mathsf{Locals} \times \mathsf{Error} \\
\overset{P}{\rightsquigarrow} &\subset \mathsf{Config} \times (\mathsf{Config} \cup \mathsf{ErrorConfig})
\end{aligned}
$$

Expression execution proceeds when a sub-expression in hole position may be reduced, as specified by OSContextE, and similarly for statements in OSContextS:

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

$$
(\textsc{OSContextS})\ \frac{\langle \Gamma, \sigma, \rho, \lambda, e\rangle \overset{P}{\rightsquigarrow} \langle \Gamma', \sigma', \rho', \lambda', e'\rangle}{\langle \Gamma, \sigma, \rho, \lambda, \mathcal{E}_\mathrm{s}[e]\rangle \overset{P}{\rightsquigarrow} \langle \Gamma', \sigma', \rho', \lambda', \mathcal{E}_\mathrm{s}[e']\rangle}
$$

We also execute statements inside partially-executed method bodies:

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

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 $e\texttt{:}r$ so that relationship attributes may be read or modified:

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

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

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

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.

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

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

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

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

Figure 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.

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

OSTO: $\langle \Gamma, \sigma, \rho, \lambda, \iota\texttt{.to} \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \iota' \rangle$ 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: $\langle \Gamma, \sigma_1, \rho_1, \lambda, r\texttt{.add(}\iota_1\texttt{,}\iota_2\texttt{)} \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma_2, \rho_2, \lambda, \iota_3 \rangle$
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: $\langle \Gamma_1, \sigma, \rho, \lambda_1, \iota.m(u)\rangle \overset{P}{\rightsquigarrow} \langle \Gamma_2, \sigma, \rho, \lambda_2, \{\ s_2\ \texttt{return}\ e_2;\ \}\rangle$

where

$\sigma(\iota) = \langle\!\langle n \| \ldots \rangle\!\rangle$ or $\sigma(\iota) = \langle\!\langle n, \_, \_, \_ \| \ldots \rangle\!\rangle$

$\mathcal{MD}_{P,n}(m) = (x, \mathcal{L}, t_1, \_, s_1\ \texttt{return}\ e_1;)$

$\mathsf{dom}(\mathcal{L}) = \{x_1, \ldots, x_i\}$

$x', x'_{\texttt{this}}, x'_1, \ldots, x'_i \notin \mathsf{dom}(\lambda_1)$

$\Gamma_2 = \Gamma_1[x' \mapsto t_1][x'_{\texttt{this}} \mapsto n][x'_{1..i} \mapsto \mathcal{L}(x_{1..i})]$

$\lambda_2 = \lambda_1[x' \mapsto u][x'_{\texttt{this}} \mapsto \iota][x'_{1..i} \mapsto \mathsf{initial}(\Gamma_2(x'_{1..i}))]$

$s_2 = s_1[x'/x][x'_{1..i}/x'_{1..i}][x'_{\texttt{this}}/\texttt{this}]$

$e_2 = e_1[x'/x][x'_{1..i}/x_{1..i}][x'_{\texttt{this}}/\texttt{this}]$

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

OSFor2: $\langle \Gamma_1, \sigma, \rho, \lambda_1, \texttt{for (}n\ x : u\texttt{) } \{s_1\}\texttt{; } s_2\rangle \overset{P}{\rightsquigarrow} \langle \Gamma_2, \sigma, \rho, \lambda_2, s_3\ \texttt{for (}n\ x : (u \setminus \iota)\texttt{) } \{s_1\}\texttt{; } s_2\rangle$

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 define a dynamic typing relation, which is identical to the typing relation presented in Sect. 3 but for the addition of the store, σ, so that values may be typed—particularly important for showing subject reduction. To specify this new relation, all typing rules with names starting TS are thus modified to yield a new set of rules with names starting DT (for example, TSFld becomes DTFld with the addition of σ). We complete the specification with the addition of two rules for typing run-time values.

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:

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

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

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

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:

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

$$\text{(WFObject2)}\ \frac{\begin{array}{c}\{f_1, \ldots, f_i\} = \mathsf{dom}(\mathcal{FD}_{P,c}) \\ \forall j \in 1..i : P, \sigma, o \vdash f_j \diamond_{\mathsf{fld}}\end{array}}{P, \sigma \vdash \langle\!\langle c \| f_1 : v_1, \ldots, f_i : v_i \rangle\!\rangle \diamond_{\mathsf{inst}}}$$

$$\text{(WFRelInst2)}\ \frac{\begin{array}{c}\mathcal{R}_P(r) = (\mathsf{dynType}(\sigma(\iota)), n_1, n_2, \mathcal{F}, \_) \\ \{f_1, \ldots, f_i\} = \mathsf{dom}(\mathcal{F}) \\ \forall j \in 1..i : P, \sigma, o \vdash f_j \diamond_{\mathsf{fld}} \\ \vdash \mathsf{dynType}(\sigma(\iota_1)) \leq n_1 \\ \vdash \mathsf{dynType}(\sigma(\iota_2)) \leq n_2\end{array}}{P, \sigma \vdash \langle\!\langle r, \iota, \iota_1, \iota_2 \| f_1 : v_1, \ldots, f_i : v_i \rangle\!\rangle \diamond_{\mathsf{inst}}}$$

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:

$$\text{(WFHeap)}\ \frac{\forall \iota \in \mathsf{dom}(\sigma) : P, \sigma \vdash \sigma(\iota) \diamond_{\mathsf{inst}}}{P \vdash \sigma \diamond_{\mathsf{heap}}} \qquad \text{(WFRelHeap)}\ \frac{\forall (r, \iota_1, \iota_2) \in \mathsf{dom}(\rho) : P, \sigma, \rho \vdash (r, \iota_1, \iota_2) \diamond_{\mathsf{rel}}}{P, \sigma \vdash \rho \diamond_{\mathsf{relheap}}}$$

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

We consider a configuration $\langle \Gamma, \sigma, \rho, \lambda, s \rangle$ to be well-formed when σ, ρ and λ are well-formed, and where s is type-correct. Error configurations, $\langle \Gamma, \sigma, \rho, \lambda, w \rangle$, 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 (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 (Subject Reduction). *In a well-typed program, P, where $\langle \Gamma_1, \sigma_1, \rho_1, \lambda_1, R \rangle$ executes to a new configuration $\langle \Gamma_2, \sigma_2, \rho_2, \lambda_2, R \rangle$, 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 (Progress). *For all well-typed programs, P, all well-formed configurations $\langle \Gamma_1, \sigma_1, \rho_1, \lambda_1, R \rangle$ execute to either:*

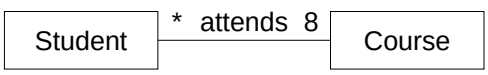
i. an error configuration $\langle \Gamma_2, \sigma_2, \rho_2, \lambda_2, w \rangle$, or

ii. a new configuration $\langle \Gamma_2, \sigma_2, \rho_2, \lambda_2, R \rangle$

By the theorems above, 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, 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.

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.

$$\frac{}{\Gamma \vdash \texttt{true}: \texttt{boolean}}\ (\textsc{TSBoolT}) \qquad \frac{}{\Gamma \vdash \texttt{false}: \texttt{boolean}}\ (\textsc{TSBoolF}) \qquad \frac{n \in \mathsf{validTypes}_P}{\Gamma \vdash \texttt{null}: n}\ (\textsc{TSNull}) \qquad \frac{n \in \mathsf{validTypes}_P}{\Gamma \vdash \texttt{empty}: \texttt{set<}n\texttt{>}}\ (\textsc{TSEmpty})$$

$$\frac{\Gamma(x) = t}{\Gamma \vdash x: t}\ (\textsc{TSVar}) \qquad \frac{c \in \mathsf{validTypes}_P}{\Gamma \vdash \texttt{new } c\texttt{()}: c}\ (\textsc{TSNew}) \qquad \frac{\Gamma \vdash e_1: n \quad \Gamma \vdash e_2: n'}{\Gamma \vdash e_1 \texttt{ == } e_2: \texttt{boolean}}\ (\textsc{TSEq}) \qquad \frac{\Gamma \vdash e: n \quad \mathcal{FD}_n(f) = t}{\Gamma \vdash e.f: t}\ (\textsc{TSFld})$$

$$\frac{\Gamma \vdash e_1: \texttt{set<}n_1\texttt{>} \quad \Gamma \vdash e_2: n_2 \quad \vdash n_1 \leq n_3 \quad \vdash n_2 \leq n_3}{\Gamma \vdash e_1 \texttt{ + } e_2: \texttt{set<}n_3\texttt{>}}\ (\textsc{TSAdd})$$

$$\frac{\Gamma \vdash e_1: \texttt{set<}n_1\texttt{>} \quad \Gamma \vdash e_2: n_2 \quad \vdash n_1 \leq n_3 \quad \vdash n_2 \leq n_3}{\Gamma \vdash e_1 \texttt{ - } e_2: \texttt{set<}n_3\texttt{>}}\ (\textsc{TSSub})$$

$$\frac{x \neq \texttt{this} \quad \Gamma \vdash x: t_1 \quad \Gamma \vdash e: t_2 \quad \vdash t_2 \leq t_1}{\Gamma \vdash x \texttt{ = } e: t_2}\ (\textsc{TSAss})$$

$$\frac{\Gamma \vdash e_1: n \quad \Gamma \vdash e_2: t_1 \quad \mathcal{FD}_n(f) = t_2 \quad \vdash t_1 \leq t_2}{\Gamma \vdash e_1.f \texttt{ = } e_2: t_1}\ (\textsc{TSFldAss})$$

$$\frac{\Gamma \vdash e_1: n \quad \Gamma \vdash e_2: t_1 \quad \mathcal{MD}_n(m) = (x, \mathcal{L}, t_2, t_3, \_) \quad \vdash t_1 \leq t_2}{\Gamma \vdash e_1.m\texttt{(}e_2\texttt{)}: t_3}\ (\textsc{TSCall})$$

$$\frac{\Gamma \vdash e: \texttt{boolean} \quad \Gamma \vdash s_1 \quad \Gamma \vdash s_2 \quad \Gamma \vdash s_3}{\Gamma \vdash \texttt{if (}e\texttt{) \{}s_1\texttt{\} else \{}s_2\texttt{\}; } s_3}\ (\textsc{TSCond})$$

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

Figure 7: The remaining type rules of RelJ

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:

$$\frac{t \in \mathsf{validTypes}_P}{P \vdash t \leq t}\ (\textsc{STRef}) \qquad \frac{P \vdash t_1 \leq t_2 \quad P \vdash t_2 \leq t_3}{P \vdash t_1 \leq t_3}\ (\textsc{STTrans}) \qquad \frac{\mathcal{C}_P(c_1) = (c_2, \_, \_)}{P \vdash c_1 \leq c_2}\ (\textsc{STClass}) \qquad \frac{\mathcal{R}_P(r_1) = (r_2, \_, \_, \_, \_)}{P \vdash r_1 \leq r_2}\ (\textsc{STRel})$$

We specify that the following trivial types are valid types in all programs, P:

$$\frac{}{P \vdash \texttt{boolean}}\ (\textsc{WTBool}) \qquad \frac{}{P \vdash \texttt{Object}}\ (\textsc{WTObject}) \qquad \frac{}{P \vdash \texttt{Relation}}\ (\textsc{WTRelation}) \qquad \frac{P \vdash n}{P \vdash \texttt{set<}n\texttt{>}}\ (\textsc{WTSet})$$

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

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 `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.

A.2 Operational Semantics

First, we give a full definition of substitution:

Definition 1 (Map Substitution). *Substitution on a map for function f, written $f[a \mapsto b]$, updates f such that $f(a) = b$, and that f remains the same everywhere else:*

i. $f[a \mapsto b](a) = b$

ii. $a \neq c \Rightarrow f[a \mapsto b](c) = f(c)$

Next we specify new, which returns an initialised class instance; $\mathsf{initial}$, which returns an appropriate initial value for a variable of type t; $\mathsf{dynType}$, 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: $\langle \Gamma, \sigma, \rho, \lambda, \texttt{empty} \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \emptyset \rangle$

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

OSFldN: $\langle \Gamma, \sigma, \rho, \lambda, \texttt{null.}f \rangle \langle \Gamma, \sigma, \rho, \lambda, \mathsf{NullPtrError} \rangle$

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

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

OSAdd: $\langle \Gamma, \sigma, \rho, \lambda, u \texttt{ + } \iota \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, u \cup \{\iota\} \rangle$

OSAddN: $\langle \Gamma, \sigma, \rho, \lambda, u \texttt{ + null} \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \mathsf{NullPtrError} \rangle$

OSSub: $\langle \Gamma, \sigma, \rho, \lambda, u \texttt{ - } \iota \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, u \setminus \{\iota\} \rangle$

OSSubN: $\langle \Gamma, \sigma, \rho, \lambda, u \texttt{ - null} \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \mathsf{NullPtrError} \rangle$

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

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

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

OSRelRemN: $\langle \Gamma, \sigma, \rho, \lambda, r\texttt{.rem(}\iota_1^{\texttt{null}}\texttt{,}\iota_2^{\texttt{null}}\texttt{)} \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, \mathsf{NullPtrError} \rangle$
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: $\langle \Gamma, \sigma, \rho, \lambda, u\texttt{; } s \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, s \rangle$

OSCondT: $\langle \Gamma, \sigma, \rho, \lambda, \texttt{if (true) } \{s_1\} \texttt{ else } \{s_2\}\texttt{; } s_3 \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, s_1\ s_3 \rangle$

OSCondF: $\langle \Gamma, \sigma, \rho, \lambda, \texttt{if (false) } \{s_1\} \texttt{ else } \{s_2\}\texttt{; } s_3 \rangle \overset{P}{\rightsquigarrow} \langle \Gamma, \sigma, \rho, \lambda, s_2\ s_3 \rangle$

B Proofs

Lemma 1 (Freshness and substitution). *Let $y' = y[a \mapsto b]$, $a \notin \mathit{dom}(a)$, then for any $c \in \mathit{dom}(y)$, $y'(c) = y(c)$.*

Proof As $a \in \mathsf{dom}(y)$ and $c \notin \mathsf{dom}(y)$, then $a \neq c$. By Definition 1, then, $y'(c) = y(c)$. □

Lemma 2. *$\mathit{validTypes}_P$ precisely specifies the well-formed types of a program P:*

$$t \in \mathit{validTypes}_P \Leftrightarrow P \vdash t$$

Proof Recall that we only consider well-formed P with respect to WFProgram.

(⇒) If $t \in \mathsf{validTypes}_P$ then either:

- $t = \texttt{boolean}$, in which case $P \vdash t$ by WTBool
- $t \in \mathsf{dom}(\mathcal{C}_P)$, in which case $P \vdash t$ by WTProgram
- $t \in \mathsf{dom}(\mathcal{R}_P)$, in which case $P \vdash t$ by WTProgram
- $t = \texttt{set<}n\texttt{>}$ where $n \in \mathsf{dom}(\mathcal{C}_P) \cup \mathsf{dom}(\mathcal{R}_P)$, in which case $P \vdash t$ by WTSet and the argument above.

(⇐) If $P \vdash t$ then either:

- the judgement arises by WTBool, in which case $t = \texttt{boolean} \in \mathsf{validTypes}_P$,
- the judgement arises by WTClass, in which case $t \in \mathsf{dom}(\mathcal{C}_P) \subset \mathsf{validTypes}_P$.
- the judgement arises by WTRelationship, in which case $t \in \mathsf{dom}(\mathcal{R}_P) \subset \mathsf{validTypes}_P$.

– the judgement arises by WTSet, in which case $t = \texttt{set<}n\texttt{>}$ such that $P \vdash n$. It must be the case that $n \in \mathsf{dom}(\mathcal{C}_P) \cup \mathsf{dom}(\mathcal{R}_P)$ by argument above, so $t \in \mathsf{validTypes}_P$ as required.

□

Lemma 3 (Subtyping relationship properties). *The subtyping relationship of a well-formed program P, $P \vdash - \leq -$, forms a set partial orders with a maximum elements* `Object` *and* `set<Object>` *respectively.*

(a) $P \vdash - \leq -$ *is a partial order.*

(b) `Object` *is a maximal element:* $P \vdash \texttt{Object} \leq t \Rightarrow t = \texttt{Object}$.

(c) `Relation` *is the maximal element of* RelName *under subtyping:*
$P \vdash \texttt{Relation} \leq r \Rightarrow r = \texttt{Relation}$.

(d) `set<Object>` *is a maximal element:*
$P \vdash \texttt{set<Object>} \leq t \Rightarrow t = \texttt{set<Object>}$.

(e) If $\not\vdash r' \leq r$ *and* $\vdash r' \leq r''$ *then* $\not\vdash r'' \leq r$.

Furthermore, when $P \vdash t \leq t'$*:*

(f) $t, t' \in \mathsf{validTypes}_P$.

*(g) Named types (*ClassName $\cup$ RelName*) are closed under subtyping:*
$t \in \mathsf{ClassName} \cup \mathsf{RelName} \Rightarrow t' \in \mathsf{ClassName} \cup \mathsf{RelName}$ *and* vice versa.

(h) Relationships are downwards-closed under subtyping:
$t' \in \mathsf{RelName} \Rightarrow t \in \mathsf{RelName}$).

(i) Set types are closed under subtyping:
$t = \texttt{set<}n\texttt{>} \Rightarrow t' = \texttt{set<}n'\texttt{>}$ *and* vice versa.

(j) {`boolean`} *is closed under subtyping:*
$t = \texttt{boolean} \Rightarrow t' = \texttt{boolean}$ *and* vice versa.

Proof

(a) $P \vdash - \leq -$ is clearly reflexive, by STRef, and transitive, by STTrans. Class and relationship inheritance is antisymmetric by the acyclicity constraint in WTProgram. The order set up on sets ($\texttt{set<} - \texttt{>}$) is isomorphic to that on n, and is also therefore a partial order.

(b) Suppose that $P \vdash \texttt{Object} \leq t$ and examine its derivation. Clearly STCov, STObject and STRel do not apply. Where the derivation ends in STRef then $t = \texttt{Object}$. Where the derivation ends in STTrans, then there is a t' such that $P \vdash \texttt{Object} \leq t'$ and $P \vdash t' \leq t$; by induction, $t' = \texttt{Object}$ and by induction again $t = t' = \texttt{Object}$. Where the derivation ends in STClass, then note that $\mathcal{C}_P(\texttt{Object}) = (\texttt{Object}, \_, \_)$, so $t = \texttt{Object}$.

(c) By induction on the derivation of $P \vdash \texttt{Relation} \leq r$. Clearly STObject, STClass, STCov do not apply. Where derivation ends in STRef then $r = \texttt{Relation}$. Where derivation ends in STTrans then there is some t'' such that $P \vdash \texttt{Relation} \leq t''$ and $P \vdash t'' \leq r$; but relationships are downwards closed under subtyping by Lemma 3(h) so $t'' \in \mathsf{RelName}$ and by induction $\texttt{Relation} = t'' = r$ as required. Where derivation ends in STRel then $\mathcal{R}_P(\texttt{Relation}) = (\texttt{Relation}, \_, \_, \_, \_)$ so $r = \texttt{Relation}$.

(d) By induction on the derivation of $P \vdash \texttt{set<Object>} \leq t$. Result follows from Lemma 3(b).

(e) Suppose that $\vdash r'' \leq r$. Then, by STTrans and $\vdash r' \leq r''$, $\vdash r' \leq r$. Contradiction.

(f) By induction on derivation of $P \vdash t \leq t'$. Where the derivation ends in STRef then $t \in \mathsf{validTypes}_P$. Where the derivation ends in STTrans then there is some t'' such that $P \vdash t \leq t''$ and $P \vdash t'' \leq t'$; by induction $t, t', t'' \in \mathsf{validTypes}_P$. Where the derivation ends in STClass then $t \in \mathsf{dom}(\mathcal{C}(P)) \in \mathsf{validTypes}_P$ by definition; by WTProgram, $P \vdash t$ so either *(i)* $P \vdash t'$ by WTClass, in which case $t' = \texttt{Object} \in \mathsf{validTypes}_P$ or $t' \in \mathsf{dom}(\mathcal{C}(P)) \subset \mathsf{validTypes}_P$, or *(ii)* $t = \texttt{Object} \in \mathsf{validTypes}_P$ so $t' = \texttt{Object}$ by Lemma 3(b). Where the derivation ends in STRel, the argument proceeds as for STClass and `Relation`. Where the derivation ends in STCov then $t = \texttt{set<}n\texttt{>}$ and $t' = \texttt{set<}n'\texttt{>}$ and $P \vdash n \leq n'$; by induction $n, n' \in \mathsf{validTypes}_P$ so $\texttt{set<}n\texttt{>}, \texttt{set<}n'\texttt{>} \in \mathsf{validTypes}_P$ by definition. Where the derivation ends in STObject, $t = \texttt{Relation} \in \mathsf{validTypes}_P$ and $t' = \texttt{Object} \in \mathsf{validTypes}_P$.

(g) By induction on derivation of $P \vdash t \leq t'$. Where the derivation ends in STRef then $t = t'$. Where the derivation ends in STTrans then there is some t'' such that $P \vdash t \leq t''$ and $P \vdash t'' \leq t'$; by induction $t'' \in \mathsf{NominalType}$ and so by induction $t' \in \mathsf{NominalType}$ (or *vice versa*). Where the derivation ends in STClass then $t = c$ and $t' = c'$ where $c, c' \in \mathsf{ClassName} \subset \mathsf{NominalType}$ as required. Similarly where the derivation ends in STRel. The derivation cannot end in STCov as no n' is such that $\texttt{set<}n'\texttt{>} \in \mathsf{NominalType}$ by definition. Where the derivation ends in STObject, $t = \texttt{Relation} \in \mathsf{NominalType}$ and $t' = \texttt{Object} \in \mathsf{NominalType}$.

(h) By induction on derivation of $P \vdash t \leq r$. Clearly STClass, STCov and STObject do not apply. Where derivation ends in STRef then $t = r$. Where derivation ends in STTrans then there is some t'' where $P \vdash t \leq t''$ and $P \vdash t'' \leq r$; by induction $t'' \in \mathsf{RelName}$, so by induction $t \in \mathsf{RelName}$. Where derivation ends in STRel then $t \in \mathsf{dom}(\mathcal{R}_P)$ so $t \in \mathsf{RelName}$ by definition.

(i) By induction on derivation of $P \vdash t \leq t'$. Clearly STClass, STRel and STObject do not apply. Where derivation ends in STRef then $t = t'$. Where derivation ends in STTrans then result follows by induction. Where derivation ends in STCov, then clearly both types are of the form $\texttt{set<}n\texttt{>}$.

(j) By trivial induction as above.

□

Lemma 4. $P \vdash n \leq n' \Rightarrow \mathcal{FD}_{P,n'} \subseteq \mathcal{FD}_{P,n}$

Proof By induction on the derivation of $P \vdash n \leq n'$.

Case 1: $P \vdash n \leq n'$ arises by STRef
Then $n = n'$ and the result follows trivially.

Case 2: $P \vdash n \leq n'$ arises by STObject
Then $n = \texttt{Relation}$ and $n' = \texttt{Object}$ so $\mathcal{FD}_{P,n'} = \mathcal{FD}_{P,n} = \emptyset$ as required.

Case 3: $P \vdash n \leq n'$ arises by STClass
Then $\mathcal{C}(n) = (n', \_, \_)$. Take some f, t such that $\mathcal{FD}_{P,n'}(f) = t$.

Suppose that $\mathcal{FD}_{P,n}(f) \neq t$. Then by definition of $\mathcal{FD}$, $\mathcal{F}_{P,n}(f) = t'$. But by WTClass, it must be the case that $P, n \vdash f$. By WTField, $f \notin \mathsf{dom}(\mathcal{FD}_{P,n'})$. Contradiction.

Conclude $\mathcal{FD}_{P,n}(f) = t$ as required.

Case 4: $P \vdash n \leq n'$ arises by STRel
As above.

Case 5: $P \vdash n \leq n'$ arises by STTrans
Then $P \vdash n \leq n''$ and $P \vdash n'' \leq n'$. By induction, $\mathcal{FD}_{P,n''} \subseteq \mathcal{FD}_{P,n}$ and $\mathcal{FD}_{P,n'} \subseteq \mathcal{FD}_{P,n''}$. Therefore, $\mathcal{FD}_{P,n'} \subseteq \mathcal{FD}_{P,n}$ as required by transitivity of $\subseteq$.

□

Lemma 5. *Where* $P \vdash n \leq n'$ *and* $\mathcal{MD}_{P,n'}(m) = (\_, t'_1, t'_2, \_, \_)$ *then* $\mathcal{MD}_{P,n}(m) = (\_, t_1, t_2, \_, \_)$ $P \vdash t'_1 \leq t_1$ *and* $P \vdash t_2 \leq t'_2$.

Proof By induction on the derivation of $P \vdash n \leq n'$.

Case 1: $P \vdash n \leq n'$ arises by STRef
Then $n = n'$, $\mathcal{MD}_{P,n} = \mathcal{MD}_{P,n'}$ and the result follows trivially from reflexivity of subtyping.

Case 2: $P \vdash n \leq n'$ arises by STObject
Then $n = \texttt{Relation}$ and $n' = \texttt{Object}$ so $\mathcal{MD}_{P,n} = \mathcal{MD}_{P,n'} = \emptyset$ as required.

Case 3: $P \vdash n \leq n'$ arises by STClass
Then $\mathcal{C}(n) = (n', \_, \_)$.

If m is (re-)declared in n, then $\mathcal{MD}_{P,n}(m) = \mathcal{M}_{P,n}(m) = (\_, t_1, t_2, \_, \_)$, and by WTClass, $P, n \vdash m$. Then, by WTMethod, $P \vdash t'_1 \leq t_1$ and $P \vdash t_2 \leq t'_2$ as required.

If m is not delcared in n, then $m \notin \mathsf{dom}(\mathcal{M}_{P,n})$ so $\mathcal{MD}_{P,n'}(m) = \mathcal{MD}_{P,n}(m)$ and the result follows trivially by reflexivity of subtyping.

Case 4: $P \vdash n \leq n'$ arises by STRel
As above.

Case 5: $P \vdash n \leq n'$ arises by STTrans
Then $P \vdash n \leq n''$ and $P \vdash n'' \leq n'$.

By induction, $\mathcal{MD}_{P,n''}(=)(\_, t''_1, t''_2, \_, \_)$ and $P \vdash t''_1 \leq t_1$, $P \vdash t'_1 \leq t''_1$, $P \vdash t_2 \leq t''_2$ and $P \vdash t''_2 \leq t'_2$. By transitivity of subtyping, $P \vdash t'_1 \leq t_1$ and $P \vdash t_2 \leq t''_2$ as required.

□

Lemma 6. *The types of run-time values in the store are preserved by store extension. If:*
Assumption 1: $P, \Gamma, \sigma \vdash u : t$
Assumption 2: $\sigma' = \sigma[\iota' \mapsto o']$
Assumption 3: $\iota' \notin \mathsf{dom}(\sigma)$
then $P, \Gamma, \sigma' \vdash u : t$.

Proof We proceed by induction on derivation of $P, \Gamma, \sigma \vdash u : t$.

Case 1: Derivation ends DTBoolF/DTBoolT/DTNull
Then the result follows immediately.

Case 2: Derivation ends DTAddr
Then $u = \iota$ and $t = n$.

1: $P \vdash \mathsf{dynType}(\sigma(u)) \leq t$	(Assm. 1, DTAddr)
2: $u \in \mathsf{dom}(\sigma)$	(1)
3: $\sigma(u) = \sigma'(u)$	(Assm. 2, 2, Lemma 1)
4: $\mathsf{dynType}(\sigma(u)) = \mathsf{dynType}(\sigma'(u))$	(3, Defn dynType)
5: $P \vdash \mathsf{dynType}(\sigma'(u)) \leq n$	(1, 4)

6: $P, \Gamma, \sigma' \vdash u : t$ (5, DTAddr)

Case complete.

Case 3: Derivation ends DTSet

Then $u = \{\iota_1, \ldots, \iota_i\}$ and $t = \texttt{set<}n\texttt{>}$.

7: $P \vdash n$ (Assm. 1, DTSet)
8: $\forall \iota \in \mathsf{dom}(u) : P, \emptyset, \sigma \vdash \iota : n$ (Assm. 1, DTSet)
9: $\forall \iota \in \mathsf{dom}(u) : P, \emptyset, \sigma' \vdash \iota : n$ (8, Inductive Hypothesis)
10: $P, \Gamma, \sigma' \vdash u : t$ (7, 9, DTSet)

Therefore, in all cases, $P, \Gamma, \sigma' \vdash u : t$. □

Lemma 7. *The well-formedness of fields in objects in the store is preserved by store extension: If:*

Assumption 1: $P, \sigma, o \vdash f \diamond_{\mathsf{fld}}$
Assumption 2: $\sigma' = \sigma[\iota' \mapsto o']$
Assumption 3: $\iota' \notin \mathsf{dom}(\sigma)$
then $P, \sigma', o \vdash f \diamond_{\mathsf{fld}}$.

Proof The only applicable rule for statement Assm. 1 is WFField:

1: $\mathsf{dynType}(o) = n$ (Assm. 1, WFField)
2: $\mathcal{FD}_{P,n}(f) = t$ (Assm. 1, WFField)
3: $P, \emptyset, \sigma \vdash o(f) : t$ (Assm. 1, WFField)
4: $P, \emptyset, \sigma' \vdash o(f) : t$ (Assm. 2, Assm. 3, 3, Lemma 6)
5: $P, \sigma, o \vdash f \diamond_{\mathsf{fld}}$ (1, 2, 4, WFField)

□

Lemma 8. *The well-formedness of objects in the store is preserved by the addition of a new object. If:*

Assumption 1: $P, \sigma \vdash \sigma(\iota) \diamond_{\mathit{inst}}$
Assumption 2: $\sigma' = \sigma[\iota' \mapsto o']$
Assumption 3: $\iota' \notin \mathsf{dom}(\sigma)$
Then: $P, \sigma' \vdash \sigma'(\iota) \diamond_{\mathit{inst}}$.

Proof Assm. 1 must arise by either WFObject, WFRelInst1 or WFRelInst2, as no other rule is applicable. We therefore proceed by case analysis:

Case 1: Assm. 1 arises from WFObject1

Then $o = \langle\!\langle \texttt{Object} \| \rangle\!\rangle$, and the result follows immediately from WFObject1.

Case 2: Assm. 1 arises from WFObject2

Then $o = \langle\!\langle c \| f_1 : v_1, \ldots, f_n : v_n \rangle\!\rangle$:

1: $\mathsf{dom}(\sigma(\iota)) = \mathsf{dom}(\mathcal{FD}_{P,c})$ (Assm. 1, WFObject2)
2: $\forall f \in \mathsf{dom}(\sigma(\iota)) : P, \sigma, \sigma(\iota) \vdash f \diamond_{\mathsf{fld}}$ (Assm. 1, WFObject2)
3: $\forall f \in \mathsf{dom}(\sigma'(\iota)) : P, \sigma', \sigma'(\iota) \vdash f \diamond_{\mathsf{fld}}$ (Assm. 2, Assm. 3, 2, Lemma 7)
4: $P, \sigma' \vdash \sigma'(\iota) \diamond_{\mathsf{inst}}$ (1, 3, WFObject)

Case 3: Assm. 1 arises from WFRelInst1

Then $o = \langle\!\langle \texttt{Relation}, \texttt{null}, \iota_1, \iota_2 \| \rangle\!\rangle$, and the result follows immediately from WFRelInst1 and that $\iota_1, \iota_2 \in \mathsf{dom}(\sigma) \subseteq \mathsf{dom}(\sigma')$.

Case 4: Assm. 1 arises from WFRelInst2

Then $o = \langle\!\langle r \neq \texttt{Relation}, \iota'', \iota_1, \iota_2 \| f_1 : v_1, \ldots, f_n : v_n \rangle\!\rangle$:

5:	$\mathsf{dom}(\sigma(\iota)) = \mathsf{dom}(\mathcal{F}_{P,c})$	(Assm. 1, WFRelInst2)
6:	$\mathcal{R}_P(r) = (\mathsf{dynType}(\sigma(\iota'')), n_1, n_2, \_, \_)$	(Assm. 1, WFRelInst2)
7:	$\iota'' \in \mathsf{dom}(\sigma)$	(6, Defn of dynType)
8:	$\sigma'(\iota'') = \sigma(\iota'')$	(Assm. 2, Assm. 3, 7, Lemma 1)
9:	$\mathcal{R}_P(r) = (\mathsf{dynType}(\sigma'(\iota'')), n_1, n_2, \_, \_)$	(6, 8)
10:	$\forall f \in \mathsf{dom}(\sigma(\iota)) : P, \sigma, \sigma(\iota) \vdash f \diamond_{\mathsf{fld}}$	(Assm. 1, WFRelInst2)
11:	$\forall f \in \mathsf{dom}(\sigma'(\iota)) : P, \sigma', \sigma'(\iota) \vdash f \diamond_{\mathsf{fld}}$	(Assm. 2, Assm. 3, 10, Lemma 7)
12:	$\vdash \mathsf{dynType}(\sigma(\iota_1)) \leq n_1$	(Assm. 1, WFRelInst2)
13:	$\iota_1 \in \mathsf{dom}(\sigma)$	(12, Defn of dynType)
14:	$\sigma(\iota_1) = \sigma'(\iota_1)$	(Assm. 2, Assm. 3, 13, Lemma 1)
15:	$\vdash \mathsf{dynType}(\sigma'(\iota_1)) \leq n_1$	(12, 14)
16:	$\vdash \mathsf{dynType}(\sigma(\iota_2)) \leq n_2$	(Assm. 1, WFRelInst2)
17:	$\vdash \mathsf{dynType}(\sigma'(\iota_2)) \leq n_2$	(16, Proof as for ι_1 above)
18:	$P, \sigma' \vdash o \diamond_{\mathsf{inst}}$	(5, 9, 11, 15, 17, WFRelInst2)

Therefore, in all cases, well-formed objects are preserved by store extension. □

Lemma 9. *Initial values of valid types are well-typed. For all well-formed P, Γ, σ and t such that $P \vdash t$, it is the case that $P, \Gamma, \sigma \vdash \mathit{initial}_P(t) : t$*

Proof Pick arbitrary Γ, σ. We then proceed by case analysis on the definition of $t \in \mathsf{Type}$.

Case 1: $t \in \mathsf{NominalType}$
Then $\mathsf{initial}_P(t) = \texttt{null}$ by definition, and $P, \Gamma, \sigma \vdash \mathsf{initial}_P(t) : t$ by DTNull.

Case 2: $t \in \{\texttt{true}, \texttt{false}\}$
Then $\mathsf{initial}_P(t) = \texttt{false}$ by definition, and $P, \Gamma, \sigma \vdash \mathsf{initial}_P(t) : t$ by DTBoolF.

Case 3: $t = \texttt{set<}n'\texttt{>}$, $n' \in \mathsf{NominalType}$
Then $\mathsf{initial}_P(t) = \emptyset$ by definition. Vacuously, $\forall \iota \in \mathsf{initial}_P(t) : P, \Gamma, \sigma \vdash \iota : n'$ so $P, \emptyset, \sigma \vdash \mathsf{initial}_P(t) : t$ by DTSet.

□

Lemma 10. *Field types are valid:*

$$\mathcal{F}_{P,n}(f) = t \Rightarrow P \vdash t$$

Proof By WTProgram, for all classes/relationships, $n \in \mathsf{dom}(\mathcal{C}_P) \cup \mathsf{dom}(\mathcal{R}_P)$, $P \vdash n$ must hold. For all such classes/relationships, by WTClass/WTRelationship, for all fields $f \in \mathsf{dom}(\mathcal{F}_n)$, $P, n \vdash f$ must hold. By WTField, for each such field, $\mathcal{F}_n(f) \in \mathsf{validTypes}_P$, so by Lemma 2 $P \vdash t$ as required. □

Corollary 11. *As for Lemma 10, but for $\mathcal{FD}$. Observe that for a type t to be in the range of $\mathcal{FD}$, there must be some n, f for which $\mathcal{F}_{P,n}(f) = t$. Proof then as above.*

Lemma 12. *Newly allocated class instances are well-formed: $\forall c \in \mathit{dom}(\mathcal{C}_P) : P, \sigma \vdash \mathit{new}_P(c) \diamond_{\mathit{inst}}$.*

Proof

Case 1: $c = \texttt{Object}$
Then $\mathsf{new}_P(c) = \langle\!\langle \texttt{Object} \| \rangle\!\rangle$, and the result follows immediately from WFObject1.

Case 2: $c \neq \texttt{Object}$
Clearly:

1: $\mathsf{dom}(\mathsf{new}_P(c)) = \mathsf{dom}(\mathcal{FD}_{P,c})$ (Definition of new)
2: $\mathsf{dynType}(\mathsf{new}_P(c)) = c$ (Definition of new)

It remains only to check that for all fields, $f \in \mathsf{dom}(\mathsf{new}_P(c))$ are well-formed. Pick an $f \in \mathsf{dom}(\mathsf{new}_P(c)) = \mathsf{dom}(\mathcal{FD}_{P,c})$. Then:

3: $\mathcal{FD}_{P,c}(f) = t$ ($f \in \mathsf{dom}(\mathcal{FD}_{P,c})$, Defn $\mathcal{FD}$)
4: $P \vdash t$ (3, Corollary 11)
5: $\mathsf{new}_P(c)(f) = \mathsf{initial}(t)$ (3, Defn of new)
6: $P, \emptyset, \sigma \vdash \mathsf{new}_P(c)(f) : t$ (4, 5, Lemma 9)
7: $P, \sigma, \mathsf{new}_P(c) \vdash f \diamond_{\mathsf{fld}}$ (2, 3, 6)

Therefore, all fields of a new instance of class c are well-formed:

8: $\forall f \in \mathsf{dom}(\mathsf{new}_P(c)) : P, \sigma, \mathsf{new}_P(c) \vdash f \diamond_{\mathsf{fld}}$ (7, f arbitrary)
9: $P, \sigma \vdash \mathsf{new}_P(c) \diamond_{\mathsf{inst}}$ (1, 8, WFObject2)

So all new object instances are well-formed, which was to be shown. □

Corollary 13. *Adding a new object to a well-formed heap gives a new well-formed heap:* $P \vdash \sigma \diamond_{heap} \wedge \iota \notin dom(\sigma) \Rightarrow P \vdash \sigma[\iota \mapsto new_P(c)] \diamond_{heap}$

Proof Let $\sigma' = \sigma[\iota \mapsto \mathsf{new}_P(c)]$ and pick arbitrary $\iota' \in \mathsf{dom}(\sigma')$. Either $\iota = \iota'$, in which case the instance at $\sigma'(\iota')$ is well-formed by Lemma 12, or $\iota \neq \iota'$, so $\iota \in \mathsf{dom}(\sigma)$, and the instance at $\sigma'(\iota')$ is well-formed by Lemma 8. □

Corollary 14. *We note that new instances of* `Relation` *in a store are well-formed, assuming the related objects are present, by* WFRelInst1.

Lemma 15. *Newly allocated relationship instances are well-formed. If:*
Assumption 1: $P \vdash \sigma \diamond_{heap}$
Assumption 2: $\sigma(\iota) = \langle\langle r', \_, \iota_1, \iota_2 \| \ldots \rangle\rangle$
Assumption 3: $\mathcal{R}_P(r) = (r', n_1, n_2, \_, \_)$
Assumption 4: $\vdash dynType(\sigma(\iota_1)) \leq n_1$
Assumption 5: $\vdash dynType(\sigma(\iota_2)) \leq n_2$
then $P, \sigma \vdash newPart_P(r, \iota, \iota_1, \iota_2) \diamond_{inst}$.

Proof
1: $\mathsf{dom}(\mathsf{newPart}_P(r, \iota, \iota_1, \iota_2)) = \mathsf{dom}(\mathcal{F}_{P,r})$ (Defn of newPart)
2: $\mathsf{dynType}(\mathsf{newPart}_P(r, \iota, \iota_1, \iota_2)) = r$ (Defn of newPart, dynType)
Pick arbitrary field $f \in \mathsf{dom}(\mathcal{F}_{P,r})$:
3: $\mathcal{F}_{P,r}(f) = t$ ($f \in \mathsf{dom}(\mathcal{F}_{P,r})$)
4: $\mathsf{newPart}_P(r, \iota, \iota_1, \iota_2)(f) = \mathsf{initial}_P(t)$ (3, Defn of newPart)
5: $P \vdash t$ (3, Lemma 10)
6: $P, \emptyset, \sigma \vdash \mathsf{newPart}_P(r, \iota, \iota_1, \iota_2)(f) : t$ (4, 5, Lemma 9)
7: $P, \sigma, \mathsf{newPart}_P(r, \iota, \iota_1, \iota_2) \vdash f \diamond_{\mathsf{fld}}$ (2, 3, 6, WFField)
So fields in the new instance are well-formed:
8: $\forall f \in \mathsf{dom}(\mathcal{F}_{P,r}) : P, \sigma, \mathsf{newPart}_P(r, \iota, \iota_1, \iota_2) \vdash f \diamond_{\mathsf{fld}}$ (7, f arbitrary)
The super-instance of r must match r's definition:
9: $\mathsf{dynType}(\sigma(\iota)) = r'$ (Assm. 2)
10: $\mathcal{R}_P(r) = (\mathsf{dynType}(\sigma(\iota)), n_1, n_2, \_, \_)$ (Assm. 3, 9)
So:
11: $P, \sigma \vdash \mathsf{newPart}_P(r, \iota, \iota_1, \iota_2) \diamond_{\mathsf{inst}}$ (Assm. 4, Assm. 5, 1, 8, 10, WFRelInst2)
□

Lemma 16. *Well-formed relationships unaffected by store extension. If:*
Assumption 1: $P, \sigma, \rho \vdash (r, \iota_1, \iota_2) \diamond_{rel}$
Assumption 2: $\iota \notin dom(\sigma)$
Assumption 3: $\sigma' = \sigma[\iota \mapsto o']$
Assumption 4: $(r', \iota'_1, \iota'_2) \notin dom(\rho)$
Assumption 5: $\rho' = \rho[(r, \iota'_1, \iota'_2) \mapsto \iota']$
then $P, \sigma', \rho' \vdash (r, \iota_1, \iota_2) \diamond_{rel}$.

Proof $P, \sigma, \rho \vdash (r, \iota_1, \iota_2) \diamond_{\mathsf{rel}}$ can arise either by WFRelation1 or WFRelation2.

Case 1: Assm. 1 arises by WFRelation1
Then $r = \texttt{Relation}$ and:

1: $\sigma(\rho(\texttt{Relation}, \iota_1, \iota_2)) = \langle\!\langle \texttt{Relation}, \texttt{null}, \iota_1, \iota_2 \| \rangle\!\rangle$	(Assm. 1, WFRelation1)
2: $\rho(\texttt{Relation}, \iota_1, \iota_2) \in \mathsf{dom}(\sigma)$	(1)
3: $\rho(\texttt{Relation}, \iota_1, \iota_2) \neq (r', \iota'_1, \iota'_2)$	(2,Assm. 4)
4: $\rho'(\texttt{Relation}, \iota_1, \iota_2) = \rho(\texttt{Relation}, \iota_1, \iota_2)$	(3, Assm. 5, Definition 1)
5: $\sigma(\rho'(\texttt{Relation}, \iota_1, \iota_2)) = \langle\!\langle \texttt{Relation}, \texttt{null}, \iota_1, \iota_2 \| \rangle\!\rangle$	(1, 4)
6: $\rho'(\texttt{Relation}, \iota_1, \iota_2) \in \mathsf{dom}(\sigma)$	(5)
7: $\rho'(\texttt{Relation}, \iota_1, \iota_2) \neq \iota$	(Assm. 2, 6)
8: $\sigma'(\rho'(\texttt{Relation}, \iota_1, \iota_2)) = \sigma(\rho'(\texttt{Relation}, \iota_1, \iota_2))$	(Assm. 3, 7, Definition 1)
9: $\sigma'(\rho'(\texttt{Relation}, \iota_1, \iota_2)) = \langle\!\langle \texttt{Relation}, \texttt{null}, \iota_1, \iota_2 \| \rangle\!\rangle$	(5, 8)
10: $P, \sigma', \rho' \vdash (r = \texttt{Relation}, \iota_1, \iota_2) \diamond_{\mathsf{rel}}$	(9, WFRelation1)

Case 2: Assm. 1 arises by WFRelation2

11: $\mathcal{R}_P(r) = (r'', \_, \_, \_, \_)$	(Assm. 1, WFRelation2)
12: $(r'', \iota_1, \iota_2) \in \mathsf{dom}(\rho)$	(Assm. 1, WFRelation2)
13: $\sigma(\rho(r, \iota_1, \iota_2)) = \langle\!\langle r, \rho(r'', \iota_1, \iota_2), \iota_1, \iota_2 \| \ldots \rangle\!\rangle$	(Assm. 1, WFRelation2)
14: $\rho(r, \iota_1, \iota_2) \in \mathsf{dom}(\sigma)$	(13)
15: $(r, \iota_1, \iota_2) \in \mathsf{dom}(\rho)$	(14)
16: $(r, \iota_1, \iota_2) \neq (r', \iota'_1, \iota'_2)$	(15, Assm. 4)
17: $(r'', \iota_1, \iota_2) \neq (r', \iota'_1, \iota'_2)$	(12, Assm. 4)
18: $\rho(r, \iota_1, \iota_2) = \rho'(r, \iota_1, \iota_2)$	(16, Assm. 5, Definition 1)
19: $\rho(r'', \iota_1, \iota_2) = \rho'(r'', \iota_1, \iota_2)$	(17, Assm. 5, Definition 1)
20: $\sigma(\rho'(r, \iota_1, \iota_2)) = \langle\!\langle r, \rho'(r'', \iota_1, \iota_2), \iota_1, \iota_2 \| \ldots \rangle\!\rangle$	(13, 18, 19)
21: $\rho'(r, \iota_1, \iota_2) \in \mathsf{dom}(\sigma)$	(20)
22: $\rho'(r, \iota_1, \iota_2) \neq \iota$	(21, Assm. 2)
23: $\sigma'(\rho'(r, \iota_1, \iota_2)) = \langle\!\langle r, \rho'(r'', \iota_1, \iota_2), \iota_1, \iota_2 \| \ldots \rangle\!\rangle$	(20, 22, Definition 1)
24: $(r'', \iota_1, \iota_2) \in \mathsf{dom}(\rho')$	(12, 19)
25: $P, \sigma', \rho' \vdash (r, \iota_1, \iota_2) \diamond_{\mathsf{rel}}$	(11, 23, 24, WFRelation2)

In both cases, therefore, the relationship is well-formed as required. □

Lemma 17. *For all* $\sigma, \sigma', \rho, \rho', r$ *and* $\iota, \iota_1, \iota_2 \in dom(\sigma)$*, where* $(\sigma', \rho') = addRel_P(r, \iota_1, \iota_2, \sigma, \rho)$*, then* $\sigma(\iota) = \sigma'(\iota)$.

Proof The proof proceeds by induction on the height of r in the relationship inheritance tree.

Case 1: $r = \texttt{Relation}$
If $(r, \iota_1, \iota_2) \in \mathsf{dom}(\rho)$ then $\sigma = \sigma'$ and the result is immediate. Otherwise, $\sigma' = \sigma[\iota' \mapsto \mathsf{newPart}_P(r, \texttt{null}, \iota_1, \iota_2)]$ with $\iota' \notin \mathsf{dom}(\sigma)$. Clearly $\iota \neq \iota'$ as $\iota \in \mathsf{dom}(\sigma)$, so by Definition 1, $\sigma(\iota) = \sigma'(\iota)$.

Case 2: $r \neq \texttt{Relation}$

Again, if $(r, \iota_1, \iota_2) \in \mathsf{dom}(\rho)$ then the result is immediate. Otherwise, $(\sigma'', \rho'') = \mathsf{addRel}_P(r', \iota_1, \iota_2, \sigma, \rho)$ where r' is the super-relationship of r. By induction, $\sigma''(\iota) = \sigma(\iota)$. The argument that $\sigma''(\iota) = \sigma'(\iota)$ then proceeds as above.

□

Lemma 18 (Safety of addRel). *addRel preserves well-formed stores and relationship stores. If:*

Assumption 1: $P \vdash \sigma \diamond_{heap}$
Assumption 2: $P, \sigma \vdash \rho \diamond_{relheap}$
Assumption 3: $(\sigma', \rho') = \mathsf{addRel}_P(r, \iota_1, \iota_2, \sigma, \rho)$
Assumption 4: $\mathcal{R}_P(r) = (\_, n_1, n_2, \_, \_)$
Assumption 5: $\vdash \mathsf{dynType}(\sigma(\iota_1)) \leq n_1$
Assumption 6: $\vdash \mathsf{dynType}(\sigma(\iota_2)) \leq n_2$

then $P \vdash \sigma' \diamond_{heap}$, $P, \sigma' \vdash \rho' \diamond_{relheap}$ *and* $(r, \iota_1, \iota_2) \in \mathsf{dom}(\rho')$.

Proof

1: $\forall \iota' \in \mathsf{dom}(\sigma) : P, \sigma \vdash \sigma(\iota) \diamond_{\mathsf{inst}}$ (Assm. 1, WFHeap)
2: $\forall (r', \iota'_1, \iota'_2) \in \mathsf{dom}(\rho) : P, \sigma, \rho \vdash (r', \iota'_1, \iota'_2) \diamond_{\mathsf{rel}}$ (Assm. 2, WFRelHeap)

We proceed by induction on the depth of r in the inheritance tree of P described by Lemma 3(a).

Case 1: $r = \texttt{Relation}$

If $\exists \iota' : \rho(r, \iota_1, \iota_2) = \iota'$, then $\sigma' = \sigma$ and $\rho' = \rho$ and the result is immediate. Assume therefore that $(r, \iota_1, \iota_2) \in \mathsf{dom}(\rho)$. Then:

3: $\sigma' = \sigma[\iota \mapsto \mathsf{newPart}_P(r, \texttt{null}, \iota_1, \iota_2)]$ (Defn of addRel)
4: $\rho' = \rho[(r, \iota_1, \iota_2) \mapsto \iota]$ (Defn of addRel)
5: $\iota \notin \mathsf{dom}(\sigma)$ (Defn of addRel)
6: $\forall \iota' \in \mathsf{dom}(\sigma) : P, \sigma' \vdash \sigma'(\iota') \diamond_{\mathsf{inst}}$ (1, 3, 5)
7: $\iota_1, \iota_2 \in \mathsf{dom}(\sigma)$ (Assm. 5,Assm. 6, Defn of dynType)
8: $\iota_1, \iota_2 \in \mathsf{dom}(\sigma')$ (3, Definition 1)
9: $\sigma'(\iota) = \mathsf{newPart}_P(r, \texttt{null}, \iota_1, \iota_2) = \langle\!\langle \texttt{Relation}, \texttt{null}, \iota_1, \iota_2 \| \rangle\!\rangle$ (3, Defn of newPart)
10: $P, \sigma' \vdash \sigma'(\iota) \diamond_{\mathsf{inst}}$ (8, 9, WFRelInst1)
11: $\forall \iota' \in \mathsf{dom}(\sigma') : P, \sigma' \vdash \sigma'(\iota') \diamond_{\mathsf{inst}}$ (3, 6, 10)
12: $P \vdash \sigma' \diamond_{\mathsf{heap}}$ (11, WFHeap)

We then show that the new relationship store is well-formed.

13: $\forall (r', \iota'_1, \iota'_2) \in \mathsf{dom}(\rho) : P, \sigma', \rho' \vdash (r', \iota'_1, \iota'_2) \diamond_{\mathsf{rel}}$ (2, 3, 4, 5, Lemma 16)
14: $\rho'(r, \iota_1, \iota_2) = \iota$ (4)
15: $\sigma'(\rho'(r, \iota_1, \iota_2)) = \mathsf{newPart}_P(r, \texttt{null}, \iota_1, \iota_2)$ (3, 14)
16: $\sigma'(\rho'(r, \iota_1, \iota_2)) = \langle\!\langle \texttt{Relation}, \texttt{null}, \iota_1, \iota_2 \| \rangle\!\rangle$ (15, Defn of newPart, $r = \texttt{Relation}$)
17: $P, \sigma', \rho' \vdash (r, \iota_1, \iota_2) \diamond_{\mathsf{rel}}$ (16, WFRelation1)
18: $\forall (r', \iota'_1, \iota'_2) \in \mathsf{dom}(\rho') : P, \sigma', \rho' \vdash (r', \iota'_1, \iota'_2) \diamond_{\mathsf{rel}}$ (13, 17)
19: $P, \sigma' \vdash \rho' \diamond_{\mathsf{relheap}}$ (18, WFRelHeap)

Finally, it is clear that $(r, \iota_1, \iota_2) \in \mathsf{dom}(\rho')$ by Statement 4 and Definition 1. Case complete.

Case 2: $r \neq \texttt{Relation}$

20: $\mathcal{R}(r) = (r', n_1, n_2, \_, \_)$ ($r \neq \texttt{Relation}$)
21: $(\sigma'', \rho'') = \mathsf{addRel}_P(r, \iota_1, \iota_2, \sigma, \rho)$ ($r \neq \texttt{Relation}$, Defn of addRel)
22: $\sigma' = \sigma''[\iota \mapsto \mathsf{newPart}_P(r, \rho''(r', \iota_1, \iota_2), \iota_1, \iota_2)]$ ($r \neq \texttt{Relation}$, Defn of addRel)

23: $\rho' = \rho''[(r, \iota_1, \iota_2) \mapsto \iota]$ ($r \neq$ `Relation`, Defn of addRel)
24: $\iota \notin \mathsf{dom}(\sigma'')$ ($r \neq$ `Relation`, Defn of addRel)

We now require the construction of the dynamic type constraints for use of the inductive hypothesis:

25: $P \vdash r$ (20, WTProgram)
26: $\mathcal{R}(r') = (\_, n'_1, n'_2, \_, \_)$ (20, 25 WTRelationship)
27: $P \vdash n_1 \leq n'_1$ (25, 26, WTRelationship)
28: $P \vdash n_2 \leq n'_2$ (25, 26, WTRelationship)
29: $P \vdash \mathsf{dynType}(\sigma(\iota_1)) \leq n'_1$ (Assm. 5, 27, STTrans)
30: $P \vdash \mathsf{dynType}(\sigma(\iota_2)) \leq n'_2$ (Assm. 6, 28, STTrans)

Then, by induction:

31: $P \vdash \sigma'' \diamond_{\mathsf{heap}}$ (Assm. 1, Assm. 2, 21, 26, 29, 30, Inductive hypothesis)
32: $P, \sigma'' \vdash \rho'' \diamond_{\mathsf{relheap}}$ (Assm. 1, Assm. 2, 21, 26, 29, 30, Inductive hypothesis)
33: $(r', \iota_1, \iota_2) \in \mathsf{dom}(\rho'')$ (Assm. 1, Assm. 2, 21, 26, 29, 30, Inductive hypothesis)

We continue to show that the extension of these stores in Statements 22 and 23 leaves them well-formed.

34: $\forall \iota' \in \mathsf{dom}(\sigma'') : P, \sigma'' \vdash \sigma''(\iota') \diamond_{\mathsf{inst}}$ (31, WFHeap)
35: $\forall \iota' \in \mathsf{dom}(\sigma'') : P, \sigma' \vdash \sigma'(\iota') \diamond_{\mathsf{inst}}$ (22, 24, 34, Lemma 8)
36: $\forall (r'', \iota'_1, \iota'_2) \in \mathsf{dom}(\rho'') : P, \sigma'', \rho'' \vdash (r'', \iota'_1, \iota'_2) \diamond_{\mathsf{rel}}$ (32, WFRelHeap)
37: $\sigma'(\iota) = \mathsf{newPart}_P(r, \rho''(r', \iota_1, \iota_2), \iota_1, \iota_2)$ (22, Definition 1)
38: $P, \sigma'', \rho'' \vdash (r', \iota_1, \iota_2) \diamond_{\mathsf{rel}}$ (33, 36)
39: $\sigma''(\rho''(r', \iota_1, \iota_2)) = \langle\!\langle r', \_, \iota_1, \iota_2 \| \ldots \rangle\!\rangle$ (38, WFRelation1/2)
40: $\vdash \mathsf{dynType}(\sigma''(\iota_1)) \leq n_1$ (Assm. 5, 21, Lemma 17)
41: $\vdash \mathsf{dynType}(\sigma''(\iota_2)) \leq n_2$ (Assm. 6, 21, Lemma 17)
42: $P, \sigma'' \vdash \sigma'(\iota) \diamond_{\mathsf{inst}}$ (20, 31, 37, 40, 41, Lemma 15)
43: $P, \sigma' \vdash \sigma'(\iota) \diamond_{\mathsf{inst}}$ (22, 24, 42, Lemma 8)
44: $\forall \iota' \in \mathsf{dom}(\sigma') : P, \sigma' \vdash \sigma'(\iota') \diamond_{\mathsf{inst}}$ (22, 35, 43)
45: $P \vdash \sigma' \diamond_{\mathsf{heap}}$ (44, WFHeap)

For the relation store:

46: $\forall (r'', \iota'_1, \iota'_2) \in \mathsf{dom}(\rho'') : P, \sigma', \rho' \vdash (r'', \iota'_1, \iota'_2) \diamond_{\mathsf{rel}}$ (22, 23, 24, 36, Lemma 16)
47: $\rho'(r, \iota_1, \iota_2) = \iota$ (23, Definition 1)
48: $\sigma'(\rho'(r, \iota_1, \iota_2)) = \langle\!\langle r, \rho''(r', \iota_1, \iota_2), \iota_1, \iota_2 \| \ldots \rangle\!\rangle$ (37, Defn newPart)
49: $\rho'(r', \iota_1, \iota_2) = \rho''(r', \iota_1, \iota_2)$ is defined (23, 33, $r \neq r'$, Definition 1)
50: $\sigma'(\rho'(r, \iota_1, \iota_2)) = \langle\!\langle r, \rho'(r', \iota_1, \iota_2), \iota_1, \iota_2 \| \ldots \rangle\!\rangle$ (48, 49)
51: $P, \sigma', \rho' \vdash (r, \iota_1, \iota_2) \diamond_{\mathsf{rel}}$ (20, 49, 50, WFRelation2)
52: $\forall (r', \iota'_1, \iota'_2) \in \mathsf{dom}(\rho') : P, \sigma', \rho' \vdash (r', \iota'_1, \iota'_2) \diamond_{\mathsf{rel}}$ (23, 46, 51)
53: $P, \sigma' \vdash \rho' \diamond_{\mathsf{relheap}}$ (52, WFRelHeap)

Finally, it is clear that $\rho'(r, \iota_1, \iota_2)$ is defined, according to Statement 23. Case complete.

Therefore, $\mathsf{addRel}(r, \iota_1, \iota_2)$ leaves both heaps well-formed, and sets up a relationship r between ι_1 and ι_2 in the relationship heap. □

Lemma 19 (Safety of remRel). *remRel preserves well-formed relationship stores. If* $P, \sigma \vdash \rho \diamond_{\mathsf{relheap}}$ *and* $\rho' = \mathsf{remRel}_P(r, \iota_1, \iota_2, \rho)$*, then* $P, \sigma \vdash \rho' \diamond_{\mathsf{relheap}}$ *.*

Proof Unfolding the definition of remRel:

$$\begin{aligned}\rho' &= \mathsf{remRel}_P(r, \iota_1, \iota_2, \rho) \\ &= \rho \setminus \{(r', \iota_1, \iota_2) \mid \vdash r' \leq r\} \\ &= \{(r', \iota_1', \iota_2') \mapsto \rho(r', \iota_1', \iota_2') \mid \not\vdash r' \leq r \vee \iota_1' \neq \iota_1 \vee \iota_2' \neq \iota_2\}\end{aligned}$$

Assume then that:

1: $P, \sigma \vdash \rho \diamond_{\mathsf{relheap}}$

2: $\forall (r', \iota_1', \iota_2') \in \mathsf{dom}(\rho) : P, \sigma, \rho \vdash (r', \iota_1', \iota_2') \diamond_{\mathsf{rel}}$ (1, WFRelHeap)

Pick arbitrary $(r', \iota_1', \iota_2') \in \mathsf{dom}(\rho')$. Then:

3: $\rho'(r', \iota_1', \iota_2') = \rho(r', \iota_1', \iota_2')$ (Defn of ρ')

4: $P, \sigma, \rho \vdash (r', \iota_1', \iota_2') \diamond_{\mathsf{rel}}$ (2, $(r', \iota_1', \iota_2') \in \mathsf{dom}(\rho') \subseteq \mathsf{dom}(\rho)$)

As $(r', \iota_1', \iota_2') \in \mathsf{dom}(\rho)$, there are three cases according to the definition of ρ':

Case 1: $\iota_1 \neq \iota_1'$

Statement 4 can arise either by WFRelation1 or WFRelation2.

Case 1.1: Statement 4 arises by WFRelation1

Then $r' = \texttt{Relation}$:

5: $\sigma(\rho(r', \iota_1', \iota_2')) = \langle\!\langle \texttt{Relation}, \texttt{null}, \iota_1, \iota_2 \| \rangle\!\rangle$ (4, WFRelation1)

6: $\sigma(\rho'(r', \iota_1', \iota_2')) = \langle\!\langle \texttt{Relation}, \texttt{null}, \iota_1, \iota_2 \| \rangle\!\rangle$ (5, 3)

7: $P, \sigma, \rho' \vdash (r', \iota_1', \iota_2') \diamond_{\mathsf{rel}}$ (6, WFRelation1)

Case 1.2: Statement 4 arises by WFRelation2

Then:

8: $\mathcal{R}_P(r') = (r'', \_, \_, \_, \_)$ (4, WFRelation2)

9: $(r'', \iota_1', \iota_2') \in \mathsf{dom}(\rho)$ (4, WFRelation2)

10: $\sigma(\rho(r', \iota_1', \iota_2')) = \langle\!\langle r', \rho(r'', \iota_1', \iota_2'), \iota_1, \iota_2 \| \ldots \rangle\!\rangle$ (4, WFRelation2)

11: $\rho'(r'', \iota_1', \iota_2') = \rho(r'', \iota_1', \iota_2')$ ($\iota_1' \neq \iota_1$, Defn of ρ')

12: $(r'', \iota_1', \iota_2') \in \rho$ (11, 9)

13: $\sigma(\rho(r', \iota_1', \iota_2')) = \langle\!\langle r', \rho'(r'', \iota_1', \iota_2'), \iota_1, \iota_2 \| \ldots \rangle\!\rangle$ (10, 12)

14: $\sigma(\rho'(r', \iota_1', \iota_2')) = \langle\!\langle r', \rho'(r'', \iota_1', \iota_2'), \iota_1, \iota_2 \| \ldots \rangle\!\rangle$ (3, 13)

15: $P, \sigma, \rho' \vdash (r', \iota_1', \iota_2') \diamond_{\mathsf{rel}}$ (8, 12, 14, WFRelation2)

Case 2: $\iota_2 \neq \iota_2'$

Similar to previous case

Case 3: $\not\vdash r' \leq r$

Again, there are two possible origins for Statement 4:

Case 3.1: Statement 4 arises by WFRelation1

Similar to proof for WFRelation1 case above.

Case 3.2: Statement 4 arises by WFRelation2

Then:

16: $\mathcal{R}_P(r') = (r'', \_, \_, \_, \_)$ (4, WFRelation2)

17: $(r'', \iota_1', \iota_2') \in \mathsf{dom}(\rho)$ (4, WFRelation2)

18: $\sigma(\rho(r', \iota_1', \iota_2')) = \langle\!\langle r', \rho(r'', \iota_1', \iota_2'), \iota_1, \iota_2 \| \ldots \rangle\!\rangle$ (4, WFRelation2)

19: $\vdash r' \leq r''$ (16, STRel)

20: $\not\vdash r'' \leq r$ (16, Lemma 3(e))

21: $\rho'(r'', \iota_1', \iota_2') = \rho(r'', \iota_1', \iota_2')$ is defined (17, 20, Defn of ρ')

22: $\sigma(\rho(r', \iota_1', \iota_2')) = \langle\!\langle r', \rho'(r'', \iota_1', \iota_2'), \iota_1, \iota_2 \| \ldots \rangle\!\rangle$ (18, 21)

23: $\sigma(\rho'(r', \iota_1', \iota_2')) = \langle\!\langle r', \rho'(r'', \iota_1', \iota_2'), \iota_1, \iota_2 \| \ldots \rangle\!\rangle$ (3, 22)
24: $P, \sigma, \rho' \vdash (r', \iota_1', \iota_2') \diamond_{\mathsf{rel}}$ (16, 21, 23, WFRelation2)

Therefore, $P, \sigma, \rho' \vdash (r', \iota_1', \iota_2') \diamond_{\mathsf{rel}}$ in all cases.
25: $\forall (r', \iota_1', \iota_2') \in \mathsf{dom}(\rho') : P, \sigma, \rho' \vdash (r', \iota_1', \iota_2') \diamond_{\mathsf{rel}}$ ((r', ι_1', ι_2') arbitrary)
26: $P, \sigma \vdash \rho' \diamond_{\mathsf{relheap}}$ (25, WFRelHeap)
□

Lemma 20 (Safety of field update). *Field update preserves well-formed fields. If:*
Assumption 1: $P, \sigma, o \vdash f \diamond_{\mathsf{fld}}$
Assumption 2: $o' = o[f \mapsto u]$
Assumption 3: $P, \emptyset, \sigma \vdash u : \mathcal{FD}_{P,\mathsf{dynType}(o)}(f)$
then $P, \sigma, o' \vdash f \diamond_{\mathsf{fld}}$ *.*

Proof
1: $\mathsf{dynType}(o) = \mathsf{dynType}(o') = n$ (Assm. 1, Assm. 2, WFField)
2: $\mathcal{FD}_{P,n}(f) = \mathcal{FD}_{P,\mathsf{dynType}(o)}(f) = \mathcal{FD}_{P,\mathsf{dynType}(o')}(f) = t$ (Assm. 1, Assm. 2, WFField)
3: $P, \emptyset, \sigma \vdash o'(f) : t$ (Assm. 2, Assm. 3)
4: $P, \sigma, o' \vdash f \diamond_{\mathsf{fld}}$ (1, 2, 3, WFField)
□

Corollary 21. *A well-formed object updated in a well-typed way will remain well-formed:* $P, \sigma \vdash o \diamond_{\mathsf{inst}} \wedge f \in \mathsf{dom}(o) \wedge P, \emptyset, \sigma \vdash u : \mathcal{FD}_{P,\mathsf{dynType}(o)}(f) \Rightarrow P, \sigma \vdash o[f \mapsto u] \diamond_{\mathsf{inst}}$

Proof The proof is by Lemma 20 for the updated field, and by simple equality of Γ and σ for the unaltered fields. We require $f \in \mathsf{dom}(o)$, in order to prevent super-relationship fields being added to sub-relationship instances. □

Lemma 22 (Safety of object update — fields). *Replacing an object in the store with an object of the same dynamic type preserves well-formed fields. If:*
Assumption 1: $P, \sigma, o \vdash f \diamond_{\mathsf{fld}}$
Assumption 2: $\sigma' = \sigma[\iota \mapsto o']$
Assumption 3: $\mathsf{dynType}(o') = \mathsf{dynType}(\sigma(\iota))$
then $P, \sigma', o \vdash f \diamond_{\mathsf{fld}}$ *.*

Proof
1: $\mathsf{dynType}(o) = n$ (Assm. 1, WFField)
2: $\mathcal{FD}_{P,n}(f) = t$ (Assm. 1, WFField)
3: $P, \emptyset, \sigma \vdash o(f) : t$ (Assm. 1, WFField)

Case 1: $t =$ `boolean`
Then $P, \emptyset, \sigma' \vdash o(f) : t$ by DTBoolF/T.

Case 2: $t \in \mathsf{NominalType}$

4: $\vdash \mathsf{dynType}(\sigma(o(f))) \leq t$ (3, DTAddr)

If $o(f) = \iota$ then $\mathsf{dynType}(\sigma(o(f))) = \mathsf{dynType}(\sigma'(o(f)))$ by Statements Assm. 2 and Assm. 3. Otherwise, if $o(f) \neq \iota$, then $\sigma'(o(f)) = \sigma(o(f))$ so $\mathsf{dynType}(\sigma(o(f))) = \mathsf{dynType}(\sigma'(o(f)))$ by Statement Assm. 2 and Definition 1. In either case:

5: $\vdash \mathsf{dynType}(\sigma'(o(f))) \leq t$ (4)
6: $P, \emptyset, \sigma' \vdash o(f) : t$ (5, DTAddr)

Case 3: $t =$ `set<`$n \in \mathsf{NominalType}$`>`

7: $\forall \iota' \in o(f) : P, \emptyset, \sigma \vdash \iota' : n$ (3, DTSet)
8: $\forall \iota' \in o(f) : P, \emptyset, \sigma' \vdash \iota' : n$ (7, Similar to previous case)
9: $P, \emptyset, \sigma' \vdash o(f) : t$ (8, DTSet)

Thus, in all cases:
10: $P, \emptyset, \sigma' \vdash o(f) : t$ (Case analysis)
11: $P, \sigma, o \vdash f \diamond_{\mathsf{fld}}$ (1, 2, 10, WFField)
□

Lemma 23 (Safety of object update — objects). *Substitution with a new well-formed object of the same type preserves well-formed objects. If:*
Assumption 1: $P, \sigma \vdash o \diamond_{\mathit{inst}}$
Assumption 2: $\sigma' = \sigma[\iota \mapsto o']$
Assumption 3: $\mathit{dynType}(o') = \mathit{dynType}(\sigma(\iota))$
then $P, \sigma' \vdash o \diamond_{\mathit{inst}}$ *.*

Proof Firstly, we observe that all objects must retain their dynamic type:
1: $\forall \iota' \in \mathsf{dom}(\sigma) : \mathsf{dynType}(\sigma(\iota')) = \mathsf{dynType}(\sigma'(\iota'))$ (Assm. 3, Definition 1)
Then, Assm. 1 could arise by WFObject1, WFObject2, WFRelInst1 or WFRelInst2. Proceed by case analysis:

Case 1: Statement Assm. 1 arises by WFObject1
Then $o = \langle\!\langle \texttt{Object} \| \rangle\!\rangle$, and the result follows immediately.

Case 2: Statement Assm. 1 arises by WFObject2
Then:

2: $\mathsf{dom}(o) = \mathsf{dom}(\mathcal{FD}_{P,c})$ (Assm. 1, WFObject2)
3: $\mathsf{dynType}(o) = c$ (Assm. 1, WFObject2)
4: $\forall f \in \mathsf{dom}(o) : P, \sigma, o \vdash f \diamond_{\mathsf{fld}}$ (Assm. 1, WFObject2)
5: $\forall f \in \mathsf{dom}(o) : P, \sigma', o \vdash f \diamond_{\mathsf{fld}}$ (Assm. 2, Assm. 3, 4, Lemma 22)
6: $P, \sigma \vdash o \diamond_{\mathsf{inst}}$ (2, 3, 5, WFObject2)

Case 3: Statement Assm. 1 arises by WFRelInst1
Then $o = \langle\!\langle \texttt{Relation}, \texttt{null}, \iota_1, \iota_2 \| \rangle\!\rangle$ and $\iota_1, \iota_2 \in \mathsf{dom}(\sigma)$. Clearly, $\iota_1, \iota_2 \in \mathsf{dom}(\sigma')$, so the result follows immediately by WFRelInst1.

Case 4: Statement Assm. 1 arises by WFRelInst2
Then $o = \langle\!\langle r, \iota', \iota_1, \iota_2 \| \ldots \rangle\!\rangle$.

7: $\mathsf{dom}(o) = \mathsf{dom}(\mathcal{F}_{P,c})$ (Assm. 1, WFRelInst2)
8: $\forall f \in \mathsf{dom}(o) : P, \sigma, o \vdash f \diamond_{\mathsf{fld}}$ (Assm. 1, WFRelInst2)
9: $\forall f \in \mathsf{dom}(o) : P, \sigma', o \vdash f \diamond_{\mathsf{fld}}$ (Assm. 2, Assm. 3, 8, Lemma 22)
10: $\mathcal{R}_P(r) = (\mathsf{dynType}(\sigma(\iota')), n_1, n_2, \_, \_)$ (Assm. 1, WFRelInst2)
11: $\mathcal{R}_P(r) = (\mathsf{dynType}(\sigma'(\iota')), n_1, n_2, \_, \_)$ (1, 10)
12: $\vdash \mathsf{dynType}(\sigma(\iota_1)) \leq n_1$ (Assm. 1, WFRelInst2)
13: $\vdash \mathsf{dynType}(\sigma'(\iota_1)) \leq n_1$ (1, 12)
14: $\vdash \mathsf{dynType}(\sigma(\iota_2)) \leq n_2$ (Assm. 1, WFRelInst2)
15: $\vdash \mathsf{dynType}(\sigma'(\iota_2)) \leq n_2$ (1, 14)
16: $P, \sigma \vdash o \diamond_{\mathsf{inst}}$ (7, 11, 13, 15, WFRelInst2)

Thus, in all cases, objects remain well-formed in a store under well-typed update, which was to be shown. □

Corollary 24 (Safety of object update — heaps).
Well-typed object update preserves the heap's well-formedness. If:
- $P \vdash \sigma \diamond_{heap}$
- $f \in dom(\sigma(\iota))$
- $P, \Gamma, \sigma \vdash u : \mathcal{FD}_{P,dynType(\sigma(\iota))}(f)$

then $P \vdash \sigma[\iota \mapsto \sigma(\iota)[f \mapsto u]] \diamond_{heap}$.

Proof Let $\sigma' = \sigma[\iota \mapsto \sigma(\iota)[f \mapsto u]]$. The updated object at $\sigma'(\iota)$ is well-formed by Corollary 21. All other objects in σ' are well-formed by Lemma 23 (as the dynamic type of the substituted $\sigma(\iota)[f \mapsto u]$ object is clearly the same as $\sigma(\iota)$). □

Lemma 25 (Objects match definition). $P \vdash \sigma \diamond_{heap} \ \wedge \ dynType(\sigma(\iota)) = n \ \wedge \ f \in dom(\mathcal{FD}_{P,n}) \Rightarrow (\sigma, \iota, f) \in dom(fld)$

Proof Pick arbitrary (σ, ι, f) such that $P \vdash \sigma \diamond_{\mathsf{heap}}$, $\mathsf{dynType}(\sigma(\iota)) = n$ and $f \in \mathsf{dom}(\mathcal{FD}_{P,n})$. Then the object $\sigma(\iota)$ must be well-formed by WFHeap.

If $\sigma(\iota)$ is an instance of `Object`, then $\mathsf{dom}(\mathcal{FD}_{P,n}) = \emptyset$, so this case does not arise.

If $\sigma(\iota)$ is an instance of a class, then $\mathsf{dom}(\mathcal{FD}_{P,n}) = \mathsf{dom}(\sigma(\iota))$ by WFObject2, and $\mathcal{FD}_{P,n}(f) = \sigma(\iota)(f)$, which is defined as required.

Suppose $\sigma(\iota)$ is an instance of a relationship, then the remainder of the proof proceeds by induction on the depth of n in the relationship hierarchy determined by Lemma 3(a).

If $\sigma(\iota)$ is an instance of `Relation`, then $\mathsf{dom}(\mathcal{FD}_{P,n}) = \emptyset$, so this case does not arise.

Then if $f \in \mathsf{dom}(\mathcal{F}_{P,n})$, then $f \in \mathsf{dom}(\sigma(\iota))$ by WFRelInst2 as required. If $f \notin \mathsf{dom}(\mathcal{F}_{P,n})$ then it must be that $f \in \mathsf{dom}(\mathcal{FD}_{P,r'})$ where $\mathcal{R}_P(n) = (r', \_, \_, \_, \_)$ by definition of $\mathcal{FD}_{P,n}$. By induction, $(\sigma, \iota, f) \in \mathsf{dom}(\mathsf{fld})$ as required. □

Lemma 26. $P \vdash \sigma \diamond_{heap} \ \wedge \ dynType(\sigma(\iota)) = n \ \wedge \ f \in dom(\mathcal{FD}_{P,n}) \Rightarrow (\sigma, \iota, f, u) \in dom(fldUpd)$

Proof Proof is as for Lemma 25, above. □

Lemma 27 (Safety of fld). *If:*
- $P \vdash \sigma \diamond_{heap}$
- $P, \Gamma, \sigma \vdash \iota : n$
- $\mathcal{FD}_{P,n}(f) = t$

then $P, \Gamma, \sigma \vdash fld(\sigma, \iota, f) : t$.

Proof By Lemma 25, $\mathsf{fld}(\sigma, \iota, f) = u$. The result follows by WFHeap. □

Lemma 28 (Safety of fldUpd).
fldUpd preserves σ's well-formedness. If:
- $P \vdash \sigma \diamond_{heap}$
- $P, \Gamma, \sigma \vdash u : \mathcal{FD}_{P,dynType(\sigma(\iota))}(f)$
- $\sigma' = fldUpd(\sigma, \iota, f, u)$

Then $P \vdash \sigma' \diamond_{heap}$.

Proof $\mathsf{fldUpd}(\sigma, \iota, f, u)$ is defined by Lemma 26. fldUpd then either takes no action or returns a σ updated as in Corollary 24. □

Lemma 29 (Safety of store extension — typing). *Values typable with any well-formed store are also typable with any (possibly larger) store whose objects have the same dynamic type.*
- $P \vdash \sigma \diamond_{heap}$
- $P \vdash \sigma' \diamond_{heap}$
- $dynType(\sigma(\iota)) = t \Rightarrow dynType(\sigma'(\iota)) = t$

– $P, \Gamma, \sigma \vdash e : t$

then $P, \Gamma, \sigma' \vdash e : t$. *Similarly statements and statement sequences.*

Proof sketch By induction on the derivation of $P, \Gamma, \sigma \vdash e : t$:

Case 1: Derivation ends in DTAddr

Then $u = \iota$ and $\vdash \mathsf{dynType}(\sigma(\iota)) \leq t$. But $\mathsf{dynType}(\sigma'(\iota)) = \mathsf{dynType}(\sigma(\iota))$, so $\vdash \mathsf{dynType}(\sigma'(\iota)) \leq t$ and by DTAddr, $P, \Gamma, \sigma' \vdash \iota : t$.

Case 2: Derivation ends in DTSet

Then $u = \{\iota_1, \ldots, \iota_n\}$ and $t = \texttt{set<}n\texttt{>}$ and for every ι_i $P, \Gamma, \sigma \vdash \iota_i : n$. By induction, for every ι_i, $P, \Gamma, \sigma' \vdash \iota_i : n$, and the result follows by DTSet

The remaining cases do not rely on σ and follow by direct induction. □

Corollary 30 (Safety of store extension — locals). *Well-formed locals stores are preserved under the same conditions as in Lemma 29:*

Assumption 1: $P \vdash \sigma_1 \diamond_{\mathsf{heap}}$

Assumption 2: $P, \Gamma, \sigma_1 \vdash \lambda \diamond_{\mathsf{locals}}$

Assumption 3: $\mathsf{dynType}(\sigma_1(\iota)) = t \Rightarrow \mathsf{dynType}(\sigma_2(\iota)) = t$

then $P, \Gamma, \sigma_2 \vdash \lambda \diamond_{\mathsf{locals}}$.

Proof An immediate consequence of Lemma 29 and WFLocals. □

Lemma 31 (Safety of store extension — relationships). *With two well-formed stores, where the objects in the first have the same dynamic types as those in the second, well-formed relationships are preserved. Suppose*

Assumption 1: $P \vdash \sigma \diamond_{\mathsf{heap}}$

Assumption 2: $P \vdash \sigma' \diamond_{\mathsf{heap}}$

Assumption 3: $\sigma(\iota) = \langle\!\langle r, \_, \iota_1, \iota_2 \| \_ \rangle\!\rangle \Rightarrow \sigma'(\iota) = \langle\!\langle r, \_, \iota_1, \iota_2 \| \_ \rangle\!\rangle$

Assumption 4: $P, \sigma, \rho \vdash (r, \iota_1, \iota_2) \diamond_{\mathsf{rel}}$

then $P, \sigma', \rho \vdash (r, \iota_1, \iota_2) \diamond_{\mathsf{rel}}$.

Proof By inspection of the rules WFRelation1 and WFRelation2. □

Corollary 32 (Safety of store extension — relationship heap).
Under the conditions of Lemma 31, $P, \sigma' \vdash \rho \diamond_{\mathsf{relheap}}$, *by* WFRelHeap.

Lemma 33 (Safety of variable update). *Well-typed updates of local variables preserve local-store's well-formedness. If:*

– $P, \Gamma, \sigma \vdash \lambda \diamond_{\mathsf{locals}}$

– $\lambda' = \lambda[x \mapsto u]$

– $P, \Gamma, \sigma \vdash u : \Gamma(x)$

then $P, \Gamma, \sigma \vdash \lambda' \diamond_{\mathsf{locals}}$.

Proof Pick arbitrary $x' \in \mathsf{dom}(\lambda')$. It must be shown that $P, \Gamma, \sigma \vdash \lambda'(x') : \Gamma(x')$. If $x \neq x'$, then $x' \in \mathsf{dom}(\lambda)$. As λ is well-formed, then $P, \Gamma, \sigma \vdash \lambda(x') : \Gamma(x')$ by GoodLocals. Clearly $\lambda(x) = \lambda'(x')$, so $P, \Gamma, \sigma \vdash \lambda'(x') : \Gamma(x')$ as required.

Where $x' = x$, $P, \Gamma, \sigma \vdash u : \Gamma(x)$ by assumption. As all locals are typed consistently with the typing environment, then, $P, \Gamma, \sigma \vdash \lambda' \diamond_{\mathsf{locals}}$. □

Lemma 34 (Context Weakening). *Extension of the typing environment preserves typing:*
$P, \Gamma_1, \sigma \vdash e : t$ *and* $\Gamma_1 \subseteq \Gamma_2$ *implies* $P, \Gamma_2, \sigma \vdash e : t$. *Similarly for* $P, \Gamma_1, \sigma \vdash s$.

Proof sketch Result follows obviously from DTVar, then by straight-forward induction over other typing rules. We give only two cases, by way of example:

Case 1: $e = x$

By DTVar, $\Gamma_1(x) = t$. $\Gamma_1 \subseteq \Gamma_2$, so $\Gamma_2(x) = t$. By DTVar, $P, \Gamma_2, \sigma \vdash e : t$ as required.

Case 2: $e = e'.f$

By DTFld, $P, \Gamma_1, \sigma \vdash e' : n$ and $\mathcal{FD}_n(f) = \mathrm{t}$. By induction, $P, \Gamma_2, \sigma \vdash e' : n$. DTFld yields $P, \Gamma_1, \sigma \vdash e'.f : t$ as required.

The rest of the proof proceeds similarly. □

Lemma 35 (Soundness of context substitution).

If:

- $P, \Gamma, \sigma \vdash e_1 : t_1$
- $P, \Gamma, \sigma \vdash e_2 : t_2$
- $P \vdash t_2 \leq t_1$

then for any context $\mathcal{E}$*,*

(a) $P, \Gamma, \sigma \vdash \mathcal{E}[e_1] : t_3 \Rightarrow P, \Gamma, \sigma \vdash \mathcal{E}[e_2] : t_4 \ \wedge \ P \vdash t_4 \leq t_3$ *and*

(b) $P, \Gamma, \sigma \vdash \mathcal{E}[e_1] \Rightarrow P, \Gamma, \sigma \vdash \mathcal{E}[e_2]$.

Proof By induction on the derivation of $P, \Gamma, \sigma \vdash \mathcal{E}[e_1] : t_1$ and $P, \Gamma, \sigma \vdash \mathcal{E}[e_1]$. Notice that $\mathcal{E}$ is either an expression context, in which case $\mathcal{E}[e_1] \in \mathsf{DynExpression}$ or a statement context, in which case $\mathcal{E}[e_1] \in \mathsf{DynStatement}$. As $\mathsf{DynExpression} \cap \mathsf{DynStatement} = \emptyset$, at most one of the above typings can hold for any given context/expression pair.

Case 1: Derivation ends with DTBoolT, DTBoolF, DTNull, DTAddr, DTSet, DTVar or DTNew

Then $P, \Gamma, \sigma \vdash \mathcal{E}[e_1] : t$ and $\mathcal{E} = \bullet$. The result follows immediately from the assumptions.

Case 2: Derivation ends with DTEq

Then $\mathcal{E}[e_1] = (e_3 \texttt{ == } e_4)$ where $\mathcal{E} = (\mathcal{E}'_{\mathrm{e}} \texttt{ == } e_4)$ or $\mathcal{E} = (e_3 \texttt{ == } \mathcal{E}'_{\mathrm{e}})$.

Suppose that $\mathcal{E} = (\mathcal{E}'_{\mathrm{e}} \texttt{ == } e_4)$. Then, by DTEq, $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_1] : n_1$ where $n_1 = t_1$, $P, \Gamma, \sigma \vdash e : n'$, and $t_3 = \texttt{boolean}$. By induction, $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_2] : n_2$ where $P \vdash n_2 \leq n_1$ and $n_2 = t_2$. Then, by DTEq, $P, \Gamma, \sigma \vdash \mathcal{E}[e_2] : \texttt{boolean}$ and $t_3 = t_4$ as required.

The case where $\mathcal{E} = (e_3 \texttt{ == } \mathcal{E}'_{\mathrm{e}})$ proceeds similarly.

Case 3: Derivation ends with DTFld

Then $\mathcal{E}[e_1] = e_3.f$ where $\mathcal{E} = \mathcal{E}'_{\mathrm{e}}.f$.

Then, by DTFld, $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_1] : n$ where $n = t_1$ and $\mathcal{FD}_n(f) = t_3$. By induction, $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_2] : n'$ where $n' = t_2$ and $P \vdash n' \leq n$, and by Lemma 4, $\mathcal{FD}_{n'}(f) = t_3$. By DTFld, $P, \Gamma, \sigma \vdash \mathcal{E}[e_2] : t_3$ so $t_3 = t_4$ as required.

Case 4: Derivation ends with DTRelObj

Then $\mathcal{E}[e_1] = e_3.r$ where $\mathcal{E} = \mathcal{E}'_{\mathrm{e}}.r$.

Then by DTRelObj, $t_3 = \texttt{set<}n_3\texttt{>}$, $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_1] : n_1$, $\mathcal{R}(r) = (\_, n_2, n_3, \_, \_)$ and $P \vdash n_1 \leq n_2$. By induction, $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_2] : n'_1$ and $P \vdash n'_1 \leq n_1$, so $P \vdash n'_1 \leq n_2$ by transitivity of subtyping. By DTRelObj, then, $P, \Gamma, \sigma \vdash \mathcal{E}[e_2] : \texttt{set<}n_3\texttt{>}$ as required.

Case 5: Derivation ends with DTRelInst

Similar to previous case.

Case 6: Derivation ends with DTFrom
Then $\mathcal{E}[e_1] = e_3\texttt{.from}$ and $\mathcal{E} = \mathcal{E}'_{\mathrm{e}}\texttt{.from}$.

By DTFrom, $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_1] : r_1$, $\mathcal{R}(r_1) = (\_, n_1, \_, \_, \_)$ and $t_3 = n_1$. By induction, $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_2] : t'$ and $P \vdash t' \leq r_1$. By Lemma 3(h), $t' = r_2$, $\mathcal{R}(r_2) = (\_, n_2, \_, \_, \_)$ and $P \vdash n_2 \leq n_1$ by the covariance portion of WTRelationship. DTFrom then yields $P, \Gamma, \sigma \vdash \mathcal{E}_{\mathrm{e}}[e_2] : n_2$ where $P \vdash n_2 \leq n_1$ as required.

Case 7: Derivation ends with DTTo
Similar to above.

Case 8: Derivation ends with DTAdd
Then $\mathcal{E}[e_1] = e_3 \texttt{ + } e_4$ and $\mathcal{E} = \mathcal{E}'_{\mathrm{e}} \texttt{ + } e_4$ or $e_3 \in \mathsf{DynValue}$ and $\mathcal{E} = e_3 \texttt{ + } \mathcal{E}'_{\mathrm{e}}$.

Case 8.1: $\mathcal{E} = \mathcal{E}'_{\mathrm{e}} \texttt{ + } e_4$
Then by DTAdd, $t_3 = \texttt{set<}n_3\texttt{>}$, $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_1] : \texttt{set<}n_1\texttt{>}$, $P, \Gamma, \sigma \vdash e_4 : n'$, $P \vdash n' \leq n_3$ and $P \vdash n_1 \leq n_3$. By induction, $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_2] : t'_2$ and $P \vdash t'_2 \leq \texttt{set<}n_1\texttt{>}$, so by Lemma 3(i) $t'_2 = \texttt{set<}n_2\texttt{>}$. $P \vdash n_2 \leq n_1$ by STCov and $P \vdash n_2 \leq n_3$ by transitivity of subtyping. Finally, by DTAdd, $P, \Gamma, \sigma \vdash \mathcal{E}_{\mathrm{e}}[e_2] : n_3$ as required.

Case 8.2: $\mathcal{E} = e_3 \texttt{ + } \mathcal{E}'_{\mathrm{e}}$
Then by DTAdd, $t_3 = \texttt{set<}n_3\texttt{>}$, $P, \Gamma, \sigma \vdash e_3 : \texttt{set<}n'\texttt{>}$, $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_1] : n_1$, $P \vdash n' \leq n_3$ and $P \vdash n_1 \leq n_3$. By induction, $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_2] : n_2$ where $P \vdash n_2 \leq n_1$ implying $P \vdash n_2 \leq n_3$ by transitivity of subtyping. By DTAdd, $P, \Gamma, \sigma \vdash \mathcal{E}_{\mathrm{e}}[e_2] : n_3$ as required.

Case 9: Derivation ends with DTSub
Similar to case above.

Case 10: Derivation ends with DTAss
Then $\mathcal{E}[e_1] = x \texttt{ = } \mathcal{E}'_{\mathrm{e}}$ and $\mathcal{E} = x \texttt{ = } \bullet$.

By DTAss, $x \neq \texttt{this}$, $P, \Gamma, \sigma \vdash x : t$, $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_1] : t'$ and $P \vdash t' \leq t$. By induction, $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_2] : t''$ where $P \vdash t'' \leq t'$. Then, $P \vdash t'' \leq t$ by transitivity and the required result follows by DTAss.

Case 11: Derivation ends with DTFldAss
Then $\mathcal{E}[e_1] = e_3\texttt{.}f \texttt{ = } e_4$ and $\mathcal{E} = \bullet\texttt{.}f \texttt{ = } e_4$ or $e_3 \in \mathsf{DynValue}$ and $\mathcal{E} = e_3\texttt{.}f \texttt{ = } e_4$.

Case 11.1: $\mathcal{E} = \mathcal{E}'_{\mathrm{e}}\texttt{.}f \texttt{ = } e_4$
By DTFldAss, $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_1] : n_1$, $P, \Gamma, \sigma \vdash e : t$, $\mathcal{FD}_{n_1}(f) = t'$ and $P \vdash t \leq t'$. By induction, $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_2] : n_2$ where $P \vdash n_2 \leq n_1$. By Lemma 4, $\mathcal{FD}_{n_2}(f) = t'$. By DTFldAss, $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_2]\texttt{.}f \texttt{ = } e : n_2$, recalling that $P \vdash n_2 \leq n_1$ as required.

Case 11.2: $\mathcal{E} = e_3\texttt{.}f \texttt{ = } \mathcal{E}'_{\mathrm{e}}$
By DTFldAss, $P, \Gamma, \sigma \vdash e_3 : n$, $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_1] : t$, $\mathcal{FD}_n(f) = t'$ and $P \vdash t \leq t'$. By induction, $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_2] : t''$, where $P \vdash t'' \leq t$, so $P \vdash t'' \leq t'$ by transitivity of subtyping. DTFldAss then yields $P, \Gamma, \sigma \vdash e_3\texttt{.}f \texttt{ = } \mathcal{E}'_{\mathrm{e}}[e_2] : t''$ as required.

Case 12: Derivation ends with DTRelAdd
Then $\mathcal{E}[e_1] = r\texttt{.add(}e_3\texttt{,}e_4\texttt{)}$ so $\mathcal{E} = r\texttt{.add(}\mathcal{E}'_{\mathrm{e}}\texttt{,}e_4\texttt{)}$ or $e_3 \in \mathsf{DynValue}$ and $\mathcal{E} = r\texttt{.add(}e_3\texttt{,}\mathcal{E}'_{\mathrm{e}}\texttt{)}$.

Case 12.1: $\mathcal{E} = r\texttt{.add(}\mathcal{E}'_{\mathrm{e}}\texttt{,}e_4\texttt{)}$

By DTRelAdd, $\mathcal{R}(r) = (\_, n_1, n_2, \_, \_)$, $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_1] : n_3$, $P, \Gamma, \sigma \vdash e_4 : n_4$, $P \vdash n_3 \leq n_1$, $P \vdash n_4 \leq n_2$ and $t_3 = r$.

By induction, $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_2] : n_5$ and $P \vdash n_5 \leq n_3$, so $P \vdash n_5 \leq n_1$ by transitivity of subtyping. $P, \Gamma, \sigma \vdash r\,\texttt{.add(}\mathcal{E}'_{\mathrm{e}}[e_2]\texttt{,}e_4\texttt{)} : r$ follows by DTRelAdd where $t_3 = r = t_4$ as required.

Case 12.2: $\mathcal{E} = r\,\texttt{.add(}e_3\texttt{,}\mathcal{E}'_{\mathrm{e}}\texttt{)}$

By DTRelAdd, $\mathcal{R}(r) = (\_, n_1, n_2, \_, \_)$, $P, \Gamma, \sigma \vdash e_3 : n_3$, $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_1] : n_4$, $P \vdash n_3 \leq n_1$, $P \vdash n_4 \leq n_2$ and $t_3 = r$.

By induction, $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_2] : n_5$, where $P \vdash n_5 \leq n_4$ so $P \vdash n_5 \leq n_2$ by transitivity of subtyping. $P, \Gamma, \sigma \vdash r\,\texttt{.add(}e_3\texttt{,}\mathcal{E}'_{\mathrm{e}}[e_2]\texttt{)} : r$ follows by DTRelAdd where $t_3 = r = t_4$ as required.

Case 13: Derivation ends with DTRelRem

Then $\mathcal{E}[e_1] = r\,\texttt{.rem(}e_3\texttt{,}e_4\texttt{)}$ and $\mathcal{E} = r\,\texttt{.rem(}\mathcal{E}'_{\mathrm{e}}\texttt{,}e_4\texttt{)}$ or $e_3 \in \mathsf{DynValue}$ and $\mathcal{E} = r\,\texttt{.rem(}e_3\texttt{,}\mathcal{E}'_{\mathrm{e}}\texttt{)}$.

Case 13.1: $\mathcal{E} = r\,\texttt{.rem(}\mathcal{E}'_{\mathrm{e}}\texttt{,}e_4\texttt{)}$

By DTRelRem, $\mathcal{R}(r) = (\_, n_1, n_2, \_, \_)$, $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_1] : n_3$, $P, \Gamma, \sigma \vdash e_4 : n_4$, $P \vdash n_3 \leq n_1$, $P \vdash n_4 \leq n_2$ and $t_3 = r$.

By induction, $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_2] : n_5$ and $P \vdash n_5 \leq n_3$, so $P \vdash n_5 \leq n_1$ by transitivity of subtyping. $P, \Gamma, \sigma \vdash r\,\texttt{.rem(}\mathcal{E}'_{\mathrm{e}}[e_2]\texttt{,}e_4\texttt{)} : r$ then follows by DTRelRem where $t_3 = r = t_4$ as required.

Case 13.2: $\mathcal{E} = r\,\texttt{.rem(}e_3\texttt{,}\mathcal{E}'_{\mathrm{e}}\texttt{)}$

By DTRelRem, $\mathcal{R}(r) = (\_, n_1, n_2, \_, \_)$, $P, \Gamma, \sigma \vdash e_3 : n_3$, $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_1] : n_4$, $P \vdash n_3 \leq n_1$, $P \vdash n_4 \leq n_2$ and $t_3 = r$.

By induction, $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_2] : n_5$ and $P \vdash n_5 \leq n_4$, so $P \vdash n_5 \leq n_2$ by transitivity of subtyping. $P, \Gamma, \sigma \vdash r\,\texttt{.rem(}e_3\texttt{,}\mathcal{E}'_{\mathrm{e}}[e_2]\texttt{)} : r$ then follows by DTRelRem where $t_3 = r = t_4$ as required.

Case 14: Derivation ends with DTCall

Then $\mathcal{E}[e_1] = e_3\,\texttt{.}m\texttt{(}e_4\texttt{)}$ so $\mathcal{E} = \mathcal{E}'_{\mathrm{e}}\,\texttt{.}m\texttt{(}e_4\texttt{)}$ or $e_3 \in \mathsf{DynValue}$ and $\mathcal{E} = e_3\,\texttt{.}m\texttt{(}\mathcal{E}'_{\mathrm{e}}\texttt{)}$.

Case 14.1: $\mathcal{E} = \mathcal{E}'_{\mathrm{e}}\,\texttt{.}m\texttt{(}e_4\texttt{)}$

By DTCall, $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_1] : n$, $P, \Gamma, \sigma \vdash e_4 : t$, $\mathcal{MD}_n(m) = (x, \_, t', t_3, \_)$ and $P \vdash t \leq t'$.

By induction, $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_2] : n'$ where $P \vdash n' \leq n$. By Lemma 5, $\mathcal{MD}_{n'}(m) = (x, \_, t'', t_4, \_)$ and $P \vdash t' \leq t''$ and $P \vdash t_4 \leq t_3$. By transitivity of subtyping, $P \vdash t \leq t''$, so $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_2]\,\texttt{.}m\texttt{(}e_4\texttt{)} : t_4$ then follows by DTCall, with $P \vdash t_4 \leq t_3$ as required.

Case 14.2: $\mathcal{E} = e_3\,\texttt{.}m\texttt{(}\mathcal{E}'_{\mathrm{e}}\texttt{)}$

By DTCall, $P, \Gamma, \sigma \vdash e_3 : n$, $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_1] : t$, $\mathcal{MD}_n(m) = (x, \_, t', t_3, \_)$ and $P \vdash t \leq t'$.

By induction $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_2] : t''$ where $P \vdash t'' \leq t$. By transitivity of subtyping, $P \vdash t'' \leq t'$. $P, \Gamma, \sigma \vdash e_3\,\texttt{.}m\texttt{(}\mathcal{E}'_{\mathrm{e}}[e_2]\texttt{)} : t_3$ then follows by DTCall.

Case 15: Derivation ends with DTMethBody

Then $\mathcal{E}[e_1] = \{\ s\ \texttt{return}\ e\texttt{;}\ \}$ and $\mathcal{E} = \{\ \mathcal{E}_{\mathrm{s}}\ \texttt{return}\ e\texttt{;}\ \}$ or $\mathcal{E} = \{\ \texttt{return}\ \mathcal{E}'_{\mathrm{e}}\texttt{;}\ \}$.

Case 15.1: $\mathcal{E} = \{\ \mathcal{E}'_{\mathrm{s}}\ \texttt{return}\ e;\ \}$

Then by DTMethBody, $P, \Gamma, \sigma \vdash e : t_3$ and $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{s}}[e_1]$. By induction, $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{s}}[e_2]$, so $P, \Gamma, \sigma \vdash \mathcal{E}[e_2] : t_3$ by DTMethBody as required.

Case 15.2: $\mathcal{E} = \{\ \texttt{return}\ \mathcal{E}'_{\mathrm{e}};\ \}$

Then by DTMethBody, $P, \Gamma, \sigma \vdash \epsilon$ and $P, \Gamma, \sigma \vdash \mathcal{E}_{\mathrm{e}}[']e_1 : t_3$. By induction, $P, \Gamma, \sigma \vdash \mathcal{E}_{\mathrm{e}}[']e_2 : t_4$ and $P \vdash t_4 \leq t_3$. By DTMethBody, $P, \Gamma, \sigma \vdash \mathcal{E}[e_2] : t_4$ as required.

Case 16: Derivation ends with DTSkip

This case does not arise, as ϵ is neither an expression to be substituted in $\mathcal{E} = \bullet$, nor is it a context.

Case 17: Derivation ends with DTFor

Then $\mathcal{E}[e_1] = \texttt{for}\ (n\ x : e_3)\ \{s_1\};\ s_2$ and $\mathcal{E} = \texttt{for}\ (n\ x : \mathcal{E}'_{\mathrm{e}})\ \{s_1\};\ s_2$.

By DTFor, $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_1] : \texttt{set<}n_2\texttt{>}$, $P, \Gamma[x \mapsto n_1], \sigma \vdash s_1$, $P, \Gamma, \sigma \vdash s_2$, $x \notin \mathsf{dom}(\Gamma)$ and $P \vdash n_2 \leq n_1$.

By induction, $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_2] : t'$ where $P \vdash t' \leq \texttt{set<}n_2\texttt{>}$. By Lemma 3(i), $t' = \texttt{set<}n'\texttt{>}$ where $P \vdash n' \leq n_2$, so $P \vdash n' \leq n_1$ by transitivity of subtyping. Therefore, by DTFor, $P, \Gamma, \sigma \vdash \texttt{for}\ (n_1\ x : \mathcal{E}'_{\mathrm{e}}[e_2])\ \{s_1\};\ s_2$ as required.

Case 18: Derivation ends with DTCond

Then $\mathcal{E}[e_1] = \texttt{if}\ (e_1)\ \{s_1\}\ \texttt{else}\ \{s_2\};\ s_3$ and $\mathcal{E} = \texttt{if}\ (\mathcal{E}'_{\mathrm{e}})\ \{s_1\}\ \texttt{else}\ \{s_2\};\ s_3$.

By DTCond, $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_1] : \texttt{boolean}$, $P, \Gamma, \sigma \vdash s_1$, $P, \Gamma, \sigma \vdash s_2$ and $P, \Gamma, \sigma \vdash s_3$.

By induction, $P, \Gamma, \sigma \vdash \mathcal{E}'_{\mathrm{e}}[e_2] : t'$ where $P \vdash t' \leq \texttt{boolean}$. By Lemma 3(j) $t' = \texttt{boolean}$. Finally, DTCond yields $P, \Gamma, \sigma \vdash \texttt{if}\ (\mathcal{E}_{\mathrm{e}}[']e_2)\ \{s_1\}\ \texttt{else}\ \{s_2\};\ s_3$ as required.

□

Lemma 36 (Typable sub-expressions). *If* $P, \Gamma, \sigma \vdash \mathcal{E}[e] : t$ *or* $P, \Gamma, \sigma \vdash \mathcal{E}[e]$ *then* $P, \Gamma, \sigma \vdash e : t'$.

Proof sketch By induction on the structure of the typing derivation. For example, in the case where the derivation of $P, \Gamma, \sigma \vdash \mathcal{E}[e] : t$ ends with DTFld, then either:

Case 1: $\mathcal{E} = \bullet$

In which case $P, \Gamma, \sigma \vdash e : t$ as required by assumption.

Case 2: $\mathcal{E} = \mathcal{E}_{\mathrm{e}}\,.\,f$

In which case $P, \Gamma, \sigma \vdash \mathcal{E}_{\mathrm{e}}[e] : t''$ by DTFld. By induction, $P, \Gamma, \sigma \vdash e : t'$ as required.

All other cases proceed similarly. □

Lemma 37. *All object dynamic types are subtypes of* `Object`*:*
$P \vdash \sigma \diamond_{\mathsf{heap}} \wedge \sigma(\iota) = o \Rightarrow \vdash \mathsf{dynType}(o) \leq \texttt{Object}$

Proof As $P \vdash \sigma \diamond_{\mathsf{heap}}$, then $P, \sigma \vdash o \diamond_{\mathsf{inst}}$. If by WFObject1, then $\mathsf{dynType}(o) = \texttt{Object}$ and the result follows by reflexivity. If by WFObject2, then $\mathsf{dom}(o) = \mathsf{dom}(\mathcal{F}_{P,c})$ where $c = \mathsf{dynType}(o)$, and by definition of $\mathcal{F}_{P,c}$, $c \in \mathsf{dom}(\mathcal{C}_P)$. By Lemma 3(a), $\vdash c \leq \texttt{Object}$ as required. Similarly for WFRelInst1 and WFRelInst2 respectively, with the addition that $\vdash \texttt{Relation} \leq \texttt{Object}$ by STObject. □

Lemma 38 (Subsumption for values). $P, \Gamma, \sigma \vdash u : t \ \wedge\ P \vdash t \leq t' \Rightarrow P, \Gamma, \sigma \vdash u : t'$

Proof By induction on the derivation of $P, \Gamma, \sigma \vdash u : t$.

Case 1: Derivation ends DTAddr
Then $P \vdash \mathsf{dynType}(\sigma(u)) \leq t$. By transitivity of subtyping, $P \vdash \mathsf{dynType}(\sigma(u)) \leq t'$, so $P, \Gamma, \sigma \vdash u : t'$ as required.

Case 2: Derivation ends DTSet
Then $t = \texttt{set<}n\texttt{>}$ and $\forall \iota \in u : P, \Gamma, \sigma \vdash \iota : n$. By Lemma 3(i), $t' = \texttt{set<}n'\texttt{>}$. By STCov, $P \vdash n \leq n'$. By induction, $\forall \iota \in u : P, \Gamma, \sigma \vdash \iota : n'$. By DTSet, $P, \Gamma, \sigma \vdash u : t'$ as required.

Case 3: Derivation ends DTBoolF/DTBoolT
Then $u \in \{\texttt{true}, \texttt{false}\}$. $t' = \texttt{boolean}$ by Lemma 3(j). The result follows trivially.

Case 4: Derivation ends DTNull
Then $t = n$, so $t' = n'$ by Lemma 3(g). The result follows trivially.

□

Lemma 39 (Typing preserved by renaming).
Consistent variable freshening preserves typing. If $P, \Gamma, \sigma \vdash e : t$ and $x' \notin \mathit{dom}(\Gamma)$ then $P, \Gamma[x' \mapsto \Gamma(x)], \sigma \vdash e[x'/x] : t$. Similarly for statements.

Proof The proof is by straightforward induction on the derivation of the typing relation, and is omitted. □

Lemma 40 (Typing preserved under statement concatenation).
If $P, \Gamma, \sigma \vdash s_1$ and $P, \Gamma, \sigma \vdash s_2$ then $P, \Gamma, \sigma \vdash s_1 s_2$.

Proof Induction on the structure of the derivation of $P, \Gamma, \sigma \vdash s_1$.
If the derivation ends in DTSkip, then $s_1 = \epsilon$ and $s_1 s_2 = s_2$ and the result follows trivially.
If the derivation ends in DTFor, then $s_1 = \texttt{for (}n\ x\ \texttt{:}\ e\texttt{)}\ \{s_3\}\texttt{;}\ s_4$, $P, \Gamma[x \mapsto n], \sigma \vdash s_3$ and $P, \Gamma, \sigma \vdash s_4$. By induction, $P, \Gamma, \sigma \vdash s_4 s_2$. By DTFor, then $P, \Gamma, \sigma \vdash \texttt{for (}n\ x\ \texttt{:}\ e\texttt{)}\ \{s_3\}\texttt{;}\ s_4\ s_2$ as required.
Similarly for derivations ending in DTCond. □

Theorem 41 (Subject Reduction).
If:
- $P \vdash \sigma_1 \diamond_{\mathit{heap}}$
- $P, \sigma_1 \vdash \rho_1 \diamond_{\mathit{relheap}}$
- $P, \Gamma_1, \sigma \vdash \lambda_1 \diamond_{\mathit{locals}}$
- $P, \Gamma_1, \sigma_1 \vdash R : t_1$ *or* $P, \Gamma_1, \sigma_1 \vdash R$
- $\langle \Gamma_1, \sigma_1, \rho_1, \lambda_1, R \rangle \overset{P}{\rightsquigarrow} \langle \Gamma_2, \sigma_2, \rho_2, \lambda_2, R' \rangle$ *or* $\langle \Gamma_2, \sigma_2, \rho_2, \lambda_2, w \rangle$

then:

(a) $P \vdash \sigma_2 \diamond_{\mathit{heap}}$ *and*

(b) $P, \sigma_2 \vdash \rho_2 \diamond_{\mathit{relheap}}$ *and*

(c) $P, \Gamma_2, \sigma_2 \vdash \lambda_2 \diamond_{\mathit{locals}}$ *and*

(d) $\Gamma_1 \subseteq \Gamma_2$ *and*

(e) $\mathit{dynType}(\sigma_1(\iota)) = t \Rightarrow \mathit{dynType}(\sigma_2(\iota)) = t$ *and*

(f) where execution is not to an error:

$$\begin{aligned} P, \Gamma_1, \sigma_1 \vdash R_1 : t_1 \quad &\Rightarrow \quad P, \Gamma_2, \sigma_2 \vdash R_2 : t_2 \text{ and } P \vdash t_2 \leq t_1 \\ P, \Gamma_1, \sigma_1 \vdash R_1 \quad &\Rightarrow \quad P, \Gamma_2, \sigma_2 \vdash R_2 \end{aligned}$$

Proof By induction on the structure of the derivation of the execution step. Notice that at most one of $P, \Gamma_1, \sigma_1 \vdash R_1 : t_1$ or $P, \Gamma_1, \sigma_1 \vdash R_1$ holds for any term, as R_1 is either an expression or a statement.

Case 1: Derivation ends with OSContextE

Then $R_1 = \mathcal{E}_e[e_3]$. By Lemma 36, $P, \Gamma_1, \sigma_1 \vdash e_3 : t_3$. By OSContextE, $\langle \Gamma_1, \sigma_1, \rho_1, \lambda_1, e_3 \rangle$ executes either to:

- $\overset{P}{\rightsquigarrow} \langle \Gamma_2, \sigma_2, \rho_2, \lambda_2, e_4 \rangle$ Then by induction $P \vdash \sigma_2 \diamond_{\mathsf{heap}}$, $P, \sigma_2 \vdash \rho_2 \diamond_{\mathsf{relheap}}$, $P, \Gamma_2, \sigma_2 \vdash \lambda_2 \diamond_{\mathsf{locals}}$, $\Gamma_1 \subseteq \Gamma_2$, $\mathsf{dynType}(\sigma_1(\iota)) = t \Rightarrow \mathsf{dynType}(\sigma_2(\iota)) = t$ and $P, \Gamma_2, \sigma_2 \vdash e_4 : t_4$ where $P \vdash t_4 \leq t_3$. By Lemma 35, $P, \Gamma_2, \sigma_2 \vdash \mathcal{E}_e[e_4] : t_2$ where $P \vdash t_2 \leq t_1$ as required.
- $\overset{P}{\rightsquigarrow} \langle \Gamma_2, \sigma_2, \rho_2, \lambda_2, w' \rangle$ Then by induction $P \vdash \sigma_2 \diamond_{\mathsf{heap}}$, $P, \sigma_2 \vdash \rho_2 \diamond_{\mathsf{relheap}}$, $P, \Gamma_2, \sigma_2 \vdash \lambda_2 \diamond_{\mathsf{locals}}$, $\Gamma_1 \subseteq \Gamma_2$ and $\mathsf{dynType}(\sigma_1(\iota)) = t \Rightarrow \mathsf{dynType}(\sigma_2(\iota)) = t$. By definition of Error, $w = \mathcal{E}_e[w']$ as required.

Case 2: Derivation ends with OSContextS

Then $R_1 = \mathcal{E}_s[e_3]$, and by Lemma 36, $P, \Gamma_1, \sigma_1 \vdash e_3 : t_3$.

Either $\langle \Gamma_1, \sigma_1, \rho_1, \lambda_1, R_1 \rangle$:

- $\overset{P}{\rightsquigarrow} \langle \Gamma_2, \sigma_2, \rho_2, \lambda_2, R_2 \rangle$ in which case $R_2 = \mathcal{E}_s[e_4]$ and $\langle \Gamma_1, \sigma_1, \rho_1, \lambda_1, e_4 \rangle \overset{P}{\rightsquigarrow} \langle \Gamma_2, \sigma_2, \rho_2, \lambda_2, e_4 \rangle$ by induction, where $P, \Gamma_2, \sigma_2 \vdash e_4 : t_4$ where $P \vdash t_4 \leq t_3$. By Lemma 35, $P, \Gamma_2, \sigma_2 \vdash \mathcal{E}_s[e_4]$ as required.
- $\overset{P}{\rightsquigarrow} \langle \Gamma_2, \sigma_2, \rho_2, \lambda_2, w \rangle$ in which case $w = \mathcal{E}_s[w']$. The remaining results follow by induction on $\langle \Gamma_1, \sigma_1, \rho_1, \lambda_1, e_3 \rangle \overset{P}{\rightsquigarrow} \langle \Gamma_2, \sigma_2, \rho_2, \lambda_2, w' \rangle$.

That $P \vdash \sigma_2 \diamond_{\mathsf{heap}}$, $P, \sigma_2 \vdash \rho_2 \diamond_{\mathsf{relheap}}$, $P, \Gamma_2, \sigma_2 \vdash \lambda_2 \diamond_{\mathsf{locals}}$, $\mathsf{dynType}(\sigma_1(\iota)) = t \Rightarrow \mathsf{dynType}(\sigma_2(\iota))$ and $\Gamma_1 \subseteq \Gamma_2$ follow by the use of induction in both cases.

Case 3: Derivation ends with OSInBody

Then $R_1 = \{\ s_1\ \texttt{return}\ e;\ \}$.

If R_1 executes to a new term, then $R_2 = \{\ s'\ \texttt{return}\ e;\ \}$ where $\langle \Gamma_1, \sigma_1, \rho_1, \lambda_1, s \rangle \overset{P}{\rightsquigarrow} \langle \Gamma_2, \sigma_2, \rho_2, \lambda_2, s_2 \rangle$. By induction, $P, \Gamma_2, \sigma_2 \vdash s_2$.

If R_1 executes to an exception, then $w = \{\ w'\ \texttt{return}\ e;\ \}$ where $\langle \Gamma_1, \sigma_1, \rho_1, \lambda_1, s_1 \rangle \overset{P}{\rightsquigarrow} \langle \Gamma_2, \sigma_2, \rho_2, \lambda_2, w' \rangle$.

In both cases, by induction, $P \vdash \sigma_2 \diamond_{\mathsf{heap}}$, $P, \sigma_2 \vdash \rho_2 \diamond_{\mathsf{relheap}}$, $P, \Gamma_2, \sigma_2 \vdash \lambda_2 \diamond_{\mathsf{locals}}$ $\Gamma_1 \subseteq \Gamma_2$ and the dynamic types of objects in σ_1 are preserved in σ_2.

Case 4: Derivation ends with OSEmpty

Then $R_1 = \texttt{empty}$, $R_2 = \emptyset$, $\Gamma_1 = \Gamma_2$, $\sigma_1 = \sigma_2$, $\rho_1 = \rho_2$, $\lambda_1 = \lambda_2$. By TSEmpty, $P, \Gamma, \sigma \vdash \texttt{empty} : n$ for some n. By DTSet, $P, \Gamma, \sigma \vdash \emptyset : n$, as there is no $\iota \in \emptyset$ for which the dynamic type must be checked. Γ_2, σ_2, ρ_2 and λ_2 are trivially well-formed by equality. Case complete.

Case 5: Derivation ends with OSNew

Then $R_1 = \texttt{new}\ c()$ and by DTNew $P, \Gamma_1, \sigma_1 \vdash R_1 : c$. $\sigma_2 = \sigma_1[\iota \mapsto \mathsf{new}_P(c)]$, where $\iota \notin \mathsf{dom}(\sigma_1)$, so $P \vdash \sigma_2 \diamond_{\mathsf{heap}}$ by Corollary 13. Old relationship instances are untouched by the update, so by Corollary 32, $P, \sigma_2 \vdash \rho_2 \diamond_{\mathsf{relheap}}$. But $\rho_2 = \rho_1$, so $P, \sigma_2 \vdash \rho_2 \diamond_{\mathsf{relheap}}$. $\lambda_2 = \lambda_1$ and $\Gamma_2 = \Gamma_1$ so $P, \Gamma_2, \sigma_2 \vdash \lambda_2 \diamond_{\mathsf{locals}}$.

Finally, $R_2 = \iota$ and $\mathsf{dynType}(\sigma_2(\iota)) = c$ by definition of new. Clearly, then $P, \Gamma_2, \sigma_2 \vdash R_2 : c$ as required. Case complete.

Case 6: Derivation ends with OSEq

Then $R_1 = u$ `==` u', $R_2 =$ `true` and $t_1 =$ `boolean`. By DTBoolT, $P, \Gamma_1, \sigma_1 \vdash$ `true` : `boolean` as required. $\Gamma_1 = \Gamma_2$, $\sigma_1 = \sigma_2$, $\rho_1 = \rho_2$, $\lambda_1 = \lambda_2$, and so are trivially well-formed. Case complete. Similarly where derivation ends with OSNeq.

Case 7: Derivation ends with OSBody

Then $R_1 =$ { `return` u; }, $R_2 = u$, $P, \Gamma_1, \sigma_1 \vdash$ { `return` u; } : t_1. By DTMethBody, $P, \Gamma_1, \sigma_1 \vdash u : t_1$. $\Gamma_1 = \Gamma_2$, $\sigma_1 = \sigma_2$, $\lambda_1 = \lambda_2$ and $\rho_1 = \rho_2$ so $P \vdash \sigma_2 \diamond_{\mathsf{heap}}$, $P, \sigma_2 \vdash \rho_2 \diamond_{\mathsf{relheap}}$, $P, \Gamma_2, \sigma_2 \vdash \lambda_2 \diamond_{\mathsf{locals}}$ and $P, \Gamma_2, \sigma_2 \vdash u : t_1$ as required.

Case 8: Derivation ends with OSVar

Then $R_1 = x$, $R_2 = \lambda(x)$, $t_1 = \Gamma(x)$. By WFLocals, $P, \Gamma, \sigma \vdash \lambda(x) : \Gamma(x)$, so $t_2 = \Gamma(x) = t_1$. $\Gamma_1 = \Gamma_2$, $\sigma_1 = \sigma_2$, $\rho_1 = \rho_2$, $\lambda_1 = \lambda_2$, and so are trivially well-formed. Case complete.

Case 9: Derivation ends with OSFldN

Then R_1 executes to $w = \mathsf{NullPtrError}$, where $\Gamma_1 = \Gamma_2$, $\sigma_1 = \sigma_2$, $\rho_1 = \rho_2$, $\lambda_1 = \lambda_2$ and all are well-formed as required. Similarly for derivations ending with OSRelObjN, OSRelInstN, OSFldAssN, OSFldAddN, OSFldSubN, OSRelAddN, OSRelSubN or OSCallN

Case 10: Derivation ends with OSFld

Then $R_1 = \iota.f$, $R_2 = \mathsf{fld}(\sigma, \iota, f)$. By DTFld, $P, \Gamma, \sigma \vdash \iota : n$ and $t_1 = \mathcal{FD}_{P,n}(f)$. By Lemma 27, $P, \Gamma, \sigma \vdash \mathsf{fld}(\sigma, \iota, f) : t_1$, so $t_1 = t_2$ as required. $\Gamma_1 = \Gamma_2$, $\sigma_1 = \sigma_2$, $\rho_1 = \rho_2$, $\lambda_1 = \lambda_2$, and so are trivially well-formed. Case complete.

Case 11: Derivation ends with OSRelObj

Then $R_1 = \iota.r$ and $R_2 = \{\iota' \mid \exists \iota'' : \rho_1(r, \iota, \iota') = \iota''\}$. By DTRelObj, $P, \Gamma_1, \sigma_1 \vdash R_1$: `set<`n_2`>` where $\mathcal{R}_P(r) = (\_, n_1, n_2, \_, \_)$, $P, \Gamma_1, \sigma_1 \vdash \iota : n_3$, and $\vdash n_3 \leq n_1$. Pick arbitrary $\iota' \in R_2$; then, $(r, \iota, \iota') \in \mathsf{dom}(\rho_1)$ and by WFRelHeap, $P, \sigma_1, \rho_1 \vdash (r, \iota, \iota') \diamond_{\mathsf{rel}}$. By WFRelation1 and WFRelation2, $\sigma_1(\rho_1(r, \iota, \iota')) = \langle\!\langle r, \iota, \iota', \_ \| \ldots \rangle\!\rangle$ so by WFHeap, $P, \sigma \vdash \langle\!\langle r, \iota, \iota', \_ \| \ldots \rangle\!\rangle \diamond_{\mathsf{inst}}$. If $P, \sigma \vdash \langle\!\langle r, \iota, \iota', \_ \| \ldots \rangle\!\rangle \diamond_{\mathsf{inst}}$ by WFRelInst1, then $r =$ `Relation` and $n_2 =$ `Object`. Also, $\iota' \in \mathsf{dom}(\sigma)$ so by Lemma 37, $\vdash \mathsf{dynType}(\sigma_1(\iota')) \leq$ `Object` and $P, \Gamma_1, \sigma_1 \vdash \iota' : n_2$. Otherwise, if $P, \sigma \vdash \langle\!\langle r, \iota, \iota', \_ \| \ldots \rangle\!\rangle \diamond_{\mathsf{inst}}$ by WFRelInst2 then $\vdash \mathsf{dynType}(\sigma_1(\iota_1)) \leq n_2$ and $P, \Gamma_1, \sigma_1 \vdash \iota' : n_2$. In both cases, therefore, by DTSet and that $\iota' \in R_2$ was chosen arbitrarily, $P, \Gamma_1, \sigma_1 \vdash R_2$: `set<`n_2`>`. $\Gamma_1 = \Gamma_2$, $\sigma_1 = \sigma_2$, $\rho_1 = \rho_2$, $\lambda_1 = \lambda_2$, which are trivially well-formed, and $P, \Gamma_2, \sigma_2 \vdash R_2$: `set<`r`>` as required. Case complete.

Case 12: Derivation ends with OSRelInst

Then $R_1 = \iota{:}r$ and $R_2 = \{\iota'' \mid \exists \iota' : \rho_1(r, \iota, \iota') = \iota''\}$. By DTRelInst, $P, \Gamma_1, \sigma_1 \vdash R_1$: `set<`r`>`. Take arbitrary $\iota' \in R_2$. Then for some ι', $\rho_1(r, \iota, \iota') = \iota''$, and by WFRelHeap, $P, \sigma_1, \rho_1 \vdash (r, \iota, \iota') \diamond_{\mathsf{rel}}$. By WFRelation1 and WFRelation2, $\mathsf{dynType}(\sigma(\iota'')) = r$ so $\vdash \mathsf{dynType}(\sigma(\iota'')) \leq r$ by reflexivity. Therefore $P, \Gamma_1, \sigma_1 \vdash \iota'' : r$ for arbitrary $\iota'' \in e_2$, so $P, \Gamma_1, \sigma_1 \vdash R_2$: `set<`r`>`. $\Gamma_1 = \Gamma_2$, $\sigma_1 = \sigma_2$, $\rho_1 = \rho_2$, $\lambda_1 = \lambda_2$, which are trivially well-formed, and $P, \Gamma_2, \sigma_2 \vdash R_2$: `set<`r`>` as required. Case complete.

Case 13: Derivation ends with OSTo

Then $R_1 = u.$`to`, $R_2 = \iota$ where $\sigma_1(u) = \langle\!\langle \_, \_, \_, \iota \| \ldots \rangle\!\rangle$, and $P, \Gamma_1, \sigma_1 \vdash R_1 : n$ where $\mathcal{R}_P(r) = (\_, \_, n, \_, \_)$ and $P, \Gamma_1, \sigma_1 \vdash u : r$ by DTTo. Therefore, $\vdash r' = \mathsf{dynType}(\sigma_1(u)) \leq r$ by DTAddr, and $u \in \mathsf{dom}(\sigma_1)$.

If $r' =$ `Relation`, then $r =$ `Relation`, $n =$ `Object` and $\sigma_1(u) = \langle\!\langle r, \_, \_, \iota \| \ldots \rangle\!\rangle$. By WFRelInst1, $\iota \in \mathsf{dom}(\sigma_1)$, so $P, \Gamma_1, \sigma_1 \vdash \iota :$ `Object` $= n$ as required.

If $r' \neq \texttt{Relation}$, then $\sigma(u) = \langle\!\langle r', \_, \_, \iota \| \ldots \rangle\!\rangle$ and $\vdash \mathsf{dynType}(\sigma_1(\iota)) \leq n'$ where $\mathcal{R}_P(r') = (\_, \_, n', \_, \_)$ by WFRelInst2. $\vdash n' \leq n$ by WTRelationship, so $\vdash \mathsf{dynType}(\sigma_1(\iota)) \leq n$ by transitivity, and $P, \Gamma_1, \sigma_1 \vdash \iota : n$ by DTAddr.

In both cases, then, $P, \Gamma_1, \sigma_1 \vdash R_2 : n$. As $\Gamma_1 = \Gamma_2$, $\sigma_1 = \sigma_2$, $\rho_1 = \rho_2$ and $\lambda_1 = \lambda_2$, $P \vdash \sigma_2 \diamond_{\mathsf{heap}}$, $P, \sigma_2 \vdash \rho_2 \diamond_{\mathsf{relheap}}$, $P, \Gamma_2, \sigma_2 \vdash \lambda_2 \diamond_{\mathsf{locals}}$ and $P, \Gamma_2, \sigma_2 \vdash R_2 : n$ as required.

Similarly the case for OSFrom.

Case 14: Derivation ends with OSAss

Then $R_1 = x \texttt{ = } u$ and $P, \Gamma_1, \sigma_1 \vdash R_1 : t_2$ where $P, \Gamma_1, \sigma_1 \vdash x : t_3$, $P, \Gamma_1, \sigma_1 \vdash u : t_2$, $\vdash t_2 \leq t_3$ and $x \neq \texttt{this}$.

$\lambda_2 = \lambda_1[x \mapsto u]$. $\Gamma(x) = t_3$ by typing of x, so by Lemma 33, $P, \Gamma_1, \sigma_1 \vdash \lambda_2 \diamond_{\mathsf{locals}}$. $\Gamma_1 = \Gamma_2$ and $\sigma_1 = \sigma_2$, so $P \vdash \sigma_2 \diamond_{\mathsf{heap}}$ and $P, \Gamma_2, \sigma_2 \vdash \lambda_2 \diamond_{\mathsf{locals}}$. As $\rho_1 = \rho_2$, ρ_2 and σ_2 are trivially well-formed.

$e_2 = u$, and $P, \Gamma_2, \sigma_2 \vdash R_2 : t_2$ as required.

Case 15: Derivation ends with OSFldAss

Then $R_1 = u\texttt{.}f \texttt{ = } u'$ and $R_2 = u'$. $P, \Gamma_1, \sigma_1 \vdash R_1 : t_1$ where $P, \Gamma_1, \sigma_1 \vdash u : n_1$, $P, \Gamma_1, \sigma_1 \vdash u' : t_1$, $\mathcal{FD}_{P,n_1}(f) = t_2$ and $\vdash t_1 \leq t_2$.

$\sigma_2 = \mathsf{fldUpd}(\sigma_1, \iota_1, f, u')$, so $P \vdash \sigma_2 \diamond_{\mathsf{heap}}$ by Lemma 28. Clearly the dynamic types of all objects remain the same (as observed in Corollary 24), so by Lemma 29, $P, \Gamma_1, \sigma_2 \vdash u' : t_1$. As $\Gamma_1 = \Gamma_2$, $P, \Gamma_2, \sigma_2 \vdash R_2 : t_1$ as required. Furthermore, $\lambda_1 = \lambda_2$, so $P, \Gamma_2, \sigma_1 \vdash \lambda_2 \diamond_{\mathsf{locals}}$, but as $P, \Gamma_2, \sigma_1 \vdash \lambda_2(x) : \Gamma_2(x) \Rightarrow P, \Gamma_2, \sigma_2 \vdash \lambda_2(x) : \Gamma_2(x)$ by Lemma 29, $P, \Gamma_2, \sigma_1 \vdash \lambda_2 \diamond_{\mathsf{locals}}$.

Clearly field update does not affect the `.to` or `.from` pseudo-fields of relationship instances (as `from` and `to` are excluded from FldName), so $P, \sigma_2 \vdash \rho_1 \diamond_{\mathsf{relheap}}$, where $\rho_2 = \rho_1$ and is therefore trivially well-formed. Case complete.

Case 16: Derivation ends with OSAdd

Then $R_1 = u \texttt{ + } \iota$, $R_2 = (u \cup \iota)$, $\Gamma_1 = \Gamma_2$, $\sigma_1 = \sigma_2$, $\rho_1 = \rho_2$, $\lambda_1 = \lambda_2$. Clearly σ_2, ρ_2 and λ_2 are all well-formed with respect to one another and to Γ_2. By DTAdd, $P, \Gamma_1, \sigma_1 \vdash u : \texttt{set<}n_1\texttt{>}$, $P, \Gamma_1, \sigma_1 \vdash \iota : n_2$, $P \vdash n_1 \leq n_3$, $P \vdash n_2 \leq n_3$ and $t_1 = \texttt{set<}n_3\texttt{>}$.

By DTSet, $\forall \iota' \in u : P, \Gamma_1, \sigma_1 \vdash \iota' : n_1$. As $P \vdash n_1 \leq n_3$, by Lemma 38 $\forall \iota' \in u : P, \Gamma_1, \sigma_1 \vdash \iota' : n_3$. Similarly, $P, \Gamma_1, \sigma_1 \vdash \iota : n_3$. The union, therefore may be typed $P, \Gamma_1, \sigma_1 \vdash (u \cup \iota) : \texttt{set<}n_3\texttt{>}$ by DTSet. The required result (with $t_2 = \texttt{set<}n_3\texttt{>} = t_1$) then follows by $(P, \Gamma_1, \sigma_1) = (P, \Gamma_2, \sigma_2)$.

Case 17: Derivation ends with OSSub

As above (the case is trivial where $\iota \notin u$).

Case 18: Derivation ends with OSRelAdd

Then $R_1 = r\texttt{.add(}\iota_1\texttt{,}\iota_2\texttt{)}$, $R_2 = \rho_2(r, \iota_1, \iota_2)$ and $(\sigma_2, \rho_2) = \mathsf{addRel}_P(r, \iota_1, \iota_2, \sigma_1, \rho_1)$. By DTRelAdd, $P, \Gamma_1, \sigma_1 \vdash e_1 : r$, where $\mathcal{R}_P(r) = (\_, n_1, n_2, \_, \_)$, $P, \Gamma_1, \sigma_1 \vdash \iota_1 : n_3$, $P, \Gamma_1, \sigma_1 \vdash \iota_2 : n_4$, $\vdash n_3 \leq n_1$ and $\vdash n_4 \leq n_2$. By DTAddr, $\vdash \mathsf{dynType}(\sigma_1(\iota_1)) \leq n_3$, and by transitivity $\vdash \mathsf{dynType}(\sigma_1(\iota_1)) \leq n_1$. Similarly for ι_2 so that $\vdash \mathsf{dynType}(\sigma_1(\iota_2)) \leq n_2$. By Lemma 18, $P \vdash \sigma_2 \diamond_{\mathsf{heap}}$ and $P, \sigma_2 \vdash \rho_2 \diamond_{\mathsf{relheap}}$.

Also by Lemma 18, $(r, \iota_1, \iota_2) \in \mathsf{dom}(\rho_2)$, so $P, \sigma_2, \rho_2 \vdash (r, \iota_1, \iota_2) \diamond_{\mathsf{rel}}$. By WFRelation1 and WFRelation2, $\mathsf{dynType}(\sigma_2(\rho_2(r, \iota_1, \iota_2))) = r$, so $\vdash \mathsf{dynType}(\sigma_2(\rho_2(r, \iota_1, \iota_2))) \leq r$ by reflexivity and $P, \Gamma_2, \sigma_2 \vdash R_2 = \rho_2(r, \iota_1, \iota_2) : r$ as required.

Finally, $\Gamma_1 = \Gamma_2$ and $\lambda_1 = \lambda_2$, so $P, \Gamma_2, \sigma_1 \vdash \lambda_2 \diamond_{\mathsf{locals}}$. By Lemma 29, $P, \Gamma_2, \sigma_2 \vdash \lambda_2(x) : \Gamma_2(x)$, so $P, \Gamma_2, \sigma_2 \vdash \lambda_2 \diamond_{\mathsf{locals}}$.

Case 19: Derivation ends with OSRelRem1

Then $R_1 = r.\texttt{rem}(\iota_1, \iota_2)$, $R_2 = \iota_2$ and $\rho_2 = \mathsf{remRel}_P(r, \iota_1, \iota_2, \rho_1)$. By DTRelSub, $P, \Gamma_1, \sigma_1 \vdash r.\texttt{rem}(\iota_1, \iota_2) : n_4$, and $P, \Gamma_1, \sigma_1 \vdash \iota_2 : n_4$.

By Lemma 19, $P, \sigma_1 \vdash \rho_2 \diamond_{\mathsf{relheap}}$. $\sigma_2 = \sigma_1$ so $P, \sigma_2 \vdash \rho_2 \diamond_{\mathsf{relheap}}$ and $P \vdash \sigma_2 \diamond_{\mathsf{heap}}$. $\Gamma_2 = \Gamma_1$ and $\lambda_2 = \lambda_1$ so $P, \Gamma_2, \sigma_2 \vdash \lambda_2 \diamond_{\mathsf{locals}}$.

Furthermore, by $\sigma_2 = \sigma_1$ and $\Gamma_2 = \Gamma_1$, $P, \Gamma_2, \sigma_2 \vdash \iota_2 : n_4$ as required.

Case 20: Derivation ends in RelRem2

Then proof is as above, except that $P, \Gamma_2, \sigma_2 \vdash e_2 = \texttt{null} : r$ by DTNull, as required.

Case 21: Derivation ends with OSCall

Then $R_1 = \iota.m(u)$. By OSCall:

1: $\mathsf{dynType}(\sigma_1(\iota)) = c$
2: $\mathcal{MD}_{P,n_1}(m) = (x, \mathcal{L}, t_2, t_3, \{\ s\ \texttt{return}\ e;\ \})$
3: $\mathsf{dom}(\mathcal{L}) = \{x_1, x_2, \ldots, x_i\}$
4: $x', x'_{\texttt{this}}, x_{1..i} \notin \mathsf{dom}(\lambda_1)$
5: $\mathcal{L}' = \{x'_{1..i} \mapsto \mathcal{L}(x_{1..i})\}$
6: $\Gamma_2 = \Gamma_1[x' \mapsto t_1][x'_{\texttt{this}} \mapsto c] \cup \mathcal{L}'$
7: $\lambda_2 = \lambda_1[x' \mapsto u][x'_{\texttt{this}} \mapsto \iota][x'_{1..i} \mapsto \mathsf{initial}(\mathcal{L}'(x'_{1..i}))]$
8: $s_2 = s'[x'/x][x'_{\texttt{this}}/\texttt{this}][x'_{1..i}/x_{1..i}]$
9: $e' = e[x'/x][x'_{\texttt{this}}/\texttt{this}][x'_{1..i}/x_{1..i}]$

By DTCall, then, $P, \Gamma_1, \sigma_1 \vdash R_1 : t_3$.

By WTMethod, $P, \{x \mapsto t_1, \texttt{this} \mapsto c\} \cup \mathcal{L} \vdash s$, so $P, \{x \mapsto t_1, \texttt{this} \mapsto c\} \cup \mathcal{L}, \sigma_1 \vdash s$. As $x', x'_{\texttt{this}}, x_{1..i} \notin \mathsf{dom}(\lambda_1)$, and hence $\notin \mathsf{dom}(\Gamma_1)$ by well-formedness of λ_1, then $P, \Gamma_2, \sigma_1 \vdash s'$ by repeated application of Lemma 39.

By similar use of WTMethod and Lemma 39, $P, \Gamma_2, \sigma_1 \vdash e' : t'_3$ is derived, where $\vdash t'_3 \leq t_3$.

Therefore, by DTMethBody, $P, \Gamma_2, \sigma_1 \vdash \{\ s'\ \texttt{return}\ e';\ \} : t'_3$. As $\sigma_2 = \sigma_1$, $P, \Gamma_2, \sigma_2 \vdash \{\ s'\ \texttt{return}\ e';\ \} : t'_3$, where $\vdash t'_3 \leq t_3$ as required.

$P \vdash \sigma_2 \diamond_{\mathsf{heap}}$ and $P, \sigma_2 \vdash \rho_2 \diamond_{\mathsf{relheap}}$ follow trivially as ρ_1 and σ_1 are unchanged. It remains to check that $P, \Gamma_2, \sigma_2 \vdash \lambda_2 \diamond_{\mathsf{locals}}$.

As all of $x', x'_{\texttt{this}}, x'_{1..i} \notin \mathsf{dom}(\lambda_1)$, and $\sigma_1 = \sigma_2$, all $x \in \mathsf{dom}(\Gamma_1)$ are such that $P, \Gamma_2, \sigma_2 \vdash \lambda_2(x) : \Gamma_2(x)$. For all new variables, $x'_{1..i}$, $\lambda_2(x'_j) = \mathsf{initial}(\Gamma_2(x'_j))$, so $P, \Gamma_2, \lambda_2 \vdash \lambda_2(x'_j) : \Gamma_2(x'_j)$ by Lemma 9. The new formal parameter variable x' is such that $\Gamma_2(x') = t_2$, but by DTCall, $P, \Gamma_1, \sigma_1 \vdash u : t'_2$ and $\vdash t'_2 \leq t_2$, where $\lambda_2(x') = u$. Therefore, $P, \Gamma_1, \sigma_1 \vdash \lambda_2(x') : \Gamma_2(x')$ and $P, \Gamma_2, \sigma_2 \vdash \lambda_2(x') : \Gamma_2(x')$ by $\sigma_1 = \sigma_2$ and the irrelevance of Γ to value typing. Finally, $\mathsf{dynType}(\sigma_2(\iota)) = c$ and $\Gamma_2(x'_{\texttt{this}}) = c$ and $\lambda_2(x'_{\texttt{this}}) = \iota$, so $P, \Gamma_2, \sigma_2 \vdash \lambda_2(x'_{\texttt{this}}) : \Gamma_2(x'_{\texttt{this}})$ by DTAddr.

Therefore, for all $x \in \mathsf{dom}(\Gamma_2)$, $P, \Gamma_2, \sigma_2 \vdash \lambda_2(x) : \Gamma_2(x)$, so conclude that $P, \Gamma_2, \sigma_2 \vdash \lambda_2 \diamond_{\mathsf{locals}}$ by WFLocals.

Finally, observe that $\Gamma_1 \subseteq \Gamma_2$.

Case 22: Derivation ends with OSStat

Then $R_1 = u;\ s_1$, $R_2 = s_1$. $\Gamma_1 = \Gamma_2$, $\sigma_1 = \sigma_2$, $\rho_1 = \rho_2$ and $\lambda_1 = \lambda_2$, so $P \vdash \sigma_2 \diamond_{\mathsf{heap}}$, $P, \sigma_2 \vdash \rho_2 \diamond_{\mathsf{relheap}}$, $P, \Gamma_2, \sigma_2 \vdash \lambda_2 \diamond_{\mathsf{locals}}$ immediately. By DTExp, $P, \Gamma_1, \sigma_1 \vdash s_1$, so $P, \Gamma_2, \sigma_2 \vdash s_1$ follows immediately.

Case 23: Derivation ends with OSCondT

Then $R_1 = \texttt{if (true) } \{s_1\} \texttt{ else } \{s_2\};\ s_3$ and $R_2 = s_1 s_3$. By DTCond, $P, \Gamma_1, \sigma_1 \vdash s_1$ and $P, \Gamma_1, \sigma_1 \vdash s_3$. But $\Gamma_1 = \Gamma_2$, $\sigma_1 = \sigma_2$ so $P, \Gamma_2, \sigma_2 \vdash s_1$ and $P, \Gamma_2, \sigma_2 \vdash s_3$. By Lemma 40, $P, \Gamma_2, \sigma_2 \vdash s_1 s_3$ as required. Additionally, as $\rho_1 = \rho_2$ and $\lambda_1 = \lambda_2$, $P \vdash \sigma_2 \diamond_{\mathsf{heap}}$, $P, \sigma_2 \vdash \rho_2 \diamond_{\mathsf{relheap}}$ and $P, \Gamma_2, \sigma_2 \vdash \lambda_2 \diamond_{\mathsf{locals}}$.

Similarly for derivations ending with OSCondF.

Case 24: Derivation ends with OSFor1

Then $R_1 = \texttt{for } (n\ x : \emptyset)\ \{s_1\};\ s_2$, $R_2 = s_2$ and $P, \Gamma_1, \sigma_1 \vdash s_2$. $\Gamma_1 = \Gamma_2$, $\sigma_1 = \sigma_2$, $\rho_1 = \rho_2$ and $\lambda_1 = \lambda_2$ so $P, \Gamma_2, \sigma_2 \vdash s_2$, $P \vdash \sigma_2 \diamond_{\mathsf{heap}}$, $P, \sigma_2 \vdash \rho_2 \diamond_{\mathsf{relheap}}$ and $P, \Gamma_2, \sigma_2 \vdash \lambda_2 \diamond_{\mathsf{locals}}$ immediately, as required.

Case 25: Derivation ends with OSFor2

Then $R_1 = \texttt{for } (n\ x : u)\ \{s_1\};\ s_2$ and $R_2 = s_1[x'/x]\ \texttt{for } (n\ x : u \setminus \iota)\ \{s_1\};\ s_2$, $\Gamma_2 = \Gamma_1[x' \mapsto n]$ and $\lambda_2 = \lambda_1[x' \mapsto \iota]$ where $\iota \in u$ and $x' \notin \mathsf{dom}(\Gamma_1)$.

By DTFor, $P, \Gamma_1, \sigma_1 \vdash u : \texttt{set<}n'\texttt{>}$, $P \vdash n' \leq n$, $P, \Gamma_1, \sigma_1 \vdash s_2$ and $P, \Gamma_1[x \mapsto n], \sigma_1 \vdash s_1$.

By DTSet, it is clear that $P, \Gamma_1, \sigma_1 \vdash (u \setminus \iota) : \texttt{set<}n'\texttt{>}$; by Lemma 34, $P, \Gamma_2, \sigma_1 \vdash (u \setminus \iota) : \texttt{set<}n'\texttt{>}$; as $\sigma_1 = \sigma_2$, $P, \Gamma_2, \sigma_2 \vdash (u \setminus \iota) : \texttt{set<}n'\texttt{>}$.

By Lemma 34, that $x \neq x'$ and that $\sigma_1 = \sigma_2$, conclude $P, \Gamma_2[x \mapsto n], \sigma_2 \vdash s_1$.

By Lemma 34, and that $\sigma_1 = \sigma_2$, conclude $P, \Gamma_2, \sigma_2 \vdash s_2$.

Therefore, $P, \Gamma_2, \sigma_2 \vdash \texttt{for } (n\ x : (u \setminus \iota))\ \{s_1\};\ s_2$.

By the consistent renaming of Lemma 39 and that $\sigma_1 = \sigma_2$, $P, \Gamma_2, \sigma_2 \vdash s_1[x'/x]$.

Finally, by Lemma 40, $P, \Gamma_2, \sigma_2 \vdash s_1[x'/x]\ \texttt{for } (n\ x : (u \setminus \iota))\ \{s_1\};\ s_2$ as required.

Clearly $P \vdash \sigma_2 \diamond_{\mathsf{heap}}$ and $P, \sigma_2 \vdash \rho_2 \diamond_{\mathsf{relheap}}$ as $\sigma_1 = \sigma_2$ and $\rho_1 = \rho_2$.

For variables $x \in \mathsf{dom}(\Gamma_1)$, $P, \Gamma_1, \sigma_1 \vdash \lambda_1(x) : \Gamma_1(x)$. As $\lambda_1(x) = \lambda_2(x)$ and $\Gamma_1(x) = \Gamma_2(x)$ then $P, \Gamma_2, \sigma_2 \vdash \lambda_2(x) : \Gamma_2(x)$. $P, \Gamma_1, \sigma_1 \vdash u : \texttt{set<}n\texttt{>}$ by DTFor, so $P, \Gamma_1, \sigma_1 \vdash \iota : n$ and $P, \Gamma_2, \sigma_2 \vdash \iota : n$ by Lemma 34. Therefore, recalling that $\lambda_2(x') = \iota$, conclude $P, \Gamma_2, \sigma_2 \vdash \lambda_2 \diamond_{\mathsf{locals}}$ as required, and $\Gamma_1 \subseteq \Gamma_2$.

□

Lemma 42 (Subterm Progress). *Progress in sub-terms can be lifted to enclosing terms:*

$$\langle \Gamma, \sigma, \rho, \lambda, e \rangle \overset{P}{\rightsquigarrow} \begin{cases} \langle \Gamma', \sigma', \rho', \lambda', e' \rangle \\ \langle \Gamma', \sigma', \rho', \lambda', w \rangle \end{cases} \quad \Rightarrow \quad \langle \Gamma, \sigma, \rho, \lambda, \mathcal{E}_{\mathrm{e}}[e] \rangle \overset{P}{\rightsquigarrow} \begin{cases} \langle \Gamma', \sigma', \rho', \lambda', \mathcal{E}_{\mathrm{e}}[e'] \rangle \\ \langle \Gamma', \sigma', \rho', \lambda', w' \rangle \end{cases}$$

Similarly for $\mathcal{E}_{\mathrm{s}}[e]$.

Proof If $\langle \Gamma, \sigma, \rho, \lambda, e \rangle$:

$\overset{P}{\rightsquigarrow} \langle \Gamma', \sigma', \rho', \lambda', e' \rangle$ Then by OSContextE, $\langle \Gamma, \sigma, \rho, \lambda, \mathcal{E}_{\mathrm{e}}[e] \rangle \overset{P}{\rightsquigarrow} \langle \Gamma', \sigma', \rho', \lambda', \mathcal{E}_{\mathrm{e}}[e'] \rangle$.

$\overset{P}{\rightsquigarrow} \langle \Gamma', \sigma', \rho', \lambda', e' \rangle$ Then by OSContextE, $\langle \Gamma', \sigma', \rho', \lambda', \mathcal{E}_{\mathrm{e}}[e] \rangle \overset{P}{\rightsquigarrow} \langle \Gamma', \sigma', \rho', \lambda', \mathcal{E}_{\mathrm{e}}[w] \rangle$. By definition of Error, $\mathcal{E}_{\mathrm{e}}[w] \in \mathsf{Error}$ as required.

Proof for $\mathcal{E}_{\mathrm{s}}[e]$ proceeds as above, but by OSContextS. □

Theorem 43 (Progress).

If

- $P \vdash \sigma \diamond_{\mathsf{heap}}$
- $P, \sigma \vdash \rho \diamond_{\mathsf{relheap}}$
- $P, \Gamma, \sigma \vdash \lambda \diamond_{\mathsf{locals}}$
- $P, \Gamma, \sigma \vdash R : t$ *or* $P, \Gamma, \sigma \vdash R$

then:

(a) $R \in \mathsf{DynValue}$, $R = \epsilon$, *or*

(b) $\langle \Gamma, \sigma, \rho, \lambda, R \rangle \overset{P}{\rightsquigarrow} \langle \Gamma', \sigma', \rho', \lambda', R' \rangle$, *or*

(c) $\langle \Gamma, \sigma, \rho, \lambda, R \rangle \overset{P}{\rightsquigarrow} \langle \Gamma', \sigma', \rho', \lambda', w \rangle$

Proof By induction over the structure of the derivation of $P, \Gamma_1, \sigma_1 \vdash e_1 : t_1$.

Case 1: Derivation ends in DTAddr, DTSet, DTNull, DTBoolF or DTBoolT
Then e is a value.

Case 2: Derivation ends in DTMethBody
Then $R = \{\ s\ \texttt{return}\ e_1\texttt{;}\ \}$.

If $s = \epsilon$ and $e_1 = u \in \mathsf{DynValue}$ then progress is made by OSBody to $u \in \mathsf{DynValue} \subset \mathsf{DynExpression}$ as required.

If $s = \epsilon$ and $e_1 \notin \mathsf{DynValue}$ then take $\mathcal{E}_{\mathrm{e}} = \{\ \texttt{return}\ \bullet\ \texttt{;}\ \}$ such that $e_1 = \mathcal{E}_{\mathrm{e}}[e_1]$. By DTMethBody, $P, \Gamma, \sigma \vdash e_1 : t$, and the result follows by the inductive hypothesis applied to e_1 and Lemma 42.

If $s \neq \epsilon$ then $P, \Gamma, \sigma \vdash s$. By induction $\langle \Gamma, \sigma, \rho, \lambda, s \rangle$:

$\overset{P}{\rightsquigarrow} \langle \Gamma', \sigma', \rho', \lambda', s' \rangle$ In which case, by OSInBody,

$$\langle \Gamma, \sigma, \rho, \lambda, \{\ s\ \texttt{return}\ e_1\texttt{;}\ \} \rangle \overset{P}{\rightsquigarrow} \langle \Gamma', \sigma', \rho', \lambda', \{\ s\ \texttt{return}\ e_1\texttt{;}\ \} \rangle$$

as required.

$\overset{P}{\rightsquigarrow} \langle \Gamma', \sigma', \rho', \lambda', w' \rangle$ In which case, by OSInBody,

$$\langle \Gamma, \sigma, \rho, \lambda, \{\ s\ \texttt{return}\ e_1\texttt{;}\ \} \rangle \overset{P}{\rightsquigarrow} \langle \Gamma', \sigma', \rho', \lambda', w \rangle$$

where $w = \{\ w'\ \texttt{return}\ e\texttt{;}\ \}$ as required.

Case 3: Derivation ends in DTVar
Then $\Gamma(x) = t$. By GoodLocals, $P, \Gamma, \sigma \vdash \lambda(x) : t$, so $x \in \mathsf{dom}(\lambda)$. Progress by OSVar to $\lambda(x) \in \mathsf{DynValue} \subset \mathsf{DynExpression}$ as required.

Case 4: Derivation ends in DTNew
Then progress is by OSNew to $\iota \in \mathsf{DynValue} \subset \mathsf{DynExpression}$ as required.

Case 5: Derivation ends in DTEq
Then $R = (e_1\ \texttt{==}\ e_2)$.

If $e_1, e_2 \in \mathsf{DynValue}$ and $e_1 = e_2$, then progress is made by OSEq to $\texttt{true} \in \mathsf{DynExpression}$ as required.

If $e_1, e_2 \in \mathsf{DynValue}$ and $e_1 \neq e_2$, then progress is made by OSNeq to $\texttt{false} \in \mathsf{DynExpression}$ as required.

If $e_1 \notin \mathsf{DynValue}$ then take $\mathcal{E}_e = (\bullet \texttt{ == } e_2)$ such that $R = \mathcal{E}_e[e_1]$. $P, \Gamma, \sigma \vdash e_1 : t_1$ by DTEq, so required result follows by inductive hypothesis on e_1 and Lemma 42.

If $e_1 \in \mathsf{DynValue}$ and $e_2 \notin \mathsf{DynValue}$, then take $\mathcal{E}_e = (e_1 \texttt{ == } \bullet)$ such that $R = \mathcal{E}_e[e_2]$. $P, \Gamma, \sigma \vdash e_2 : t_2$ by DTEq, so required result follows by inductive hypothesis on e_2 and Lemma 42.

Case 6: Derivation ends in DTFld
Then $R = e_1 \,.\, f$.

If $e_1 \in \mathsf{DynValue}$ then $e_1 = \texttt{null}$ (by DTNull) or $e_1 = \iota \in \mathsf{dom}(\sigma)$ where $P \vdash \mathsf{dynType}(\sigma_1(\iota)) \leq n$ (by DTAddr). If $e_1 = \texttt{null}$ then progress is by OSFldN, and $w = \mathsf{NullPtrError} \in \mathsf{Error}$ as required. If $e_1 = \iota \in \mathsf{dom}(\sigma)$ then, by Lemma 25, $e' = \mathsf{fld}(\sigma, \iota, f) \in \mathsf{DynValue} \subset \mathsf{Expression}$ as required.

Suppose $e_1 \notin \mathsf{DynValue}$, and take $\mathcal{E}_e = \bullet \,.\, f$ such that $R = \mathcal{E}_e[e_1]$. Required result follows by inductive hypothesis on e and Lemma 42.

Case 7: Derivation ends in DTAdd
Then $R = e_1 \texttt{ + } e_2$. By DTAdd, $P, \Gamma, \sigma \vdash e_1 : \texttt{set<}n_1\texttt{>}$ and $P, \Gamma, \sigma \vdash e_2 : n_2$.

If $e_1, e_2 \in \mathsf{DynValue}$, then by typing $e_1 \subset \mathsf{Address}$, $e_2 \in \mathsf{Address} \cup \{\texttt{null}\}$. Progress is then made by OSAdd to $e_1 \cup \{e_2\} \in \mathsf{DynValue} \subset \mathsf{DynExpression}$ or by OSAddN to $\mathsf{NullPtrError} \in \mathsf{Error}$ as required.

If $e_1 \notin \mathsf{DynValue}$ then take $\mathcal{E}_e = \bullet \texttt{ + } e_2$ such that $R = \mathcal{E}_e[e_1]$. Required result then follows by application of inductive hypothesis to e_1 and Lemma 42.

If $e_1 \in \mathsf{DynValue}$ and $e_2 \notin \mathsf{DynValue}$ then $\mathcal{E}_e = e_1 \texttt{ + } \bullet$ such that $R = \mathcal{E}_e[e_2]$. Required result then follows by application of inductive hypothesis to e_2 and Lemma 42.

Case 8: Derivation ends in DTSub
Similar to case for DTAdd.

Case 9: Derivation ends in DTRelObj
Then $R = e_1 \,.\, r$ and $P, \Gamma, \sigma \vdash e_1 : n_1$.

If $e_1 = u \in \mathsf{DynValue}$ then by typing, $e_1 \in \mathsf{Address}$ or $e_1 = \texttt{null}$. If $e_1 \in \mathsf{Address}$, then progress is made by OSRelObj to $\{\iota' \mid \exists \iota'' : \rho(r, e_1, \iota') = \iota''\} \subseteq \mathsf{Address} \subset \mathsf{DynValue} \subset \mathsf{DynExpression}$ as required. If $e_1 = \texttt{null}$ then progress is made by OSRelObjN to NullPtrError.

If $e_1 \notin \mathsf{DynValue}$ then take $\mathcal{E}_e = \bullet \,.\, r$ such that $R = \mathcal{E}_e[e_1]$. Result follows then by the inductive hypothesis on e_1 and Lemma 42.

Case 10: Derivation ends in DTRelInst
Then $R = e_1 \texttt{:} r$ and $P, \Gamma, \sigma \vdash e_1 : n_1$.

If $e_1 = u \in \mathsf{DynValue}$ then by typing, $e_1 \in \mathsf{Address}$ or $e_1 = \texttt{null}$. If $e_1 \in \mathsf{Address}$, then progress is made by OSRelInst to $\{\iota'' \mid \exists \iota' : \rho(r, e_1, \iota') = \iota''\} \subseteq \mathsf{Address} \subset \mathsf{DynValue} \subset \mathsf{DynExpression}$ as required. If $e_1 = \texttt{null}$ then progress is made by OSRelInstN to NullPtrError.

If $e_1 \notin \mathsf{DynValue}$ then take $\mathcal{E}_e = \bullet \texttt{:} r$ such that $R = \mathcal{E}_e[e_1]$. Result follows then by the inductive hypothesis on e_1 and Lemma 42.

Case 11: Derivation ends in DTFrom
Then $R = e_1 \texttt{.from}$ and $P, \Gamma, \sigma \vdash e_1 : r$.

If $e_1 = u \in \mathsf{DynValue}$ then by typing $e_1 \in \mathsf{Address}$ or $e_1 = \texttt{null}$. If $e_1 \in \mathsf{Address}$, then $P \vdash \mathsf{dynType}(\sigma(e_1)) \leq r$ by DTAddr. By WFRelInst1/2, $\sigma(\iota) = \langle\!\langle \mathsf{dynType}(\sigma(e_1)), \_, \iota', \_ \| \ldots \rangle\!\rangle$. Progress is made by OSFrom to $\iota' \in \mathsf{Address} \subset \mathsf{DynValue} \subset \mathsf{DynExpression}$ as required. Where $e_1 = \texttt{null}$ the progress is made by OSFromN to NullPtrError.

If $e_1 \notin \mathsf{DynValue}$ then take $\mathcal{E}_\mathrm{e} = \bullet\texttt{.from}$ such that $R = \mathcal{E}_\mathrm{e}[e_1]$. Required result follows from induction on e_1 and Lemma 42.

Case 12: Derivation ends in DTTo
Similar to the case for DTFrom.

Case 13: Derivation ends in DTAss
Then $R = (x \texttt{ = } e_1)$ and $P, \Gamma, \sigma \vdash e_1 : t_1$.

If $e_1 \in \mathsf{DynValue}$ then progress is made by OSVarAss to $u \in \mathsf{DynValue} \subset \mathsf{DynExpression}$ (under updated λ).

If $e_1 \notin \mathsf{DynValue}$ then take $\mathcal{E}_\mathrm{e} = x \texttt{ = } \bullet$ such that $R = \mathcal{E}_\mathrm{e}[e_1]$. Required result follows by induction on e_1 and Lemma 42.

Case 14: Derivation ends in DTFldAss
Then $R = e_1\texttt{.}f \texttt{ = } e_2$, $P, \Gamma, \sigma \vdash e_1 : n_1$ and $P, \Gamma, \sigma \vdash e_2 : t_2$.

If $e_1, e_2 \in \mathsf{DynValue}$ then by typing $e_1 \in \mathsf{Address}$ or $e_1 = \texttt{null}$. If $e_1 \in \mathsf{Address}$ then progress is made by OSFldAss to $u \in \mathsf{DynValue} \subset \mathsf{DynExpression}$ under new store $\mathsf{fldUpd}(\sigma, \iota, f, u)$, which is defined by Lemma 26. If $e_1 = \texttt{null}$ then progress is made by OSFldAssN to NullPtrError.

If $e_1 \notin \mathsf{DynValue}$ then take $\mathcal{E}_\mathrm{e} = \bullet\texttt{.}f \texttt{ = } e_2$ such that $R = \mathcal{E}_\mathrm{e}[e_1]$ and the result follows by induction on e_1 and Lemma 42.

If $e_1 \in \mathsf{DynValue}$ and $e_2 \notin \mathsf{DynValue}$, then take $\mathcal{E}_\mathrm{e} = e_1\texttt{.}f \texttt{ = } \bullet$ such that $e = \mathcal{E}_\mathrm{e}[e_2]$ and the result follows by induction on e_1 and Lemma 42.

Case 15: Derivation ends in DTRelAdd
Then $R = r\texttt{.add(}e_1\texttt{,}e_2\texttt{)}$, $P, \Gamma, \sigma \vdash e_1 : n_1$ and $P, \Gamma, \sigma \vdash e_2 : n_2$.

If $e_1, e_2 \in \mathsf{DynValue}$ then $e_1, e_2 \in \mathsf{Address} \cup \{\texttt{null}\}$. If either $e_1 = \texttt{null}$ or $e_2 = \texttt{null}$ then progress is made to NullPtrError by OSRelAddN. In the case where e_1 and e_2 are addresses, then progress is made by OSRelAdd to $\rho'(r, e_1, e_2)$ where $(\sigma', \rho') = \mathsf{addRel}_P(r, e_1, e_2, \sigma, \rho)$ and where $(r, e_1, e_2) \in \mathsf{dom}(\rho)$ by Lemma 18. Therefore $\rho'(r, e_1, e_2) \in \mathsf{Address} \subset \mathsf{DynValue} \subset \mathsf{DynExpression}$ as required.

If $e_1 \notin \mathsf{DynValue}$ then take $\mathcal{E}_\mathrm{e} = r\texttt{.add(}\bullet\texttt{,}e_2\texttt{)}$ such that $R = \mathcal{E}_\mathrm{e}[e_1]$ and the result follows by induction on e_1 and Lemma 42.

If $e_1 \in \mathsf{DynValue}$ and $e_2 \notin \mathsf{DynValue}$ then take $\mathcal{E}_\mathrm{e} = r\texttt{.add(}e_1\texttt{,}\bullet\texttt{)}$ such that $e = \mathcal{E}_\mathrm{e}[e_2]$ and the result follows by induction on e_2 and Lemma 42.

Case 16: Derivation ends in DTRelRem
Then $R = r\texttt{.rem(}e_1\texttt{,}e_2\texttt{)}$, $P, \Gamma, \sigma \vdash e_1 : n_1$ and $P, \Gamma, \sigma \vdash e_2 : n_2$.

If $e_1, e_2 \in \mathsf{DynValue}$ then $e_1, e_2 \in \mathsf{Address} \cup \{\texttt{null}\}$. If either $e_1 = \texttt{null}$ or $e_2 = \texttt{null}$ then progress is made to NullPtrError by OSRelRemN. Assume, then, that $e_1, e_2 \in \mathsf{Address}$. If $(r, e_1, e_2) \in \mathsf{dom}(\rho)$ then progress is made by OSRelRem1 to $\rho(r, e_1, e_2) \in \mathsf{Address} \subset \mathsf{DynValue} \subset \mathsf{DynExpression}$ as required. If $(r, e_1, e_2) \notin \mathsf{dom}(\rho)$ then progress is made by OSRelRem2 to $\texttt{null} \in \mathsf{DynValue} \subset \mathsf{DynExpression}$ as required.

If $e_1 \notin \mathsf{DynValue}$ then take $\mathcal{E}_\mathrm{e} = r\texttt{.rem(}\bullet\texttt{,}e_2\texttt{)}$ such that $R = \mathcal{E}_\mathrm{e}[e_1]$ and the result follows by induction on e_1 and Lemma 42.

If $e_1 \in \mathsf{DynValue}$ and $e_2 \notin \mathsf{DynValue}$ then take $\mathcal{E}_\mathrm{e} = r\texttt{.rem(}e_1\texttt{, }\bullet\texttt{)}$ such that $R = \mathcal{E}_\mathrm{e}[e_2]$ and the result follows by induction on e_2 and Lemma 42.

Case 17: Derivation ends in DTCall
Then $R = e_1\texttt{.}m\texttt{(}e_2\texttt{)}$, $P, \Gamma, \sigma \vdash e_1 : n_1$, $P, \Gamma, \sigma \vdash e_2 : t_2$ and $\mathcal{MD}_{P,n_1}(m) = (\_, \_, \_, \_, s_1\ \texttt{return}\ e_3\texttt{;})$.

If $e_1, e_2 \in \mathsf{DynValue}$ then $e_1 \in \mathsf{Address}$ or $e_1 = \texttt{null}$. If $e_1 \in \mathsf{Address}$ then progress is made by OSCall to $\{\ s_1\ \texttt{return}\ e_3\texttt{;}\ \} \in \mathsf{DynExpression}$ as required, under new Γ and λ. If $e_1 = \texttt{null}$ then progress is made by OSCallN to $\mathsf{NullPtrError}$.

If $e_1 \notin \mathsf{DynValue}$ then take $\mathcal{E}_\mathrm{e} = \bullet\texttt{.}m\texttt{(}e_2\texttt{)}$ such that $R = \mathcal{E}_\mathrm{e}[e_1]$ and the result follows by induction on e_1 and Lemma 42.

If $e_1 \in \mathsf{DynValue}$ and $e_2 \notin \mathsf{DynValue}$ then take $\mathcal{E}_\mathrm{e} = e_1\texttt{.}m\texttt{(}\bullet\texttt{)}$ such that $R = \mathcal{E}_\mathrm{e}[e_2]$ and the result follows by induction on e_2 and Lemma 42.

Case 18: Derivation ends in DTSkip
Then $R = \epsilon$.

Case 19: Derivation ends in DTExp
Then $R = e_1\texttt{;}\ s_1$. By DTExp, $P, \Gamma, \sigma \vdash e_1 : t$.

If $e_1 = u \in \mathsf{DynValue}$ then progress is made by OSStat to $s_1 \in \mathsf{DynStatement}$ as required.

If $e_1 \notin \mathsf{DynValue}$ then take $\mathcal{E}_\mathrm{s} = \bullet\ s_1$ such that $R = \mathcal{E}_\mathrm{s}[e_1]$. The result follows by induction on e_1 and Lemma 42.

Case 20: Derivation ends in DTCond
Then $R = \texttt{if (}e_1\texttt{) \{}s_1\texttt{\} else \{}s_2\texttt{\};}\ s_3$.

If $e_1 = u \in \mathsf{DynValue}$, then by DTCond, $P, \Gamma, \sigma \vdash u : \texttt{boolean}$ so $u = \texttt{true}$ or $u = \texttt{false}$. If $u = \texttt{true}$ then s_1 makes progress to $s_1\ s_3$. If $u = \texttt{false}$ then s_1 makes progress to $s_2\ s_3$.

If $e_1 \notin \mathsf{DynValue}$, then take $\mathcal{E}_\mathrm{s} = \texttt{if (}\bullet\texttt{) \{}s_1\texttt{\} else \{}s_2\texttt{\};}\ s_3$ such that $R = \mathcal{E}_\mathrm{s}[e_1]$. R then makes progress by application of the inductive hypothesis to e_1, and Lemma 42.

Case 21: Derivation ends in DTFor
Then $R = \texttt{for (}n\ x\ \texttt{:}\ e_1\texttt{) \{}s_2\texttt{\};}\ s_3$.

If $e_1 = u \in \mathsf{DynValue}$ and $u = \emptyset$ then R makes progress by OSFor1 to s_3.

If $e_1 = u \in \mathsf{DynValue}$ and $u \neq \emptyset$, then by DTFor, $P, \Gamma, \sigma \vdash e_1 : \texttt{set<}n_2\texttt{>}$, so $u \subseteq \mathsf{Address}$. We can therefore pick one $\iota \in u$ such that R makes progress to $s_4\ \texttt{for (}n\ x\ \texttt{:}\ u \backslash \iota\texttt{) \{}s_2\texttt{\};}\ s_3$ by OSFor2, where s_4 is s_2 with some consistent renaming of the iteration variable x (also applied to Γ, which is elided here).

If $e_1 \notin \mathsf{DynValue}$, then take $\mathcal{E}_\mathrm{s} = \texttt{for (}n\ x\ \texttt{: }\bullet\texttt{) \{}s_2\texttt{\};}\ s_3$ such that $R = \mathcal{E}_\mathrm{s}[e_1]$, which makes progress by the inductive hypothesis and Lemma 42.

□

Conclusion: All RelJ programs that are not in a terminal or exceptional state may execute to a new state by Theorem 43, which will be well-typed and well-formed by Theorem 41. Therefore, RelJ is type sound.