*Technical Report*

UCAM-CL-TR-702
ISSN 1476-2986

Number 702

**UNIVERSITY OF CAMBRIDGE**

**Computer Laboratory**

# Relationships for object-oriented programming languages

Alisdair Wren

November 2007

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

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

© 2007 Alisdair Wren

This technical report is based on a dissertation submitted March 2007 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Sidney Sussex College.

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

# Abstract

Object-oriented approaches to software design and implementation have gained enormous popularity over the past two decades. However, whilst models of software systems routinely allow software engineers to express relationships between objects, object-oriented programming languages lack this ability. Instead, relationships must be encoded using complex reference structures. When the model cannot be expressed directly in code, it becomes more difficult for programmers to see a correspondence between design and implementation — the model no longer faithfully documents the code. As a result, programmer intuition is lost, and error becomes more likely, particularly during maintenance of an unfamiliar software system.

This thesis explores extensions to object-oriented languages so that relationships may be expressed with the same ease as objects. Two languages with relationships are specified: RelJ, which offers relationships in a class-based language based on Java, and **Q**$\varsigma$, which is an object calculus with heap query.

In RelJ, relationship declarations exist at the same level as class declarations: relationships are named, they may have fields and methods, they may inherit from one another and their instances may be referenced just like objects. Moving into the object-based world, **Q**$\varsigma$ is based on the $\varsigma$-calculi of Abadi and Cardelli, extended with the ability to query the heap. Heap query allows objects to determine how they are referenced by other objects, such that single references are sufficient for establishing an inter-object relationship observable by all participants. Both RelJ and **Q**$\varsigma$ are equipped with a formal type system and semantics to ensure type safety in the presence of these extensions.

By giving formal models of relationships in both class- and object-based settings, we can obtain general principles for relationships in programming languages and, therefore, establish a correspondence between implementation and design.

# Acknowledgments

My thanks go first to my supervisors, Gavin Bierman and Andrew Pitts, both of whom have given generously of their time, knowledge and experience during the preparation of this work. This thesis could not have been completed without their diligent supervision and encouragement. I also owe special thanks to Matthew Parkinson for being a guide, sounding board, indefatigable proof-reader and a source of excruciating puns, none of which have been included in this thesis deliberately. I am also very grateful to my examiners, Giorgio Ghelli and Andrew Kennedy, for all their efforts to make this a better thesis.

For their comments and suggestions on this work, I would like to thank Dave Clarke, Sophia Drossopoulou and Imperial College's SLURP group, Alan Mycroft, James Noble, David Pearce, Peter Sewell, Gareth Stoyle, Darren Willis, the Programming Principles and Tools group at Microsoft Research Cambridge, the attendees of *Semantics Lunch* and *Logic and Semantics for Dummies* at the Computer Laboratory, and the anonymous referees of FOOL 2005 and ECOOP 2006. Further thanks go to Sophia Drossopoulou and to Paul Kelly, both of Imperial College, without whose tutelage as an undergraduate I would certainly not have embarked on a PhD at all.

Away from work, I have greatly enjoyed my time at Sidney Sussex College, and I am grateful to all of Sidney's students, staff and fellows, and in particular to Sidney's graduate society, the MCR. Finally, thank you to my family for their constant support and to my friends for their tolerance and their wit.

# Contents

**1 Introduction** **9**
1.1 Object-orientation . . . 9
1.2 Encoding relationships . . . 19
1.3 Thesis . . . 22
1.4 Contribution . . . 23
1.5 Related work . . . 24

**2 RelJ: Relationships for Java** **29**
2.1 Core language . . . 29
2.2 Relationships . . . 31
2.3 Language definition . . . 40
2.4 Type system . . . 44
2.5 Semantics . . . 52
2.6 Soundness . . . 61
2.7 Correctness . . . 66
2.8 Conclusion . . . 68

**3 Extending RelJ** **69**
3.1 An alternative model of relationship inheritance . . . 69
3.2 Relationships with multiplicities . . . 76
3.3 Other extensions . . . 81
3.4 Conclusion . . . 83

**4 Qς: An object calculus with heap query** **85**
4.1 Core language . . . 85
4.2 Heap query . . . 91
4.3 Type system . . . 96
4.4 Semantics . . . 101
4.5 Soundness . . . 105
4.6 Extensions . . . 110
4.7 Conclusion . . . 114

**5 Bridging the gap** **115**

5.1 Translation overview . . . . . . . . . . 115
5.2 **Q**ς booleans and conditionals . . . . . . . . . . 120
5.3 Formalization of the translation . . . . . . . . . . 122
5.4 Conclusion . . . . . . . . . . 136

**6 Conclusion 137**
6.1 Future work . . . . . . . . . . 137

**Guide to notation 141**

**Bibliography 147**

Chapter 1

# Introduction

Relationships between objects are as important as the objects themselves [62]. However, whilst objects or, more generally, entities are well-represented throughout the software development process, relationships are often lost after the design phase and must be implemented indirectly [56, 57].

Certainly, developers are aware of objects — cars and roads, students and universities — and the relationships that exist between them — cars drive on roads, students are educated at universities. This intuition is largely preserved in the design phase: Unified Modelling Language (UML) [47, 68], the *de facto* standard, has associations between classes of objects, whilst Entity-Relationship (ER) diagrams [21] may be used to design an application's data store. At implementation, however, object-oriented programming languages are unable to faithfully represent relationships. Instead, we are left with references, which provide only the most impoverished abstraction of inter-object relationships: unlike objects and relationships from the models, they enjoy neither state nor behaviour and, in the presence of unconstrained mutability and aliasing, can lead to confusion and programmer error [4, 6, 23, 44].

In this thesis, we explore how relationships can be given the same status as objects. Both species of object-oriented language — class-based and object-based — are examined and extended to include relationships, and in both cases formal methods are used to ensure that these new features may be implemented safely. Despite the apparent differences between the class- and object-based worlds, common principles emerge which suggest a consistent, general approach for the correct and safe implementation of relationships in object-oriented languages.

## 1.1 Object-orientation

### 1.1.1 The briefest of histories

Object-oriented programming languages have enjoyed enormous popularity since the 1990s, such that the object has largely overtaken the procedure and the module as the primitive means of breaking up large software systems.

By the 1970s, the procedure was well-established as a means by which large lists of commands could be broken into smaller, reusable units [19, 75]. In combination with finer-grained structured programming disciplines, these procedures could be considered in some degree of isolation such that the programmer could more easily satisfy himself that the program was correct, especially in the presence of some form of specification [8, 19, 28, 43].

In 1972, Parnas' ideas on information hiding further contributed to looser coupling between program components, so that modules could also be considered in increasing isolation, with only a shared, abstract interface [58]. Thus, the decomposition of behaviour into procedures was joined by the decomposition of state and of a program's procedure set into modules.

It seems a natural step from modules to *abstract data types*, a means by which to take multiple copies of the module and to extend a language's built-in type system [40, 51], and from there to objects. However, Simula-67 by Dahl and Nygaard in 1967 [26] already had the necessary machinery to support information hiding and possessed many of the features familiar to modern object-oriented programmers: classes, objects, inheritance and subtyping. Thus, it became the first exemplar of what has become one of the pre-eminent software engineering disciplines.

These remarks can only function as the most potted history of an evolution of object-oriented programming that is well-explored in the literature [19, 73], and which does not end with Simula: Smalltalk [48] and Self [71] exemplify the continuing development of object-oriented programming languages before C++ [18], Java [38] and, later, C# [55] came to prominence in industry. Nevertheless, this evolution demonstrates the rôle objects have in structuring software.

A new rôle for objects in software engineering was promoted in 1976 when Chen proposed Entity-Relationship models as a means for modelling data [21].[1] Entities in these models lack ideas of encapsulation and behaviour, and hence program structure, but they demonstrate how a programmer may use objects to structure a program according to some intuition of the world that is being modelled. If a system has been developed according to an intuitive model, then a programmer equipped with this intuition will be able to maintain the software more effectively and with a decreased chance of error, especially when he was not responsible for its original implementation. Thus, the Entity-Relationship model was a fore-runner of other object-oriented modelling languages such as the Booch Method [15], the Object Modelling Technique (OMT) [63] and, most recently, the Unified Modelling Language [47], commonly used today to both design and document object-oriented systems.

[1] We use 'object' and 'entity' interchangably when discussing the development of software models, as both represent an abstraction of some real-world concept represented directly in a software design.

### 1.1.2 Object-oriented programming languages

Throughout this evolution, what it means for a programming language to be object-oriented has been the subject of debate: it is not unusual for an object-oriented language to lack a feature declared elsewhere to be indispensable. Simula, for example, lacks dynamic dispatch (see below), but the designer of C++, Bjarne Stroustrup, believes that a language does not support object-oriented programming without — in C++ parlance — virtual functions [69].

At the very least, however, an object is a package with a unique identity, some state and some behaviour. For the purposes of this work, an object's identity will be its address in memory. An object's state will be formed from a collection of named *fields*, which take values including object identities — thus, an object may hold a *reference* to another object, or even to itself. Where a field does not hold such a reference, its value is said to be 'null'. An object's behaviour will be formed from a collection of named *methods*, which contain commands that, amongst other actions, operate on the object's fields. An object's fields and methods together form its set of *attributes*. Through references, an object method may access the attributes of other objects as well as the attributes of its own object: a method always knows the identity of the object to which it belongs, known as a reference to *self*. In general, the target of a message invocation is known as the *receiver* of the method call.

We now review other features common to object-oriented languages beyond the fundamental definition given above. Some of these ideas are intertwined, and are often named inconsistently in the literature and among practitioners of different software engineering disciplines; therefore, this list also serves to lay out definitions for use in our discussion.

**Encapsulation** We have already discussed the history of object-oriented programming languages with respect to their ability to modularize a software system by encapsulating some state and behaviour. Depending on the available language features, an object's state can be hidden from the outside world so that the object forms a boundary around some of its fields. In C++, for example, a field may be declared `private`, so that it can only be accessed by a method invoked on the same object. This protection may be short-lived, however, as such a method may legitimately read a private field's value and transmit it out of the object as its return value. Where languages lack annotations for protecting individual fields, the language designer is left with a choice between exposing all fields to the outside world, or none at all. The choice makes little difference to the capabilities of the language: where fields are shielded entirely from external access, methods may be used to indirectly return or update field values. Such methods are often known as *getter* and *setter methods* respectively.

**Abstraction** By encapsulating state, an object can ensure that the environment does not manipulate its state in an unexpected way. Where a language supports

the specification of hidden attributes, those that remain public form an *interface* for the object. An object representing a car may, for example, expose methods that allow the driver to switch the car on, turn left and right, change speed and to switch it off. It would not, however, expose methods that allow individual spark plugs to be fired — such a method might form part of the car's implementation, but the driver has no need to view the implementation of the car in such detail. As well as by encapsulation, interfaces may be specified explicitly for objects: objects with very different definitions may still conform to the same interface.

**Generalization** It is expected that some objects will share common properties: for example, vehicles usually have an engine and can carry passengers, regardless of whether they are cars or aeroplanes. Rather than specifying such properties for every vehicle, we can regard 'being a vehicle' as a property that all vehicle objects share. By attaching attributes to vehicles in general rather than to individual objects, we ensure that all vehicles are specified consistently and make it easier to maintain properties associated with being a vehicle. Where a type system is used, generalizations may form types: here, a type whose values are vehicles. Generalizations also form abstractions, or interfaces — an abstract view of a car is that it is a vehicle.

**Reuse of behaviour** A special case of generalization involves the reuse of behaviour or, more specifically, the code that implements that behaviour. Not only does this help enforce the idea that vehicles behave similarly, but the ability to reuse code to implement the behaviour of several objects improves the maintainability of the code: a bug fixed in one object's behaviour is fixed for all objects using that code. Re-use of behaviour usually involves some mechanism for obtaining methods from other objects, or from a generalization.

**Specialization** Whilst groups of objects may be ostensibly the same, slight variations may be accommodated: like other vehicles, a rocket may carry passengers and has an engine, but unlike other vehicles it also has a heat shield, for example. To start from scratch with a new concept of 'being a heat-shielded vehicle' would involve the reimplementation of engines and the advantages of generalization would be lost. Instead, we specialize vehicles to accomodate the presence of a heat shield and, in doing so, create a new generalization of space vehicles subordinate to the original generalization of vehicles. Of course, one can imagine that the relationship between the generalizations can be inverted: vehicles could be space vehicles specialized to omit the heat shield. Under the influence of a type system, there is a general preference for adding attributes as objects are specialized rather than removing them. In this case, the type of space vehicles is considered a *subtype* of vehicles.

**Overriding of behaviour** A method is overridden where its implementation, derived from some generalization, is replaced. The attributes possessed by the resulting object will match those of the original object, but the new object's method will behave differently. As such, overriding does not always imply the creation of a new type.

**Subtype polymorphism** Subtype polymorphism permits an object of a subtype to be used in place of an object of a supertype. For example, a space vehicle may be used in any position where a vehicle is expected. In this case, the object's *dynamic type* would be that of space vehicles, whilst its *static type* would be that of vehicles.

**Dynamic dispatch** Suppose that vehicles have a `startEngine` method, which turns on the vehicle's engine. Buses, cars and lorries all run on internal combustion engines, so they may all use the same implementation of `startEngine`. However, a rocket — derived from the generalization of space vehicles — may have a much more complicated `startEngine` method, which overrides that defined for vehicles. Code that expects a vehicle and invokes its `startEngine` method may use a space vehicle according to subtype polymorphism. However, the behaviour of that method depends on how the invocation is *dispatched*. Static dispatch selects the behaviour according to the receiver's static type: it would invoke the vehicle's definition of `startEngine`, as the code was designed to operate on vehicles and not space vehicles. Dynamic dispatch, however, uses the dynamic type to select behaviour, so the space vehicle's version of `startEngine` is invoked: it determines, at run-time, which is the most appropriate method to invoke.

### Object-based versus class-based

Object-oriented languages may be divided into two broad categories: object-based and class-based. Whilst some languages straddle the boundary, this division largely determines how the properties specified above manifest themselves in the language.

**Object-based languages** In object-based languages, the generalization mechanism is based on *prototypes*: new objects may be created by *cloning* an existing object, after which the object may be specialized by the addition of new fields and methods, by the update of existing methods or, in some cases, by the removal of attributes.

Behaviour reuse is implemented by *delegation*: an object based on a template object may use methods defined by the template object rather than unnecessarily re-implementing those methods — the object delegates method calls to the template. If the template's behaviour is updated, all the objects that delegate method calls to that template will receive the benefits of that update. For example, suppose objects `bus` and `car` are constructed from a template object, `vehicle`,

to which they both delegate behaviour for their startEngine methods. After some time, vehicle's startEngine method is updated because all vehicles must now use unleaded fuel: implicitly, buses and cars can be started with unleaded fuel too. Conversely, a flyingSaucer may be constructed from vehicle but is sufficiently different that its startEngine method is re-implemented: it will not be affected by the conversion of Earth-bound vehicles to unleaded fuel.

Whilst these dynamic features allow rapid software development, they leave programs vulnerable to 'message not understood' errors, where an object receives a request for a field or method that does not exist. Such errors can be caught before run-time using a static analysis or, more usually, a structural type system, which specifies the attributes available for a given object [1, 7]. The drawback of object-based type systems is that they tend to be either too restrictive or too complicated for use by inexpert programmers. Nevertheless, dynamically-typed languages such as Self [71] and Javascript [33] remain suitable for prototyping and for situations where safety is not critical, such as in the implementation of web applications. Object-based calculi have also been studied in order to determine the theoretical 'essence' of object-oriented programming, perhaps most famously by Abadi and Cardelli [1], whose calculi shall feature prominently in this work.

**Class-based languages** Generalization in class-based languages stems from the declaration of classes, which specify the fields and methods present in objects belonging to that class, otherwise known as *instances* of that class. An object's class is therefore similar to a prototype object in an object-based language, although it is not usually an object itself. As a result, classes are often regarded as object factories.

Subtyping and reuse are often conflated and implemented using *inheritance*: a new class (the *subclass*) is created by inheriting the fields and methods of an existing class (the *superclass*). At the same time, classes may be specialized: new fields and methods may be added to those that have been inherited. It is also possible to *override* inherited methods, so that instances of the subclass obtain new behaviour. Other re-use mechanisms have been discussed, for example Mixins [35] and Traits [64], but these are largely outside the scope of our discussion.

Returning to our previous example, vehicle would be represented as a class in a class-based language. As flyingSaucer is a specialization of vehicle, it must also be represented as a class: a subclass of vehicle. If car and bus require specialization, or if more than one car or bus is needed, then they too will be represented as subclasses of vehicle; if not, they may be instances of vehicle.

Type systems for class-based languages are generally *nominal*: an object's class determines its available attributes so, in the simple case, the types are class names, and subclasses are subtypes.[2] Such a type system is simpler — if a vari-

[2] The adoption of generics has started to introduce structural considerations to class-based type systems, particularly in the presence of variance [34].

able is declared to have type flyingSaucer then it can hold references to flying saucers without listing all of their attributes — so it is unsurprising that popular languages such as Java and C++ are class-based.

### Aggregation

Regardless of whether a language is class- or object-based, *aggregation* is an important part of object-oriented design. Whilst the car object above could be constructed monolithically, such that all the attributes representing the car's components were contained by a single object, this strategy neither supports the re-use of behaviour when specializing cars nor encapsulation of the components' properties. Instead, car can hold references to an engine object, to four wheel objects, and so on. When car receives a startEngine invocation, it may do some of its own work such as turning on the car's stereo, but most of the work will be performed by invoking engine's start method. In this arrangement, engine is a *component* of the car, and car is an *aggregate object* built from a number of objects that help to provide the object's state and behaviour.

By breaking the object up in this way, the properties of the engine are protected from interference by car's code, depending on the language's ability to restrict access to engine's attributes. For example, one would expect the car to be able to turn the engine on and off, increase and decrease its power, and inspect certain operational properties but one would not expect the car to be able to arbitrarily alter the engine's preferred air-to-fuel mixture. Thus, putting the engine in its own object has created an abstraction of engines. This abstraction promotes re-use when specializing a car to an environmentally friendly car: an electric car engine has the same interface, but a different implementation. As the car only knows about the interface, the original engine object can be swapped for an electric engine without adjusting the car's code: we obtain an electric car without needlessly specializing the entire car object.

Earlier, methods for protecting the state of individual objects were discussed. There has also been considerable work involved in protecting the internal objects of an aggregate — like engine in the example above — from outside interference: Ownership Domains [4] and Ownership Types [17, 23], Balloons [6] and Islands [44] all attempt to express boundaries around groups of co-operating objects.

### Design patterns

Design patterns are programming idioms, often used to solve particular problems, originally due to Alexander [5] and applied to object-oriented programs by 'the gang of four': Gamma, Helm, Johnson and Vlissides [36]. A pattern usually involves a number of objects that collaborate in some way, but does not give any concrete implementation details: instead, it is a document that gives the problem addressed by the pattern, gives an abstract, graphical, object-oriented

specification for how that problem may be solved, and lists the consequences of using that pattern. The specification is normally given in UML, which is described below.

Patterns help to ensure that programmers do not implement a sub-optimal or unclear solution to a problem that others have already solved, but they also underline a deficiency in the implementation language. Such a deficiency may be desirable: adding each design pattern as a language primitive would certainly increase language complexity [16]. Additionally, design patterns can tempt programmers to force an elegant design into a less elegant, pattern-based design — exactly what software engineers hope to avoid.

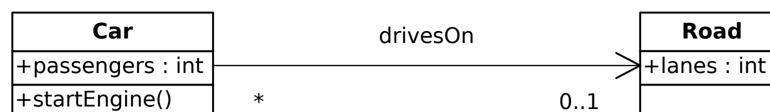
### 1.1.3 Modelling languages

In the process of designing a software system, a software engineer may use a variety of techniques to express and improve the design. In an effort to find some common ground between how a programmer thinks and how a programmer writes code, we focus here only on models used to express the design of a system, either before development or afterwards, as documentation. We concentrate on two models in particular, both of which are regarded as *de facto* standards: the class diagrams of Unified Modelling Language (UML) [47], and Entity-Relationship diagrams [21].

#### Unified Modelling Language

Unified Modelling Language (UML) [47] arose from earlier object-oriented software modelling techniques including Object Modelling Technique (OMT) [63], the Booch method [15] and Object-Oriented Software Engineering (OOSE) [46]. Whilst we focus here on UML's class diagrams, UML also contains a variety of disciplines for expressing higher level structures such as packages, for modelling state and interactions between objects and with users.

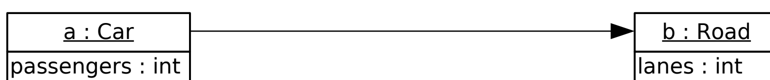
At their simplest, UML class diagrams are composed of classes and associations between them:




The diagram above contains two classes, Car and Road, and an association between them, drivesOn. Cars have a field, passengers, which holds an integer, and a method startEngine as discussed earlier.

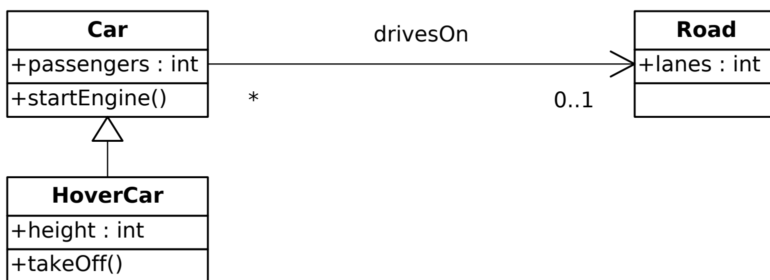
The association, drivesOn, indicates that instances of Car may be associated with instances of Road. The arrowhead indicates the association's *navigability*, in this case that the association may only be traversed in one direction: from an instance of Car we may reach the Road it is driving on, but not the other way

around. Each end of the association is annotated with a *multiplicity*: the '*' indicates that any number of cars may drive on each road, but the '0..1' annotation indicates that each car may drive on at most one road. These annotations — navigability and multiplicity — may be omitted in order to relax the restrictions placed upon the association. As we shall see later, a many-to-one, one-way association is precisely expressible by a reference from instances of Car to instances of Road: many Cars may have a drivesOn field, each of which may hold a reference to a Road instance. This may be demonstrated with a UML object digram, which shows a possible instantiation of a class diagram into objects and references:



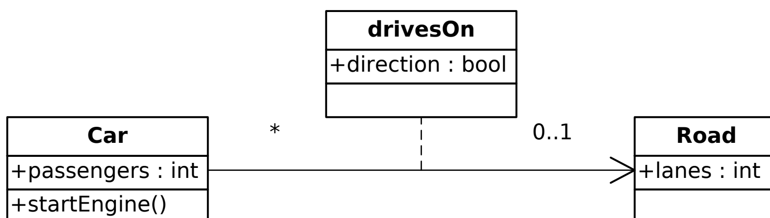

Each object has a name (sometimes omitted) and the name of its class, separated by a colon and underlined. Object a is an instance of class Car, and object b is an instance of class Road. The arrow represents a reference from a to b. Here, it indicates that a drives on b, but references may be named where the context leaves this unclear.

Class diagrams also contain information about inheritance. In the following, Car generalizes HoverCar, which can takeOff to some height:




The inheritance is indicated with a hollow-headed arrow. As HoverCar is a specialization of Car, it may also drive on a Road.

Finally, we may add some attributes to the drivesOn association with an *association class*, connected to an association with a dotted line:




Thus, when a Car instance drivesOn a Road instance, it does so with a certain direction. Here, drivesOn has no methods.

UML class diagrams can be much richer than demonstrated here, yet already we see that there is a rich model of assocations between instances of classes.

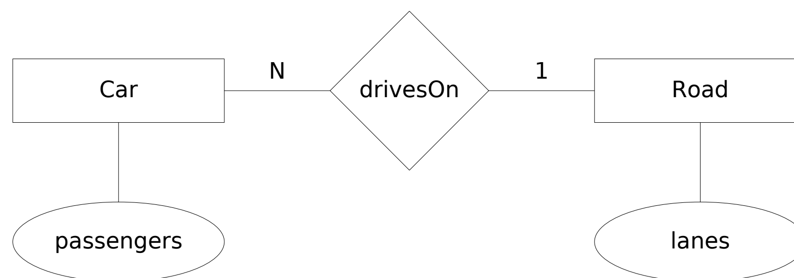
As these modelling languages have evolved according to the needs of software engineers, we argue that these diagrams are at least as expressive as the models that engineers have in mind during development.

### Entity-Relationship diagrams

Entity-Relationship diagrams (ER diagrams) [21] have a less prominent rôle in modern software engineering, as they tend to be used to model databases rather than software. Nevertheless, they pre-date other object-oriented models by nearly twenty years and have had a tremendous influence on data modelling. As data has acquired behaviour with the advent of object-oriented programming, their influence is keenly felt in languages such as UML.

An ER diagram consists of entities, relations and attributes, which correspond to classes, associations and fields/methods in UML diagrams. As a model of data, based on tables of tuples, they lack inheritance and direction of relations.

The Car/Road example above may be modelled like so:




Entities are enclosed in square boxes, relations in diamonds, and attributes by ellipses. These are joined by lines to indicate the participation of attributes in entities and of entities in relations. Relation participations are annotated with cardinalities — just as before, drivesOn is a many-to-one relation. Attributes may be added to relations as well as entities, just as for associations in UML. Furthermore, relations may participate in other relations alongside entities: this is known as *aggregation* [66].

Again, this description of ER diagrams does not aim to be exhaustive, but serves to demonstrate that a rich model of relationships between entities existed in the earliest models of data. In some regards, this structure is erased in the implementation of the model in a relational database. For example, an implementation in tables may have the form:

**Cars**

| carId | passengers | drivesOn |
|---|---|---|
| A | 4 | (null) |
| B | 2 | M25 |

**Roads**

| roadId | lanes |
|---|---|
| A101 | 2 |
| M25 | 4 |

Two cars, A and B, carry four and two passengers respectively. The first is not being driven on a road. The second is being driven on the M25, which has four

lanes. Notice that the identity of entities, and relations between them have all been reduced to table attributes. A separate table could be used for the drivesOn relation: this would imply a many-to-many relation that is not in the model, and would still conflate relations and entities.

## 1.2 Encoding relationships

As we have seen, software models rely on objects/entities and relationships between them. At the same time, object-oriented programming languages possess objects with the same power as objects in the models, but only have inter-object references rather than relationships.

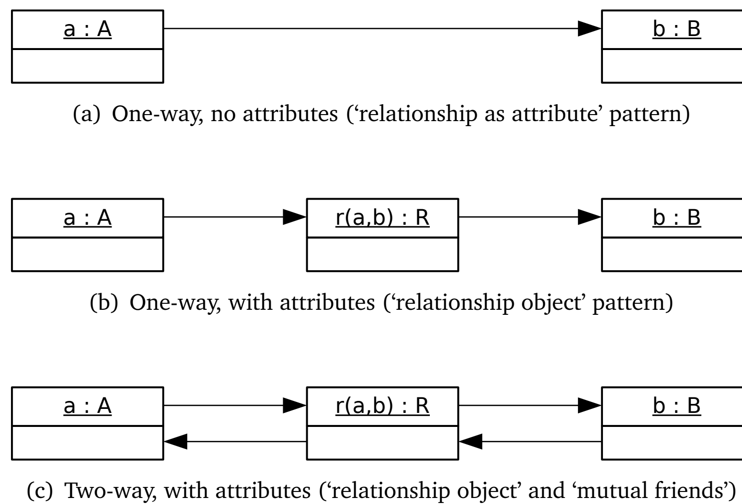
Alone, an inter-object reference from one instance to another can be followed only in one direction, can only reference a single object, and cannot have attributes. As several objects may have a reference to any given object, references can only directly express a one-way, many-to-one relationship without attributes. Beyond this, relationships must be encoded.

Noble has proposed design patterns for expressing relationships in object-oriented languages [56], which lay out strategies for breaking a single UML association into an aggregate of objects and references — indeed, the patterns can be seen as strategies for transforming associations of, say, a UML diagram so that the diagram is composed only of associations that can be represented by references. Similar strategies have been employed in a variety of mappings from UML models to program code, such as that by Harrison et al. [41]. We describe, using Noble's pattern names, four simplified versions of these patterns:

**Relationship as attribute** As discussed, a many-to-one, one-way, association without attributes between class A and class B requires no transformation. It can be directly expressed at run-time by a reference to a B-instance from an attribute of an A-instance. Figure 1.1(a) shows objects a and b, which are instances of classes A and B respectively, in such a relationship.

**Relationship object** Where the association has an association class — put differently, where the relationship has attributes — we are left with the problem of where to store the relationship's data. The data can be distributed amongst the relationship's participants, but this does not represent the data in a way that respects encapsulation. Instead, then, the data is stored in its own object. For a many-to-one relationship, R, between classes A and B, we create a class for R and have an A-instance reference an R-instance, in which R's attributes are stored. The R-instance then references the related B-instance. In Figure 1.1(b), a and b are related by an instance of relationship R, r(a,b).

**Mutual friends** A two-way relationship between classes A and B can be modelled with two references — one from an A-instance to a B-instance, and one in




(a) One-way, no attributes (‘relationship as attribute’ pattern)

(b) One-way, with attributes (‘relationship object’ pattern)

(c) Two-way, with attributes (‘relationship object’ and ‘mutual friends’)

Figure 1.1: Objects representing many-to-one relationship, R, between A and B

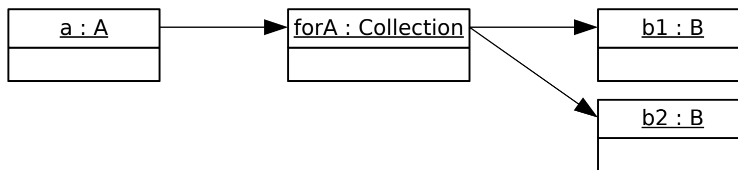
reverse. This can be combined with the relationship object pattern where the association has attributes, as shown in Figure 1.1(c). Owing to the structure's symmetry, such a relationship is one-to-one rather than many-to-one.

**Collection object** Many-to-many associations can be implemented with collection objects, such as linked lists: rather than an object directly referencing its relative, it references a collection object that holds references to several relatives. This situation is shown in Figure 1.2(a), where a is related to two B-instances, b1 and b2, through the collection object forA.

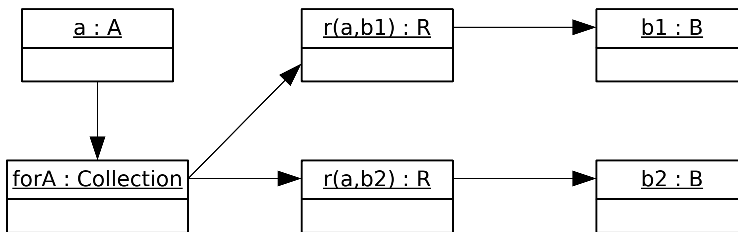
Collection objects can be combined with the relationship object pattern, as shown in Figure 1.2(b), to represent many-to-many relationships with attributes. This can be further combined with the mutual friends pattern to obtain a two-way, many-to-many relationship with attributes, an example of which is shown in Figure 1.2(c).

### Criticism

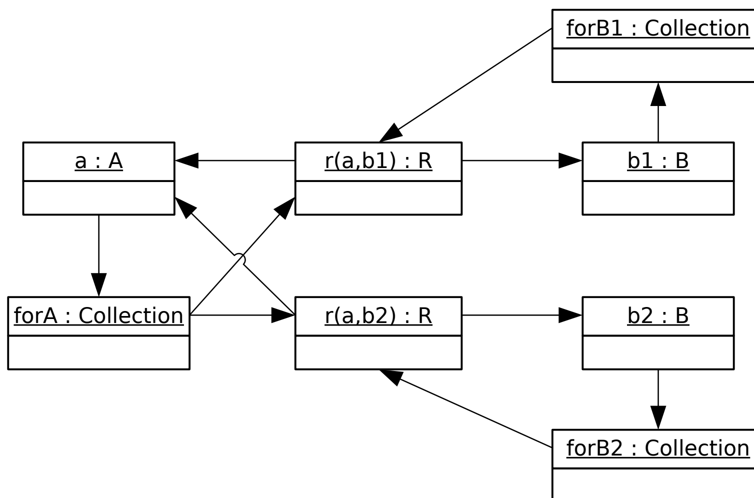
Whilst these patterns provide a systematic way to translate UML associations to designs expressible in object-oriented languages, their use does not permit the relationship to be faithfully represented in the program. As a result, the original UML association cannot easily be recovered from its implementation, although Guéhéneuc and Albin-Amiot have proposed an approximate technique using static and dynamic analyses [39]. An inability to equate a software model with its implementation complicates maintenance and encourages programmer error, sometimes referred to as the *traceability problem* [16, 67].




(a) One-way, without attributes ('collection object' pattern)



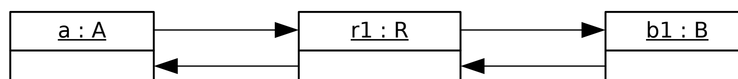

(b) One-way, with attributes ('collection object' and 'relationship object')



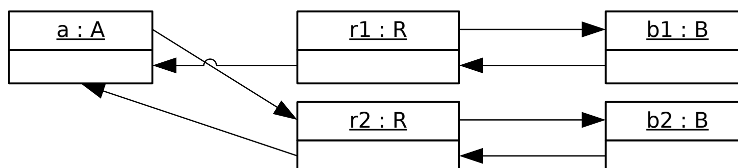

(c) Two-way, with attributes (as above, with 'mutual friends')

Figure 1.2: Objects representing many-to-many relationship, R, between A and B

The patterns also yield complicated structures that can easily become inconsistent, especially in the presence of association classes and two-way associations. Taking a simple example, the objects in Figure 1.1(c), which represent the implementation of a two-way, one-to-one relationship with attributes, may have the following form:




A programmer may then update this structure with the intention of relating a to a different B-instance. However, as information about the relationship is distributed across three classes, the programmer cannot reason about the relationship in isolation and may make a mistake — here, he has set up a new instance of R correctly, but has not updated the participants of the old relationship. Thus, following references from b1 it appears that a and b1 are still related, whereas references from a indicate that it is related to b2:




There is a variety of work on maintaining invariants on aggregate structures such as these. For example, the Contracts of Helm et al. permit the specification of dependencies between objects and their obligations to one another [42]. Ducasse et al. similarly propose a means to automatically manage dependencies [32]. Nevertheless, we are still left without a direct abstraction of relationships in program code.

Finally, one-to-one and one-to-many relationships between A and B cannot be reliably represented without creating two-way structures as found in Figure 1.1(c): in general, more than one A-instance is free to reference any given B-instance. Known as *aliasing*, this gives object-oriented programming much of its power, but also poses a significant challenge for producing reliable, secure software. There are a variety of techniques for detecting, declaring and restricting aliasing [4, 6, 17, 22, 23, 44, 74], but in many cases these come with a significant overhead, and are not sufficiently expressive to impose all of the constraints we require.

## 1.3 Thesis

Software engineers and software design tools are well equipped to consider relationships between objects and entities, whilst programming languages are not. As has been argued elsewhere, relationships should be available all the way through software development, from design to implementation [57, 62]. Only a language

extension can give relationships the same status as objects: we have already seen that encoding relationships is cumbersome and potentially error prone. Libraries can package up new abstractions, but they cannot provide the typing guarantees one expects from such a fundamental feature of object-orientation, and relationships remain encoded backstage. In an age when object-oriented programs are routinely compiled to code for abstract machines, relationship information can be maintained so that expected invariants and hence security are maintained at run-time — this is impossible when relationships are erased from the program text.

We therefore conclude that object-oriented languages should be extended to accommodate relationships with the same standing as objects. We further conclude that such language extensions be subject to the rigours of formal reasoning to ensure the programming language's safety: specifically, that no well-typed program may reach an unanticipated state. As we shall see in the following chapters, typing relationships is not necessarily straightforward, just as typing has caused difficulties for object-oriented programming languages in the past [25, 27, 30].

## 1.4 Contribution

This thesis conducts a formal exploration of relationships in object-oriented languages, covering both the class-based and object-based worlds. In doing so, we consider relationships with all of the features offered to classes and objects including subtyping and generalization/specialization.

We first demonstrate the practicality of including relationships as class-like constructs in a class-based object-oriented programming language called RelJ. RelJ is based on Java, but the ideas demonstrated are broadly applicable to other strongly and statically typed class-based languages, such as C#. Relationships in RelJ share many the characteristics of association classes from UML: they relate two classes (or, indeed, other relationships), they may have fields and methods, and they may inherit from other relationships. At run-time, relationships are instantiated when a pair of objects is placed in a relationship, so that the resulting relationship instance holds the relationship state for that pair. That relationship instance may be referenced just like a class instance. RelJ provides operators for accessing relationships: for example, an operator to determine for a given object which objects are related to it. A formal type system and semantics is given, and we give a soundness result which specifies that a well-typed program cannot get into an unsafe state.

The semantics of RelJ is based on an abstraction of a program heap, which is queried upon relationship access: the complex implementation of relationships using references and objects is, to some extent, maintained but responsibility is wrested from the programmer. As a result, we can ensure formally that the semantics maintains consistency of these structures, and that a safe abstraction is presented to the programmer.

Moving into the object-based world, we go further by eliminating an explicit relationship abstraction, and introducing more flexible heap queries into an object-based language, **Q**$\varsigma$, based on Abadi-Cardelli's $\varsigma$-calculus. In **Q**$\varsigma$, when an object has a reference to two other objects, a query can be written to discover that fact and those two objects are seen to be related. The referencing object's state becomes the relationship's state. No back-pointers are required to maintain bi-directionality; no collection objects must be maintained: objects and relationships are reunited as a single abstraction with heap query. Again, we give a formal type system and semantics and show that this language extension is safe. Finally, we demonstrate that **Q**$\varsigma$'s heap query can represent RelJ's relationships by giving a translation from RelJ to **Q**$\varsigma$.

Related literature is summarized in the remainder of this chapter, after which this thesis is organized as follows:

**Chapter 2** We introduce RelJ, a class-based language similar to Java or C# with relationships. Although the model of relationships is kept simple, relationships may have fields and methods, their instances may be referenced like objects, and they may inherit from one another. We give a formal type system and semantics, and show that the new language is type-safe.

**Chapter 3** A new form of inheritance for RelJ relationships is introduced, alongside other variants of RelJ to show how more advanced aspects of, for example, UML relationship models may be accommodated.

**Chapter 4** Relationships are investigated in an object-based setting. We extend the Abadi-Cardelli $\varsigma$-calculus to create **Q**$\varsigma$, which adds the ability to query the heap. We find that heap query generalizes the idea of relationships from RelJ, and that query removes the need for the object model and relationship models to differ.

**Chapter 5** We consolidate the correspondence between relationships and heap query by giving a translation from RelJ programs to **Q**$\varsigma$.

**Chapter 6** This chapter discusses future directions for research and concludes the thesis.

## 1.5 Related work

Rumbaugh suggested the addition of relationships to object-oriented programming languages in 1987 [62] and, with others, made the first steps towards a relationship-capable language with DSM [65]. Whilst DSM addresses the constraint aspects of relationship modelling, relationships are not represented quite at the same level as classes — inheritance and attributes are not available. As an

aside, Rumbaugh also proposes that operations such as object deletion should be propagated through relations, suggesting an interesting rôle for relationships in defining the boundaries of an aggregate object [61].

Kristensen describes Complex Associations [49], a richer model of relationships than is available in UML, where classes may be nested inside classes, and where associations may link those classes. Further, associations may be nested inside other associations, linking classes nested inside the participants of the larger association. Complex associations can therefore function as conduits for relationships between components of aggregate structures. Kristensen also informally describes a language of binary associations and nesting that supports the model. Whilst the model is quite expressive and embodies many of the principles found in the literature on encapsulating aggregate objects, we believe a model of simple associations should be established before a more complicated model is introduced.

Some language extensions have focused on higher-level abstractions, but a means to express relationships has emerged incidentally: for example, Bosch's layered object model [16] aims to improve the representation of design patterns in object-oriented programming languages by adding a new layer of abstraction. At the top level, design patterns may be described abstractly, and subsequently instantiated for specific situations, promoting re-use. The resulting language is powerful, but is significantly more complicated than Java and C++. Furthermore, relationships are not directly represented at the same level as classes, so there will remain a mismatch between the design and the language. We argue that relationships are an important part of expressing many design patterns, and that their addition would make both a simpler and more fundamental step towards the faithful representation of patterns in programming languages.

Classages [52] by Liu and Smith are designed to improve the specification of inter-object collaborations — where objects work together as an aggregate object — and are therefore capable of representing inter-object relationships. Each classage declares fields and methods as usual, but also declares various types of connector, which serve as interfaces for a specific collaboration between classage instances, known as objectages. Connectors may import and export methods, and may have their own fields. By plugging the connector of one objectage to the corresponding connector of another objectage, a relationship between those objectages is formed. Relationships have state by virtue of the state contained in the participating connectors. The connection has a handle which may be passed around as a value, just like a reference to a RelJ relationship instance, but its passage across relationships is restricted in order to support an improved model of encapsulation provided by the declaration of minimal interfaces for each of an objectage's collaborations. Typing is static, and structural, but there is not yet a soundness result. Relationships expressed by classages differ from RelJ's model and from relationships as represented in **Q**ς, most notably because they are declared in two places, as part of each participating classage. As a result, subtyping and reuse for relationships declared in this way are bound up in the subtyping

and reuse between their participating classes. Despite this, classages represent an encouraging move towards more powerful abstractions for dealing with collaborative relationships between objects, and for improving encapsulation of those relationships.

Balzer, Gross and Eugster [10] have also focused on relationships as a vital aspect of object collaborations. Whilst the features available for their relationships are partly based on RelJ, they focus on exposing a richer abstraction for relationships — that of discrete mathematical relations — which offers more power when, for example, specifying invariants; more power does, of course, reduce opportunities for statically ensuring invariants are satisfied. Similarly to the use of triggers by Albano et al. [2], they propose a number of run-time strategies to maintain declared invariants. Further, the authors introduce *member interposition*, which imbues objects with extra fields and methods when they participate in certain relationships. This extends the ability for relationships to have their own state.

Pearce and Noble [59] have recently described some aspect-oriented design patterns for relationships, and provided them in a library. The ability for aspects to cross-cut standard class boundaries allows classes participating in a relationship to be equipped with the required fields and methods without their declarations being distributed across classes in code. As a result, once relationships are declared using the Relationship Aspect Library, their system provides a convincing implementation of a RelJ-like language with only a small performance penalty over hand-coded implementations

Along with Willis, Pearce and Noble have gone on to explore query as a means to express (or discover) inter-object relationships [76]. They observe that a running program has implicit relationships (e.g. students whose names are identical) that are not explicit in the program text. Collection objects are often used to express such a relationship, but it must be updated at every update of a student's name. They propose that the set of objects with such a property could instead be obtained by querying the heap, and offer a library to efficiently evaluate such queries. Whilst typing issues are not discussed, their work certainly reflects how **Q**ς's object query, mapped into a class-based language, might be implemented: the application of a formal model such as **Q**ς remains important so that the programmer may expect complete and type-correct results.

A number of query systems exist outside programming languages for object-oriented heaps: Lencevicius et al.'s query based debugging [50] is able to alert the programmer when pre-specified relationships — here, conjunctions of conditions on references — are broken. Off-line query-based debuggers such as the Java Heap Analysis Tool [70] enable the programmer to query dumps of object-oriented heaps for inconsistent data structures. By extending the language, our aim is to maintain relationships such that their debugging is never required.

The database community has a long tradition of relational reasoning, so it is not surprising that objects have acquired some relational features as they have been introduced to databases. Objects in object-oriented databases defined by

the Object Data Management Group (ODMG) [20], for example, may have relationships between them. Also in an object-oriented database, Albano, Ghelli and Orsini [2] propose a strongly-typed language that incorporates inter-object relationships. Their model closely matches the expressiveness of UML associations, but they rely on the richer data model provided by the database system — for example, triggers and transactions. A more lightweight approach was taken by Hwang and Lee [45], who did not address typing or constraints. In both cases, classes are represented as a special case of relationships. Despite these works, however, progress towards inter-object relationships in object-oriented databases has yet to be mirrored in the programming languages that communicate with them.

CHAPTER 2

# RelJ: Relationships for Java

Class-based programming languages form the most popular family of object-oriented languages, which includes Java [38] and C# [55] as well as languages which have class-based features such as C++ [18] and Python [53]. In this chapter, we explore the addition of relationships to a language based on Java, though the ideas may be applied to any similar language. We formalize this language, which we call RelJ, and extend it with facilities to create and manipulate relationships between objects. By giving a formal description to its type system and semantics, we then demonstrate some important safety properties.

An earlier, condensed version of this work has appeared, with Bierman, at the 2005 European Conference on Object-Oriented Programming (ECOOP) [12].

## 2.1 Core language

The Java language provides programmers with a large number of features, but this power makes a formal treatment intractable for the full language. Instead, we select a subset that preserves the essence of Java, but excludes features that are irrelevant to the addition of relationships, and which would serve only to complicate the presentation.

### 2.1.1 Java fragment

We choose a fragment of similar expressivity to other 'middle-weight' Java formalizations [14, 30, 35]. A program in this subset consists of a sequence of simple class declarations, each of which:

i. declares the new class's name, which then represents the type of all instances of the class;

ii. specifies the name of a previously-declared superclass, from which the new class will inherit fields and methods and which will become a supertype of the new class's name;

iii. gives a list of field names and types which join those fields inherited from the superclass; and

iv. gives a list of methods which join (or may override) those methods inherited from the superclass.

We assume that the superclass specifications form a tree of classes, at which `Object` — the class without fields or methods that is implicitly present in all programs — is the root.

Unlike Java, where the superclass need not be specified if it is `Object`, we require that superclasses are always given. Furthermore, we assume all methods have exactly one parameter and always return a value, whereas Java allows any number of parameters and permits methods to take the `void` return type. To further simplify the presentation, objects must be explicitly initialized — constructors are not provided — and we do not permit the use of generics. An example class declaration, which forms part of our running example, is given below:

```
class Student extends Object {
   String name;

   String setName(String newName) {
      this.name = newName;
      return newName;
   }
}
```

Instances of the `Student` class, otherwise referred to as `Student` objects, each have a field called `name` which stores a reference to a `String` object (or `null`). A method, `setName`, which also returns a `String`, allows this field to be set with a string provided by the method's caller. As usual, `this` refers to the receiver; that is, the object on which the method was invoked.

We specialize `Student` by extending it to become `LazyStudent`:

```
class LazyStudent extends Student {
   int hoursOfSleep;

   String setName(String newName) {
      String n = newName.append(this.hoursOfSleep);
      this.name = n;
      return n;
   }
}
```

This class has both a `name` field, inherited from `Student`, and an `hoursOfSleep` field. The `setName` method is overidden so that a student's sleeping time is ap-

pended to his name when it is set, by invoking the append method of the String class, and storing the result in the local variable n.

To simplify the presentation, we do not permit Java's field shadowing or method overloading when extending classes — method names must be unique in each class. We do, however, permit overriding of method implementations.

### 2.1.2 Addition of sets

We do not have generics or casts in this restricted subset of Java, owing to their relative complexity, so we provide first-class, generic sets, much like the generic literal types used by the ODMG [20]. Such a container type is useful in order to process the result of following one-to-many relationships, for example.

Values of set type, written set<$c$> for some class type $c$, are sets of references to $c$-instances. Sets are immutable values: the addition of an element to a set results in a new set. We therefore allow them to be covariant — set<$c$> is a subtype of set<$c'$> where $c$ is a subclass of $c'$. In the presence of generic container classes, even without variance in type parameters, we could give up this container type in favour of a container class without harming our ability to work with relationships.[1]

We also provide an iterator, similar to that of Java 1.5, to examine the elements of a set. For example:

```
set<LazyStudent> sleepers;
  ...
for (LazyStudent s : sleepers) {
   s.snooze(new Dream());
}
```

The variable sleepers is a set of LazyStudent objects. The for iterator sequentially removes an element from the set, binds the element to the variable s, and executes the code in its body. In this case, each LazyStudent has his snooze method invoked.

## 2.2 Relationships

### 2.2.1 Choosing a model of relationships

Broadly speaking, there are two models of relationships relevant to programmers: that of the UML association, and that of the mathematical relation.

#### UML associations

We have already seen how relationships are expressed by associations between classes in a UML class diagram. To recap, we review the features that a software

[1] Of course, some expressivity would be lost owing to the absence of covariance, but this is not vital to the treatment of relationships.

designer may expect from a programming language designed to implement such associations:

**Navigability** In UML it is possible to specify the direction in which an association may be followed. For example, if $A$ is related one-way to $B$, then $A$ can observe its relationship to $B$, but $B$ cannot (directly) observe its relationship to $A$.

**Multiplicity** UML permits multiplicity annotations on associations, which specify constraints on the numbers of objects that may be related by any given association. For example, an association representing 'married-to' may be limited to a one-to-one association between two people, whereas an association representing 'is-child' may be many-to-one: many children may share the same parent. UML provides a rich language for specifying multiplicities, including wildcards, numerical ranges, and lists.

**Aggregation/Composition** Aggregation and composition associations both denote an 'is-part-of' relationship between two classes. For example, an employee may be part of a company. However, UML distinguishes the strength of this association: aggregation generally indicates that the part may be shared with other objects — an employee may work for other companies — whilst composition indicates stronger ownership of the part by the whole — the employee may only work for one company. The difference between aggregation and composition has no bearing on implementation [68].

**Association classes** A class may be added to an association, in which properties relevant to the association but not to the participating classes may be stored. For example, a 'married-to' association may have an association class with the property 'wedding-date'.

Inheritance may exist between association classes.

### Mathematical relations

A mathematical relation is a set whose elements are tuples of elements of its underlying sets. The binary relation $R$ on the sets $X$ and $Y$ is written $R \subseteq (X \times Y)$. When an element $x \in X$ is related to an element $y \in Y$, it may be written $xRy$ or $R(x, y)$, where the latter is more suitable for ternary relationships or those of greater arity.

There are many properties that a mathematician may expect from a programming language abstraction designed to implement relations, of which the most common are summarized here:

**Relation arity** UML associations are binary, but as we have already mentioned, mathematical relations may be have arbitrary arities. For example, the edge relation of a weighted graph holds triples — two nodes and a distance for the edge.

**Multi-relations** Mathematical relations are based on sets and therefore the presence of a tuple is binary — either the objects are related, or they are not. Basing a relation on a multi-set would permit multiple instances of the same relation. For example, two companies may be related by an 'is-contracted-to' relation, but the presence of multiple contracts may imply more than one instance. Of course, this can be encoded by moving from a pair to a triple, with a contract count as the third member.

Relational databases are often based on multi-relations.

**Constraints** There are a number of commonly required constraints on the pairs present in binary relations. Their maintenance usually requires action when new relation elements are added, or when new elements are added to the relation's underlying sets: in total and surjective relations, new pairs are induced by the addition of members to the underlying sets; the addition of a new pair to an injective or functional relation may require the removal of another element; and the addition of new pairs may imply the addition of further pairs to reflexive, transitive or symmetric endorelations. Balzer et al. have surveyed the effect of these constraints on object collaborations in greater detail [9].

### RelJ's model

Not only is there a tension between the mathematical perspective and that of UML, but also with the practicality of respecting such properties statically in a programming language. In this case, we also wish to keep the presentation simple, leaving properties that may be available from modest extensions for later consideration.

In this chapter's presentation of relationships, therefore, our relationships are one-way, many-to-many, and binary. We disregard aggregation and composition associations, given that they do not influence implementation. We do not provide facilities to automatically ensure that relationships remain total, injective, functional, surjective, reflexive, symmetric or transitive.

Finally, we only allow relationships to occur once between each pair of objects; that is, our relations are based on sets rather than multi-sets. The provision of multi-relations is dicussed in Chapter 3, as is the provision of $n$-ary relations, which, as observed above, provide some of the power of multi-relations.

**Terminology** Terminology used for mathematical relations does not match that used for UML associations. We therefore define the terms in which we will couch our discussion of relationships as they are to be presented in this chapter:

**Relationship** Broadly represents the mathematical concept of *relation*, and the UML concept of *association*.

**Relationship instance** Analagous to an element of a mathematical relation, or an instance of a UML *association class*. When discussing instances of a specific relationship, $r$, we will refer to an $r$-instance or $r$-object.

**Participants** Depending on context, refers to the types between which a relationship is defined, or the instances related by a relationship-instance.

### 2.2.2 Declaring relationships

At the top level, a Java program consists of a set of class declarations. A RelJ program adds to this a set of relationship declarations. A relationship declaration expresses an association between pairs of classes or, as we shall see later, between classes and relationships or pairs of relationships.

We earlier gave declarations for `Student` and `LazyStudent`. We would normally expect a student to attend courses, so we declare a `Course` class, which has a field to represent the course's code and some methods (elided here):

```
class Course extends Object {
   String code;
   ...
}
```

To model the attendance of a student at a course, rather than resorting to the structures discussed in Section 1.2, RelJ simply declares the following relationship:

```
relationship Attends extends Relation (Student, Course) {
   int mark;
   Certificate getCertificate(Academic signatory) {
      ...
   }
}
```

The declaration above specifies:

i. the name of the new relationship, `Attends`, which becomes the type for instances of this new relationship;

ii. the name of the new relationship's super-relationship, from which it inherits fields and methods;

iii. the participants in the relationship, `Student` and `Course` — we sometimes refer to the first type as the *source* and the second as the *destination*;

iv. a list of fields, in this case, the field representing the mark that the participating student gets for attending the participating course; and

v. a list of methods that can be invoked on instances of the new relationship; here, a method that returns a `Certificate`, signed by an `Academic`, for the student's course attendance.

Relation is the implicit root of the inheritance tree of relationships, just as Object is for classes. Relation has no methods, and has two special fields, to and from, through which the objects participating in the relationship may be accessed; we give examples later. [2]

In the declaration above, we used two classes as the participants in the Attends relation. However, relationships themselves may take part in further relationships — a feature known as *aggregation* in ER-modelling [66]. For example, suppose a Tutor object is able to recommend that a student attends a particular course. This may be written:

```
relationship Recommends extends Relation (Tutor, Attends) {
   String reason;
}
```

### 2.2.3 Manipulating relationships

Relationships function as factories for relationship instances just as classes function as factories for objects. Classes may be instantiated with new:

```
Student alice = new Student();
Course semantics = new Course();
```

However, relationships may only be instantiated in the course of relating two objects:

```
Attends attnds = Attends.add(alice, semantics);
```

Thus, alice is related to semantics through the Attends relationship. The result of establishing that relationship is a new instance of Attends, in which are stored values for Attends's declared fields, and on which its declared methods may be invoked:

```
int aliceMark = attnds.mark;
Certificate aliceCert = attnds.getCertificate(new Academic());
```

If alice had already been in the Attends relationship with semantics before the invocation of add, then no action would have been taken. The instance of Attends returned by add would have been that which related alice and semantics before the attempted addition.

Whilst the instance of Attends relating alice and semantics may be obtained at the site where the relationship is established, we may obtain the set of all Attends relationship instances pertaining to alice using the ':' operator:

```
set<Attends> aliceAttends = alice:Attends;
```

[2] The fields are 'special' as they are covariant, immutable, and distinguished in the semantics. This note is only to satisfy the curious — to and from are discussed further in Section 2.5.

As discussed previously, our relationships are one-way — the ':' operator may only be applied to the source instance of a relationship. Without this restriction, ambiguity would arise for any relationship whose participants were not distinguishable by type: for example, where a relationship links objects of the same class. The extensions in Chapter 3 include the provision of a bidirectional relationship model.

With the set of Attends instances that relate alice to some course stored in the aliceAttends variable, we may use the for iterator to print the marks Alice obtains for her courses:

```
for (Attends a : alice:Attends) {
   System.println("Alice got " + a.mark
                             + " for " + a.to.code);
}
```

Recall that all relationship instances have a to field, which, in the case of an instance returned by alice:Attends, evaluates to a Course. If we wish to obtain the set of courses that Alice attends, then we may collect these by iterating over the contents of alice:Attends and applying .to to each member. For convenience, we overload the '.' operator to perform the same function:

```
set<Course> aliceCourses = alice.Attends;
```

The '.' operator, just as for ':', may only be applied to the source instance of a relationship.

At some point in the future, it might be the case that Alice stops attending Semantics. In that case, alice and semantics may be unrelated:

```
Attends.rem(alice, semantics);
```

Recall however that references to relationship instances may be stored in variables and fields just like references to any other object, which raises the question of what it means to unrelate two instances. Suppose the following situation:

```
myObj.f = Attends.add(alice, semantics);
Attends x = myObj.f;

Attends.rem(alice, semantics);

set<Course> aliceAttds = alice.Attends;
```

Thus, a reference to the instance relating alice and semantics is stored in both field f of object myObj, and in the variable x. After alice and semantics are unrelated, clearly semantics ought not appear in the set assigned to aliceAttds, but what of the Attends-instance in the heap? The available options include:

**Take no action** The instance is allowed to remain in the heap, but is not returned in the results of relationship access.

```
relationship ActiveRel extends Relation (A, B) {
   boolean isActive() {
      boolean x;
      x = false;
      for (ActiveRel r : this.from:ActiveRel) {
         if (r == this) {
            x = true;
         }
      }
      return x;
   }
}
```

Figure 2.1: Provision of an `isActive` flag for relationship instances

**Delete instance** The instance could be removed from the heap. In this case, however, we have references to the deleted instance in `myObj.f` and in `x`, which may result in an attempt to dereference a dangling pointer. Whilst dangling pointers are allowed in languages such as C++, where they pose a significant security risk, they are not compatible with the abstractions provided by Java or C#.

**Nullify references** To prevent the pursuit of dangling pointers, we might nullify references to the deleted instance. In this case, `x` and `myObj.f` would be set to `null` when `alice` and `semantics` are unrelated. However, this merely results in a null pointer exception at dereference time — more secure, but still potentially surprising.

Given the difficulties with the second and third options, we leave the instance in the heap. The change in state, therefore, is represented by the absence of `semantics` from `alice.Attends`, and of the relating `Attends`-instance from the result of `Alice:attends`. We refer to relationship instances in the heap that are returned by relationship access as *active*, and those no longer returned by relationship access as *inactive*.

We conjecture that storage of the sets returned by the two relationship access operators would in fact be rather rare: it is more likely that the set would be obtained, examined with the `for` iterator, and discarded:

```
for (Course c : alice.Attends) {
   System.out.println("Alice attends " + c.code);
}
```

However, if the programmer requires a flag to determine a particular relationship instance's activity, this may be provided by checking for its membership in the set returned by the '`:`' operator applied to its own source address, as shown in Figure 2.1. Recall that `this.from:ActiveRel` is the set of `ActiveRel`-instances

relating `this.from` to some other object. Clearly any relationship instance will find itself in this set until such time as its participating objects are unrelated.

This idea of active and inactive relationship instances bears some similarity to the idea of extents in object databases [20]. There, a class's extent is the set containing all instances of the class, which may then be queried. The ':' and '.' operators, applied to a relationship, essentially query the extent of active instances for that relationship.

### Relationships and inheritance

Relationship inheritance is defined for the purposes of this chapter exactly as for classes. Specifically, where relationship $r_1$ `extends` $r_2$, then:

- $r_1$ is a subtype of $r_2$: instances of $r_1$ may be assigned to fields/variables whose type is $r_2$;
- $r_1$ inherits all fields and methods of $r_2$ (subject to overriding of method implementations).

Alongside those properties of inheritance common to classes, however, there are additional concerns that arise from the involvement of a relationship's two participating types. In particular, when we declare a relationship, we allow covariance in those types. For example, when extending the `Attends` relationship to `ReluctantlyAttends`, we may express that the relationship only applies to lazy students attending difficult courses:

```
relationship Attends extends Relation
                        (Student, Course) { ... }

relationship ReluctantlyAttends extends Attends
                        (LazyStudent, HardCourse) { ... }
```

where we have already specified that `LazyStudent extends Student` and where we assume that `HardCourse extends Course`.

In turn, the pseudo-fields `to` and `from` are typed covariantly, such that `Attends.to` has type `Course`, but `ReluctantlyAttends.to` has type `HardCourse`. Normally, fields' types are fixed down the inheritance hierarchy.[3] Whilst such covariance is normally unsound, it is available for `to` and `from` because the instances related by any given relationship instance are immutable during the lifetime of that relationship instance:

```
Attends a = ...
a.to = new Student();  //  Not permitted
```

This restriction is enforced syntactically, by precluding `to` and `from` as field names.

[3] We do not consider field shadowing, though shadowed fields cannot be said to be 'the same' as the fields they shadow.

It remains to discuss the effect of inheritance on the results of relationship access with the ':' and '.' operators. Consider the following situation, where lazy student Bob attends two courses:

```
LazyStudent bob = new LazyStudent();
HardCourse  rocketScience = new HardCourse();

Attends.add(bob, semantics);
ReluctantlyAttends.add(bob, rocketScience);
```

It must then be decided what the results of bob:Attends and bob.Attends should be. The choice depends on the extent to which inheritance impacts on the semantics of relationships:

i. if we regard relationship inheritance merely as a reuse and subtyping mechanism, then only semantics should occur in bob.Attends;

ii. if inheritance also implies that ReluctantlyAttends's membership is contained in Attends's membership — just as instances of type ReluctantlyAttends are included in the Attends type — then both semantics and rocketScience should occur in the result of bob.Attends.

Both possibilities are sound — a HardCourse may be included in a set of Course-instances — but both results can be counter-intuitive. In this example, we would certainly expect Bob to attend semantics if he reluctantly attends semantics: the first result seems incomplete. The second option does reflect the intuition that a reluctant attendance is still an attendance, but this strategy gives an unexpected result in the case where we make the following addition:

```
Attends.add(bob, rocketScience);
```

There, rocketScience will occur twice in bob.Attends, once for each of Attends and ReluctantlyAttends. Although one of these attendances stems from ReluctantlyAttends, the result is not compatible with the idea that each pair of objects may only be related once through each relationship. We might remedy this situation by removing duplicates from the results of relationship accesses, but there are two relationship instances for the (bob, rocketScience) pair. If only one is to be returned, it is difficult to decide which — there is not necessarily a 'best' instance. Moving away from our example for a moment, suppose the following triangular relationship inheritance hierarchy:

```
relationship R extends Relation (A, B) { ... }

relationship S extends R(A, B) { ... }
relationship T extends R(A, B) { ... }
```

and that we wish to evaluate the following:

| | |
|---|---|
| $c \in$ ClassName | Class names |
| $r \in$ RelName | Relationship names |
| $f \in$ FldName | Class/relationship field names |
| $m \in$ MethName | Class/relationship method names |
| $x \in$ VarName | Program variables |
| $\iota \in$ Address | Object identifiers/addresses |

Figure 2.2: Names used in RelJ and their meta-variables

```
a = new A();  b = new B();
...
Set<R> rs = a:R;
```

That is, rs receives the set of R-instances relating a to some B-instance. Our goal is to return only one instance for the relationship of a to b through R or any of its sub-relationships, but it quickly emerges that there is no consistent strategy to select an appropriate instance:

- Where a and b are related by R, S and T, then there is no unique, least type. However, R is the unique, greatest type and its instance may be safely returned.
- Where a and b are related only by S and T but not R, then there is no instance of the greatest type that may be returned. In the absence of a unique, least type it only makes sense to return both relationships' instances.

With these considerations in mind, then, we choose the inheritance model that affects only reuse and not relationship membership: when accessing bob:Attends we return only instances whose dynamic type is *precisely* Attends. Similarly, bob.Attends only returns those Course-instances that are related through Attends and not those related through ReluctantlyAttends. The larger result may still be constructed by explicitly accessing Attends and all of its sub-relationships.

We later discuss an alternative model of inheritance that reflects the containment of ReluctantlyAttends in Attends whilst addressing the difficulties discussed above. However, the semantics for this alternative model is significantly more complicated than the semantics given here: in this chapter we therefore proceed with our simple model of inheritance and present the more complicated model as an extension to RelJ in Chapter 3.

## 2.3 Language definition

In this and the following sections, we give a formal description of RelJ, its type system and semantics. A grammar for RelJ programs and types is given in Figure 2.3, which uses the various sets of atomic names defined in Figure 2.2. We assume these sets of names are disjoint.

| | | |
|---|---|---|
| $p \in$ Program $::=$ | ClassDecl$^*$ RelDecl$^*$ $s$ | |
| ClassDecl $::=$ | `class` $c$ `extends` $c'$ { FieldDecl$^*$ MethDecl$^*$ } | |
| RelDecl $::=$ | `relationship` $r$ `extends` $r'$ `(`$n$`,` $n'$`)` { FieldDecl$^*$ MethDecl$^*$ } | |
| $n \in$ RefType $::=$ | $c \mid r$ | |
| $t \in$ Type $::=$ | `boolean` $\mid n \mid$ `set<`$n$`>` | |
| FieldDecl $::=$ | $t\ f$ `;` | |
| MethDecl $::=$ | $t\ m$`(`$t'\ x$`) {` LocalDecl$^*$ $s$ `return` $e$`; }` | |
| LocalDecl $::=$ | $t\ x$ `;` | |
| $mb \in$ MethBody $::=$ | `{` $s$ `return` $e$`; }` | |
| $v \in$ Value $::=$ | `true` $\mid$ `false` $\mid$ `null` $\mid$ `empty` $\mid$ | value literals |
| | $\iota \mid \{\iota_1, \iota_2, \ldots, \iota_{n\geq 0}\}$ | run-time values |
| $l \in$ LValue $::=$ | $x \mid$ | |
| | $e$`.`$f$ | field access |
| $e \in$ Expression $::=$ | $v \mid$ | value |
| | $l \mid$ | l-value |
| | $e_1$ `==` $e_2 \mid$ | equality test |
| | $e_1$ `+` $e_2 \mid e_1$ `-` $e_2 \mid$ | set addition/removal |
| | $e$`.`$r \mid e$`:`$r \mid$ | relationship access |
| | $e$`.from` $\mid$ | relationship source |
| | $e$`.to` $\mid$ | relationship destination |
| | `new` $c$`()` $\mid$ | instantiation |
| | $l$ `=` $e \mid$ | assignment |
| | $r$`.add(`$e$`,`$e'$`)` $\mid r$`.rem(`$e$`,`$e'$`)` $\mid$ | relationship addition/removal |
| | $e$`.`$m$`(`$e'$`)` $\mid$ | method call |
| | `{` $s$ `return` $e$`; }` | method body (run-time only) |
| $s \in$ Statement $::=$ | $\epsilon \mid$ | empty statement |
| | $e$`;` $s_1 \mid$ | expression |
| | `if (`$e$`) {`$s_1$`} else {`$s_2$`};` $s_3 \mid$ | conditional |
| | `for (`$n\ x$ `:` $e$`) {`$s_1$`};` $s_2$ | set iteration |
| $R \in$ Term $::=$ | $s \mid e$ | RelJ terms |

Figure 2.3: The grammar of RelJ types and programs

$$
\begin{aligned}
\mathscr{C} &\in \mathsf{ClassTable} : \mathsf{ClassName} \rightarrow \mathsf{ClassName} \times \mathsf{FieldMap} \times \mathsf{MethMap} \\
\mathscr{R} &\in \mathsf{RelTable} : \mathsf{RelName} \rightarrow \mathsf{RelName} \times \mathsf{RefType} \times \mathsf{RefType} \times \mathsf{FieldMap} \times \mathsf{MethMap} \\
\mathscr{F}, \mathscr{F}^{+} &\in \mathsf{FieldMap} : \mathsf{FldName} \rightarrow \mathsf{Type} \\
\mathscr{M}, \mathscr{M}^{+} &\in \mathsf{MethMap} : \mathsf{MethName} \rightarrow \mathsf{VarName} \times \mathsf{LocalMap} \times \mathsf{Type} \times \mathsf{Type} \times \mathsf{MethBody} \\
\mathscr{L} &\in \mathsf{LocalMap} : \mathsf{VarName} \rightarrow \mathsf{Type}
\end{aligned}
$$

Figure 2.4: Signatures of class and relationship tables

**Types** In common with Java, RelJ types include the names of all the classes defined in the program, ranged over by $c$, as well as the names of primitive types. Here, only `boolean` is provided, but the addition of other primitive types does not significantly impact on the formalization. Added to this are the names of all relationships defined in the program, ranged over by $r$. We use $n$ to range over the names of both classes and relationships, which form the set of *reference types*. Finally, the `set<−>` type constructor may be applied to any reference type, but not to primitive types or set types: sets-of-sets are not permitted. RelJ types are ranged over by $t$, and the set of types available in a program $p$ is written $\mathscr{T}_p$. Context usually renders the subscript redundant, in which case it is omitted.

**Class and relationship declarations** A program $p$ is a set of class declarations, a set of relationship names, and a statement (analagous to Java's `main` method). In common with other calculi, we abstract from the syntax by regarding a program as a collection of maps from names to definitions [29]. A program, $p$, is therefore abstracted by a triple, $\mathscr{P}$, of a class map, a relationship map and a 'main' statement:

$$\mathscr{P} ::= (\mathscr{C}, \mathscr{R}, s)$$

Similarly, class maps and relationship maps provide abstractions of their respective definitions in the syntax.

A class map, $\mathscr{C}$, maps a class name $c$ to a triple of $c$'s superclass, and maps for $c$'s fields and methods:

$$\mathscr{C}(c) = (c', \mathscr{F}, \mathscr{M})$$

We usually refer to $c$'s field map and method map simply as $\mathscr{F}_c$ and $\mathscr{M}_c$ respectively. A class's field map, $\mathscr{F}_c$ merely maps field names, $f$, to the declared type for that field, $t$:

$$\mathscr{F}_c(f) = t$$

Similarly, a method map takes method names, $m$, to a tuple containing $m$'s formal parameter, types for local variables, argument and result types, and method body:

$$\mathscr{M}_c(m) = (x, \mathscr{L}, t_1, t_2, mb)$$

Method bodies are simply statements terminated by a `return` statement, and are ranged over by *mb*. In the abstract syntax, the method body does not include local variable declarations — they are represented by $\mathscr{L}$, which maps variable names to their declared types:

$$\mathscr{L}(x) = t$$

Relationships are represented by a relationship map, $\mathscr{R}$, which maps relationship names, $r$, to a tuple containing the name of $r$'s super-relationship, its participating type and, as for classes, field and method maps:

$$\mathscr{R}(r) = (r', n_1, n_2, \mathscr{F}, \mathscr{M})$$

Recall that the `Object` class is implicitly defined for all programs, as is the `Relation` relationship. We therefore assume the following entries in $\mathscr{C}$ and $\mathscr{R}$:

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

We also formalize the set of a program's valid types in terms of the class and relationship maps:

$$\mathscr{T} \mathrel{\widehat{=}} \{\texttt{boolean}\} \cup \mathsf{dom}(\mathscr{C}) \cup \mathsf{dom}(\mathscr{R}) \cup \{\texttt{set<}n\texttt{>} \mid n \in \mathsf{dom}(\mathscr{C}) \cup \mathsf{dom}(\mathscr{R})\}$$

Finally, we give versions of method and field lookup that respect inheritance, written $\mathscr{M}^+$ and $\mathscr{F}^+$ respectively, and which include fields and methods inherited from supertypes:

$$\mathscr{M}^+_c(m) = \begin{cases} \mathscr{M}_c(m) & \text{if } m \in \mathsf{dom}(\mathscr{M}_c)\\ \mathscr{M}^+_{c'}(m) & \text{otherwise, if } c \neq \texttt{Object} \text{ and } \mathscr{C}(c)_{\mathsf{super}} = c'\end{cases}$$

$\mathscr{M}^+$ for relationships and $\mathscr{F}^+$ are defined similarly. Signatures for all these maps are to be found in Figure 2.4.

Alongside the shorthands for field and method maps, $\mathscr{F}_c$ and $\mathscr{M}_c$, we sometimes use a subscript to project away portions of tuples in which we are not interested. For example, we may write $\mathscr{C}(c)_{\mathsf{super}} = c'$ to express that $c'$ is the superclass of $c$. We extend this shorthand to the results of other lookups:

$$\begin{aligned}\mathscr{R}(r) &= (\mathscr{R}(r)_{\mathsf{super}}, \mathscr{R}(r)_{\mathsf{from}}, \mathscr{R}(r)_{\mathsf{to}}, \mathscr{F}_r, \mathscr{M}_r)\\ \mathscr{M}_c(m) &= (\mathscr{M}_c(m)_{\mathsf{argvar}}, \mathscr{M}_c(m)_{\mathsf{locvars}}, \mathscr{M}_c(m)_{\mathsf{arg}}, \mathscr{M}_c(m)_{\mathsf{ret}}, \mathscr{M}_c(m)_{\mathsf{body}})\end{aligned}$$

A list of such subscripts yields the obvious tuple:

$$\mathscr{M}_c(m)_{\mathsf{arg,ret}} = (t_1, t_2) \text{ where } \mathscr{M}_c(m) = (x, \mathscr{L}, t_1, t_2, mb)$$

**Expressions** Expressions are ranged over by $e$. They include standard features such as variables, field access, equality test and value literals. Some expressions, for example object addresses, $\iota$, and sets of addresses, $\{\iota_1, \ldots, \iota_{i\geq 0}\}$, may not be written in static program text. However, they will occur during execution of terms — these are discussed further in due course. Set manipulation operators for set addition, +, and subtraction, −, are provided, along with a literal for the empty set, `empty`. Sets of addresses are also legitimate values during execution of a program. Relationship access operators are also provided: $e$.$r$ gives access to the objects related to the result of $e$ by relationship $r$; $e$:$r$ gives those instances of $r$ pertaining to the result of $e$. Also included are side-effecting expressions: object allocation with `new`, assignment and method invocation, as well as the relationship addition and removal operators. Finally, method bodies are included in the set of expressions in order to simplify the modelling of method call: the intention is that the method body's statement is executed, after which the block is replaced by the value formed from the `return` expression — this is discussed further in Section 2.5.

Notice that we demand more regular syntax than Java: receivers of method invocations or field accesses must be written in full, and `new` never takes any arguments — we do not consider constructor methods.

**Statements** Statements, ranged over by $s$, may be empty or may be a statement prepended to a further sequence of statements. The set iterator `for`, with syntax matching that of Java 1.5, is provided along with an `if` statement. Expressions may be used as statements when suffixed with a semi-colon but, for simplicity and unlike Java, we do not restrict such 'promotion' only to side-effecting expressions [14, 38].

**Terms** A RelJ term is either a statement or an expression. RelJ terms are ranged over by $R$.

## 2.4 Type system

The rôle of the type system is ostensibly to prevent the invocation of a non-existent method on some object, and to ensure that programs cannot get into states where they may no longer execute (except for result states). The RelJ type system is composed of various judgements, all of which are annotated $\vdash_{\mathbf{R}}$ in order to distinguish them from the judgements given for the object calculus in Chapter 4.

**Subtyping** A subtyping relation, written $\vdash_{\mathbf{R}} t_1 \leq t_2$ where $t_1$ is a subtype of $t_2$, is defined in Figure 2.5. (RSub Class) and (RSub Rel) reflect those subtypes established in class and relationship declarations, whilst (RSub Ref) and (RSub Trans) close these under reflexivity and transitivity. The covariance of `set<−>`

(RSUB CLASS)
$$\frac{\mathscr{C}(c_1)_{\mathsf{super}} = c_2}{\vdash_{\mathbf{R}} c_1 \leq c_2}$$

(RSUB REL)
$$\frac{\mathscr{R}(r_1)_{\mathsf{super}} = r_2}{\vdash_{\mathbf{R}} r_1 \leq r_2}$$

(RSUB SET)
$$\frac{\vdash_{\mathbf{R}} n_1 \leq n_2}{\vdash_{\mathbf{R}} \texttt{set<}n_1\texttt{>} \leq \texttt{set<}n_2\texttt{>}}$$

(RSUB REF)
$$\frac{t \in \mathscr{T}}{\vdash_{\mathbf{R}} t \leq t}$$

(RSUB TRANS)
$$\frac{\vdash_{\mathbf{R}} t_1 \leq t_2 \qquad \vdash_{\mathbf{R}} t_2 \leq t_3}{\vdash_{\mathbf{R}} t_1 \leq t_3}$$

(RSUB OBJECT)
$$\frac{}{\vdash_{\mathbf{R}} \texttt{Relation} \leq \texttt{Object}}$$

Figure 2.5: Definition of RelJ's subtyping relation

types is established in (RSUB SET), and we join the relationship and class hierarchies — desirable in the absence of generics — by making `Relation` a subtype of `Object` in (RSUB OBJECT).

**Typing environments** We type expressions and statements in the presence of a typing environment, $\Gamma \in \mathsf{TypeEnv}$, which maps variable names and addresses to types:

$$\Gamma(x) = t \qquad\qquad \Gamma(\iota) = n$$

We shall discuss the circumstances under which addresses, $\iota$, receive bindings in $\Gamma$ when discussing the semantics; for the moment we shall consider only variable bindings. Notice that $\Gamma$'s signature contains that of method local variable maps, $\mathscr{L}$. We can therefore extend type environments with local variable maps when, for example, entering the scope of a method body as long as $\Gamma$ and $\mathscr{L}$ assign types to disjoint sets of variables (local variable maps do not include addresses in their domains).

$$\Gamma' = \Gamma \uplus \mathscr{L}$$

$\Gamma'$ then agrees with $\Gamma$ and $\mathscr{L}$ unless their domains overlap, in which case the union is not defined. A type environment may be updated with $[x \mapsto t]$:

$$\Gamma' = \Gamma[x \mapsto t]$$

Here, $\Gamma'$ agrees with $\Gamma$ except for $x$, which it maps to $t$.

**Typing relations** Type checking is performed with two typing relations. The first indicates that an expression $e$ has type $t$, whilst the other checks that a statement $s$ is well-typed, both in the presence of environment $\Gamma$:

$$\Gamma \vdash_{\mathbf{R}} e : t \qquad\qquad \Gamma \vdash_{\mathbf{R}} s$$

The rules that define these typing relations are given in Figure 2.6.

(RTyp Sub)
$$\frac{\Gamma \vdash_{\mathbf{R}} e : t_1 \qquad \vdash_{\mathbf{R}} t_1 \leq t_2}{\Gamma \vdash_{\mathbf{R}} e : t_2}$$

(RTyp Bool)
$$\frac{}{\Gamma \vdash_{\mathbf{R}} \texttt{true} : \texttt{boolean} \quad \Gamma \vdash_{\mathbf{R}} \texttt{false} : \texttt{boolean}}$$

(RTyp NullEmpty)
$$\frac{n \in \mathscr{T}}{\Gamma \vdash_{\mathbf{R}} \texttt{null} : n \quad \Gamma \vdash_{\mathbf{R}} \texttt{empty} : \texttt{set<}n\texttt{>}}$$

(RTyp Var)
$$\frac{\Gamma(x) = t}{\Gamma \vdash_{\mathbf{R}} x : t}$$

(RTyp Addr)
$$\frac{\Gamma(\iota) = n}{\Gamma \vdash_{\mathbf{R}} \iota : n}$$

(RTyp Set)
$$\frac{\forall j \in 1..i : \Gamma \vdash_{\mathbf{R}} \iota_j : n}{\Gamma \vdash_{\mathbf{R}} \{\iota_1, \ldots, \iota_i\} : \texttt{set<}n\texttt{>}}$$

(RTyp New)
$$\frac{c \in \mathscr{T}}{\Gamma \vdash_{\mathbf{R}} \texttt{new}\ c\texttt{()} : c}$$

(RTyp Eq)
$$\frac{\Gamma \vdash_{\mathbf{R}} e_1 : n \qquad \Gamma \vdash_{\mathbf{R}} e_2 : n'}{\Gamma \vdash_{\mathbf{R}} e_1 \texttt{ == } e_2 : \texttt{boolean}}$$

(RTyp Ass)
$$\frac{\Gamma(x) = t \quad \Gamma \vdash_{\mathbf{R}} e : t}{\Gamma \vdash_{\mathbf{R}} x \texttt{ = } e : t} \quad x \neq \texttt{this}$$

(RTyp Fld)
$$\frac{\mathscr{F}_n^+(f) = t \quad \Gamma \vdash_{\mathbf{R}} e : n}{\Gamma \vdash_{\mathbf{R}} e\texttt{.}f : t}$$

(RTyp FldAss)
$$\frac{\mathscr{F}_n^+(f) = t \quad \Gamma \vdash_{\mathbf{R}} e_1 : n \qquad \Gamma \vdash_{\mathbf{R}} e_2 : t}{\Gamma \vdash_{\mathbf{R}} e_1\texttt{.}f \texttt{ = } e_2 : t}$$

(RTyp Body)
$$\frac{\Gamma \vdash_{\mathbf{R}} s \quad \Gamma \vdash_{\mathbf{R}} e : t}{\Gamma \vdash_{\mathbf{R}} \texttt{\{ } s\, \texttt{return } e\texttt{; \}} : t}$$

(RTyp Add)
$$\frac{\Gamma \vdash_{\mathbf{R}} e_1 : \texttt{set<}n\texttt{>} \qquad \Gamma \vdash_{\mathbf{R}} e_2 : n}{\Gamma \vdash_{\mathbf{R}} e_1 \texttt{ + } e_2 : \texttt{set<}n\texttt{>}}$$

(RTyp Subtract)
$$\frac{\Gamma \vdash_{\mathbf{R}} e_1 : \texttt{set<}n\texttt{>} \qquad \Gamma \vdash_{\mathbf{R}} e_2 : n}{\Gamma \vdash_{\mathbf{R}} e_1 \texttt{ - } e_2 : \texttt{set<}n\texttt{>}}$$

(RTyp RelObj)
$$\frac{\mathscr{R}(r)_{\mathsf{from,to}} = (n_1, n_2) \quad \Gamma \vdash_{\mathbf{R}} e : n_1}{\Gamma \vdash_{\mathbf{R}} e\texttt{.}r : \texttt{set<}n_2\texttt{>}}$$

(RTyp RelInst)
$$\frac{\mathscr{R}(r)_{\mathsf{from}} = n \quad \Gamma \vdash_{\mathbf{R}} e : n}{\Gamma \vdash_{\mathbf{R}} e\texttt{:}r : \texttt{set<}r\texttt{>}}$$

(RTyp From)
$$\frac{\mathscr{R}(r)_{\mathsf{from}} = n \quad \Gamma \vdash_{\mathbf{R}} e : r}{\Gamma \vdash_{\mathbf{R}} e\texttt{.from} : n}$$

(RTyp To)
$$\frac{\mathscr{R}(r)_{\mathsf{to}} = n \quad \Gamma \vdash_{\mathbf{R}} e : r}{\Gamma \vdash_{\mathbf{R}} e\texttt{.to} : n}$$

(RTyp RelAdd)
$$\frac{\mathscr{R}(r)_{\mathsf{from,to}} = (n_1, n_2) \quad \Gamma \vdash_{\mathbf{R}} e_1 : n_1 \qquad \Gamma \vdash_{\mathbf{R}} e_2 : n_1}{\Gamma \vdash_{\mathbf{R}} r\texttt{.add(}e_1\texttt{,}e_2\texttt{)} : r}$$

(RTyp RelRem)
$$\frac{\mathscr{R}(r)_{\mathsf{from,to}} = (n_1, n_2) \quad \Gamma \vdash_{\mathbf{R}} e_1 : n_1 \qquad \Gamma \vdash_{\mathbf{R}} e_2 : n_2}{\Gamma \vdash_{\mathbf{R}} r\texttt{.rem(}e_1\texttt{,}e_2\texttt{)} : r}$$

(RTyp Skip)
$$\frac{}{\Gamma \vdash_{\mathbf{R}} \epsilon}$$

(RTyp Exp)
$$\frac{\Gamma \vdash_{\mathbf{R}} se : t \quad \Gamma \vdash_{\mathbf{R}} s}{\Gamma \vdash_{\mathbf{R}} se\texttt{;}\ s}$$

(RTyp Call)
$$\frac{\mathscr{M}_n^+(m)_{\mathsf{arg,ret}} = (t_1, t_2) \quad \Gamma \vdash_{\mathbf{R}} e_1 : n \qquad \Gamma \vdash_{\mathbf{R}} e_2 : t_1}{\Gamma \vdash_{\mathbf{R}} e_1\texttt{.}m\texttt{(}e_2\texttt{)} : t_2}$$

(RTyp Cond)
$$\frac{\Gamma \vdash_{\mathbf{R}} e : \texttt{boolean} \quad \Gamma \vdash_{\mathbf{R}} s_1 \qquad \Gamma \vdash_{\mathbf{R}} s_2 \qquad \Gamma \vdash_{\mathbf{R}} s_3}{\Gamma \vdash_{\mathbf{R}} \texttt{if (}e\texttt{) \{}s_1\texttt{\} else \{}s_2\texttt{\};}\ s_3}$$

(RTyp For)
$$\frac{\Gamma \vdash_{\mathbf{R}} e : \texttt{set<}n\texttt{>} \qquad \Gamma[x \mapsto n] \vdash_{\mathbf{R}} s_1 \quad \Gamma \vdash_{\mathbf{R}} s_2 \qquad x \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash_{\mathbf{R}} \texttt{for (}n\ x \texttt{ : } e\texttt{) \{}s_1\texttt{\};}\ s_2}$$

Figure 2.6: Subsumption-based typing rules for RelJ expressions, $\Gamma \vdash_{\mathbf{R}} e : t$, and statements, $\Gamma \vdash_{\mathbf{R}} s$

Most of these rules are standard. (RTYP SUB) implements subtype polymorphism by allowing a term to be viewed at a supertype of any previously assigned type. (RTYP BOOL) assigns the `boolean` type to boolean literals, whilst (RTYP NULLEMPTY) allows `null` to take type $n$ and `empty` to take type `set`<$n$> for any reference type $n$ defined in the program. Variables' types are taken directly from the environment in (RTYP VAR), as are those of addresses and address sets in (RTYP ADDR) and (RTYP SET). In the presence of subsumption, sets may contain addresses of several types as long as they have a common supertype. New class instances are given the obvious type in (RTYP NEW), as long as the class is defined in the program, whilst the equality test for object references in (RTYP EQ) returns a `boolean` as long as the operands have reference types.[4] Variable update is typed by (RTYP ASS), which requires that the variable's new value can be typed with the variable's declared type.

The result of field access takes its type from the field's definition in (RTYP FLD). Field assignment, typed by (RTYP FLDASS), requires that the updated value can be given the field's declared type. The result of a field assignment has the field's type, in order to support chaining of assignments.

(RTYP BODY) checks that the constituent statement and expression are typable, and propagates the return expression's type to the outside of the block.

The set addition and subtraction operators are typed by (RTYP ADD) and (RTYP SUBTRACT) respectively. In both cases, the affected set and the item being added (or removed) must agree on the type of the set's elements.

Relationship access is typed by (RTYP RELOBJ) and (RTYP RELINST). When accessing the objects related to the receiver, typed by (RTYP RELOBJ), the result will be a set of the relationship's destination type. When accessing the instances of the relationship pertaining to the receiver, the result will be a set of relationship instances, as specified by (RTYP RELINST).

The `to` and `from` operators may be applied to any relationship instance, and are typed directly from the relationship's definition in (RTYP TO) and (RTYP FROM) respectively.

Relationship addition is typed by (RTYP RELADD) directly from the relationship's declaration, just as for (RTYP RELREM). In both cases, the type of the result is simply the relationship type: for addition, this reflects the desire to return the newly-created relationship instance for field initialization; in the case of deletion, the inactivated relationship instance may be useful for housekeeping.

**Syntax-directed typing** The type-rules of Figure 2.6 are not syntax-directed owing to the presence of the subsumption rule, (RTYP SUB). As a result, a type-checker may choose to apply a rule that later results in an unsuccessful derivation, even though the term in question is typable — backtracking will be required to check all possibilities before a term can be rejected.

[4] This is a relaxation of the Java rule, which requires that the types be directly comparable. In this case, we only require that the types are both subtypes of `Object`.

(RTyp Bool$^{\text{alg}}$)
$$\frac{}{\Gamma \vdash_{\mathbf{R}} \texttt{true} : \texttt{boolean} \quad \Gamma \vdash_{\mathbf{R}} \texttt{false} : \texttt{boolean}}$$

(RTyp NullEmpty$^{\text{alg}}$)
$$\frac{n \in \mathscr{T}}{\Gamma \vdash_{\mathbf{R}} \texttt{null} : n \quad \Gamma \vdash_{\mathbf{R}} \texttt{empty} : \texttt{set<}n\texttt{>}}$$

(RTyp Var$^{\text{alg}}$)
$$\frac{\Gamma(x) = t}{\Gamma \vdash_{\mathbf{R}} x : t}$$

(RTyp Addr$^{\text{alg}}$)
$$\frac{\Gamma(\iota) = n}{\Gamma \vdash_{\mathbf{R}} \iota : n}$$

(RTyp Set$^{\text{alg}}$)
$$\frac{n = \bigsqcup\{\Gamma(\iota_1), \ldots, \Gamma(\iota_i)\}}{\Gamma \vdash_{\mathbf{R}} \{\iota_1, \ldots, \iota_i\} : \texttt{set<}n\texttt{>}}$$

(RTyp New$^{\text{alg}}$)
$$\frac{c \in \mathscr{T}}{\Gamma \vdash_{\mathbf{R}} \texttt{new}\ c\texttt{()} : c}$$

(RTyp Eq$^{\text{alg}}$)
$$\frac{\Gamma \vdash_{\mathbf{R}} e_1 : n \qquad \Gamma \vdash_{\mathbf{R}} e_2 : n'}{\Gamma \vdash_{\mathbf{R}} e_1 \ \texttt{==}\ e_2 : \texttt{boolean}}$$

(RTyp Ass$^{\text{alg}}$)
$$\frac{\Gamma(x) = t_1 \qquad \Gamma \vdash_{\mathbf{R}} e : t_2 \qquad \vdash_{\mathbf{R}} t_2 \leq t_1}{\Gamma \vdash_{\mathbf{R}} x \ \texttt{=}\ e : t_1} \quad x \neq \texttt{this}$$

(RTyp Fld$^{\text{alg}}$)
$$\frac{\Gamma \vdash_{\mathbf{R}} e : n \qquad \mathscr{F}_n^+(f) = t}{\Gamma \vdash_{\mathbf{R}} e\texttt{.}f : t}$$

(RTyp FldAss$^{\text{alg}}$)
$$\frac{\Gamma \vdash_{\mathbf{R}} e_1 : n \qquad \mathscr{F}_n^+(f) = t_1 \qquad \Gamma \vdash_{\mathbf{R}} e_2 : t_2 \qquad \vdash_{\mathbf{R}} t_2 \leq t_1}{\Gamma \vdash_{\mathbf{R}} e_1\texttt{.}f \ \texttt{=}\ e_2 : t_1}$$

(RTyp Body$^{\text{alg}}$)
$$\frac{\Gamma \vdash_{\mathbf{R}} s \qquad \Gamma \vdash_{\mathbf{R}} e : t}{\Gamma \vdash_{\mathbf{R}} \texttt{\{}\ s\,\texttt{return}\ e\texttt{; \}} : t}$$

(RTyp Add$^{\text{alg}}$)
$$\frac{\Gamma \vdash_{\mathbf{R}} e_1 : \texttt{set<}n_1\texttt{>} \qquad \Gamma \vdash_{\mathbf{R}} e_2 : n_2 \qquad n_3 = n_1 \sqcup n_2}{\Gamma \vdash_{\mathbf{R}} e_1 \ \texttt{+}\ e_2 : \texttt{set<}n_3\texttt{>}}$$

(RTyp Subtract$^{\text{alg}}$)
$$\frac{\Gamma \vdash_{\mathbf{R}} e_1 : \texttt{set<}n_1\texttt{>} \qquad \Gamma \vdash_{\mathbf{R}} e_2 : n_2}{\Gamma \vdash_{\mathbf{R}} e_1 \ \texttt{-}\ e_2 : \texttt{set<}n_1\texttt{>}}$$

(RTyp RelObj$^{\text{alg}}$)
$$\frac{\mathscr{R}(r)_{\mathsf{from,to}} = (n_2, n_3) \qquad \Gamma \vdash_{\mathbf{R}} e : n_1 \qquad \vdash_{\mathbf{R}} n_1 \leq n_2}{\Gamma \vdash_{\mathbf{R}} e\texttt{.}r : \texttt{set<}n_3\texttt{>}}$$

(RTyp RelInst$^{\text{alg}}$)
$$\frac{\mathscr{R}(r)_{\mathsf{from}} = n_2 \qquad \Gamma \vdash_{\mathbf{R}} e : n_1 \qquad \vdash_{\mathbf{R}} n_1 \leq n_2}{\Gamma \vdash_{\mathbf{R}} e\texttt{:}r : \texttt{set<}r\texttt{>}}$$

(RTyp From$^{\text{alg}}$)
$$\frac{\mathscr{R}(r)_{\mathsf{from}} = n \qquad \Gamma \vdash_{\mathbf{R}} e : r}{\Gamma \vdash_{\mathbf{R}} e\texttt{.from} : n}$$

(RTyp To$^{\text{alg}}$)
$$\frac{\mathscr{R}(r)_{\mathsf{to}} = n \qquad \Gamma \vdash_{\mathbf{R}} e : r}{\Gamma \vdash_{\mathbf{R}} e\texttt{.to} : n}$$

(RTyp RelAdd$^{\text{alg}}$)
$$\frac{\mathscr{R}(r)_{\mathsf{from,to}} = (n_1, n_2) \qquad \Gamma \vdash_{\mathbf{R}} e_1 : n_3 \qquad \Gamma \vdash_{\mathbf{R}} e_2 : n_4 \qquad \vdash_{\mathbf{R}} n_3 \leq n_1 \qquad \vdash_{\mathbf{R}} n_4 \leq n_2}{\Gamma \vdash_{\mathbf{R}} r\texttt{.add(}e_1\texttt{,}e_2\texttt{)} : r}$$

(RTyp RelRem$^{\text{alg}}$)
$$\frac{\mathscr{R}(r)_{\mathsf{from,to}} = (n_1, n_2) \qquad \Gamma \vdash_{\mathbf{R}} e_1 : n_3 \qquad \Gamma \vdash_{\mathbf{R}} e_2 : n_4 \qquad \vdash_{\mathbf{R}} n_3 \leq n_1 \qquad \vdash_{\mathbf{R}} n_4 \leq n_2}{\Gamma \vdash_{\mathbf{R}} r\texttt{.rem(}e_1\texttt{,}e_2\texttt{)} : r}$$

(RTyp Skip$^{\text{alg}}$)
$$\frac{}{\Gamma \vdash_{\mathbf{R}} \epsilon}$$

(RTyp Exp$^{\text{alg}}$)
$$\frac{\Gamma \vdash_{\mathbf{R}} e : t \qquad \Gamma \vdash_{\mathbf{R}} s}{\Gamma \vdash_{\mathbf{R}} e\texttt{;}\ s}$$

(RTyp Call$^{\text{alg}}$)
$$\frac{\Gamma \vdash_{\mathbf{R}} e_1 : n \qquad \Gamma \vdash_{\mathbf{R}} e_2 : t_1 \qquad \mathscr{M}_n^+(m)_{\mathsf{arg,ret}} = (t_2, t_3) \qquad \vdash_{\mathbf{R}} t_1 \leq t_2}{\Gamma \vdash_{\mathbf{R}} e_1\texttt{.}m\texttt{(}e_2\texttt{)} : t_3}$$

(RTyp Cond$^{\text{alg}}$)
$$\frac{\Gamma \vdash_{\mathbf{R}} e : \texttt{boolean} \qquad \Gamma \vdash_{\mathbf{R}} s_1 \qquad \Gamma \vdash_{\mathbf{R}} s_2 \qquad \Gamma \vdash_{\mathbf{R}} s_3}{\Gamma \vdash_{\mathbf{R}} \texttt{if (}e\texttt{) \{}s_1\texttt{\} else \{}s_2\texttt{\};}\ s_3}$$

(RTyp For$^{\text{alg}}$)
$$\frac{\Gamma \vdash_{\mathbf{R}} e : \texttt{set<}n_1\texttt{>} \qquad \Gamma[x \mapsto n_2] \vdash_{\mathbf{R}} s_1 \qquad \Gamma \vdash_{\mathbf{R}} s_2 \qquad \vdash_{\mathbf{R}} n_1 \leq n_2 \qquad x \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash_{\mathbf{R}} \texttt{for (}n_2\ x \texttt{ : } e\texttt{) \{}s_1\texttt{\};}\ s_2}$$

Figure 2.7: Alternative, syntax-directed typing rules for RelJ

Figure 2.7 gives a version of the type system that suggests a more practical type-checking algorithm: subtyping is applied only when necessary, and a type-checker can determine, for any term, which type rule will be applicable. The types of compound terms are determined uniquely from the types of sub-terms.

For the purposes of this discussion, we write $\vdash^{\mathbf{sub}}_{\mathbf{R}}$ and $\vdash^{\mathbf{alg}}_{\mathbf{R}}$ to distinguish the typing relations derived from the subsumption-based rules of Figure 2.6 and syntax-directed rules of Figure 2.7 respectively. We refer to the two resulting calculi as $\mathsf{RelJ}^{\mathbf{sub}}$ and $\mathsf{RelJ}^{\mathbf{alg}}$.

Most of the $\mathsf{RelJ}^{\mathbf{alg}}$ rules are similar in form to those from $\mathsf{RelJ}^{\mathbf{sub}}$, except for those rules that type sets and their operations. Rather than assigning sets a type formed from an arbitrary bound on its constituent addresses' types, we take the *least* such bound. The least upper bound of a collection of types, written $\bigsqcup\{t_1, \ldots, t_n\}$, is the closest super-type of $t_{1..i}$. Formally:

**Definition 2.1** (Least upper bound). *The least upper bound (LUB) of two types $t_1$ and $t_2$ is written $t_1 \sqcup t_2$ and is the unique type that:*

- *is a supertype of $t_1$ and $t_2$: $\vdash_{\mathbf{R}} t_1 \leq t_1 \sqcup t_2$ and $\vdash_{\mathbf{R}} t_2 \leq t_1 \sqcup t_2$*
- *is a subtype of all other such types:*

$$\textit{for all } t_3 : \vdash_{\mathbf{R}} t_1 \leq t_3 \textit{ and } \vdash_{\mathbf{R}} t_2 \leq t_3 \textit{ implies } \vdash_{\mathbf{R}} t_1 \sqcup t_2 \leq t_3$$

*The definition extends to the least upper bounds of sets of types:*

$$\bigsqcup\{t_1, t_2, \ldots, t_n\} = t_1 \sqcup t_2 \sqcup \ldots \sqcup t_n$$

*The empty set of types has no least upper bound: $\bigsqcup\emptyset$ is undefined. The least upper bound of a singleton set is the singleton itself: $\bigsqcup\{t\} = t$.*

That least upper bounds exist for all non-empty sets of types arises, via standard order theory, from the properties of subtyping discussed in Section 2.6.1.

(RTYP ADD$^{\mathbf{alg}}$) is are similarly modified, so that the bound on the type is uniquely determined from the types of the sub-terms. (RTYP SUBTRACT$^{\mathbf{alg}}$) assigns the type of the old set to the result set, as the removal of an element cannot weaken the type.

The remaining rules have simply been extended with subtyping at the use-sites of sub-terms, with the exception of sub-terms that are the target of field-, method- or relationship-lookup: recall that the lookup functions respect inheritance by definition.

The subsumption-based and syntax-directed type systems are equally expresssive, but not strictly equivalent. All derivable typings in $\mathsf{RelJ}^{\mathbf{alg}}$ are derivable in $\mathsf{RelJ}^{\mathbf{sub}}$, but the reverse does not hold. However, of all the types that can be assigned to a term in $\mathsf{RelJ}^{\mathbf{sub}}$, the 'best' of those types — that is, the least type according to the subtype relation — will be derivable for the term in $\mathsf{RelJ}^{\mathbf{alg}}$. No operation permitted by $\mathsf{RelJ}^{\mathbf{sub}}$ is excluded by $\mathsf{RelJ}^{\mathbf{alg}}$.

(RWD FIELD)

$$\frac{\mathscr{C}(n)_{\mathsf{super}} = n' \text{ or } \mathscr{R}(n)_{\mathsf{super}} = n' \qquad f \notin \mathsf{dom}(\mathscr{F}^{+}_{n'}) \qquad \mathscr{F}_n(f) \in \mathscr{T}}{n \vdash_{\mathbf{R}} f}$$

(RWD METHOD)

$$\frac{\begin{array}{ll} 1: & \mathscr{C}(n)_{\mathsf{super}} = n' \text{ or } \mathscr{R}(n)_{\mathsf{super}} = n' \\ 2: & \mathscr{M}_n(m) = (x, \mathscr{L}, t_1, t_2, \texttt{\{ } s \texttt{ return } e\texttt{; \}}) \\ 3: & t_1, t_2 \in \mathscr{T} \\ 4: & \texttt{this}, x \notin \mathsf{dom}(\mathscr{L}) \\ 5: & \{x \mapsto t_1, \texttt{this} \mapsto n\} \cup \mathscr{L} \vdash_{\mathbf{R}} \texttt{\{ } s \texttt{ return } e\texttt{; \}} : t_2 \\ 6: & m \in \mathsf{dom}(\mathscr{M}^{+}_{n'}) \text{ implies } \Rightarrow \vdash_{\mathbf{R}} \mathscr{M}^{+}_{n'}(m)_{\mathsf{arg}} \leq t_1 \text{ and } \vdash_{\mathbf{R}} t_2 \leq \mathscr{M}^{+}_{n'}(m)_{\mathsf{ret}} \end{array}}{n \vdash_{\mathbf{R}} m}$$

Figure 2.8: Definition of well-declared fields and method

**Theorem 2.1.** *The type systems of* RelJ$^{\mathbf{sub}}$ *and* RelJ$^{\mathbf{alg}}$ *are equivalent up to subtyping. More precisely, for all* $\Gamma$*,* $e$ *and* $t$*:*

$$\Gamma \vdash^{\mathbf{sub}}_{\mathbf{R}} e : t \textit{ iff there exists } t' \textit{ such that } \vdash_{\mathbf{R}} t' \leq t \textit{ and } \Gamma \vdash^{\mathbf{alg}}_{\mathbf{R}} e : t'$$

*The systems are equivalent for statements:* $\Gamma \vdash^{\mathbf{sub}}_{\mathbf{R}} s$ *iff* $\Gamma \vdash^{\mathbf{alg}}_{\mathbf{R}} s$

*Proof.* The proof proceeds by demonstrating that the properties above are closed under the rules of the appropriate type system. For example, we need to show, for each rule in RelJ$^{\mathbf{alg}}$, that the rule's consequence can be derived from its antecedents using the rules of RelJ$^{\mathbf{sub}}$. This follows by induction and the application of the subsumption rule at the points indicated by subtyping in the RelJ$^{\mathbf{alg}}$ rules.

The process is similar in the other direction: we need only show that a derivation exists in RelJ$^{\mathbf{alg}}$ that takes the antecedents of each RelJ$^{\mathbf{sub}}$ rule to its consequence, but potentially at a subtype. This holds vacuously for the subsumption rule, (RTYP SUB), and follows by induction and the properties of least upper bounds for the syntax-directed rules. □

For the remainder of this work, we use the subsumption-based type system as the basis for RelJ.

**Well-declared fields and methods** There are a number of constraints imposed upon field and method declarations in classes and relationships, many of which were discussed earlier. These are now formalized in the rules of Figure 2.8.

When a field $f$ or method $m$ is well-declared in a class or relationship $n$, we write:

$$n \vdash_{\mathbf{R}} f \qquad\qquad n \vdash_{\mathbf{R}} m$$

A field is well-declared according to (RWD FIELD), in which it is checked that a field in class (or relationship) $n$ has not been declared in $n$'s direct or indirect su-

perclasses (or super-relationships), and that the field's declared type is contained with the set of the program's recognized types.

A method is well-declared according to (RWD METHOD), the lines of which have been numbered for ease of explanation. The rule checks:

1. the super-type of the class/relationship;
2. the method's declaration;
3. that the method's input and return types are valid;
4. that the parameter name and `this` do not clash with any of the method's local variable names;
5. that the method body is well-typed with the method's return type when the parameter, `this` and local variables are assigned the types specified in the method definition;
6. that the input type of this method is a supertype of any previous declaration of $m$ in a supertype of $n$, 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.

In fact, we only need to check that the method's argument type is in the set of program types, $\mathcal{T}$, explicitly. The return type is included in the subtyping relation and, as we shall see later, this relation only includes valid types.

**Well-formed types** Figure 2.9 defines a relation specifying the well-formedness of RelJ types. We write $\vdash_{\mathbf{R}} t$ when $t$ is a well-formed type. Primitive types, in this case only `boolean`, are always well-formed as expressed in (RVT BOOL). Similarly, the implicitly-defined `Object` and `Relation` types are always well-formed, according to (RVT OBJECT) and (RVT RELATION) respectively.

(RVT CLASS) specifies that a class type is well-formed if its superclass is declared, and if all of its methods and fields are well-declared. (RVT RELATIONSHIP) imposes many of the same restrictions as (RVT CLASS), but also checks that the relationship's participating types are subtypes of its super-relationship's respective participating types.

**Well-formed programs** Finally, the program is well-formed, written $\vdash_{\mathbf{R}} \diamond$, if it conforms to (RPROGRAM) of Figure 2.9. Specifically, all declared classes and relationships must be well-formed and the subtyping relation must be antisymmetric.

By ensuring the asymmetry of the subtyping relation we preclude declaration of classes as subclasses of themselves. Notice also that, in a well-formed program, all valid types $t \in \mathcal{T}$ are well-formed.

We only consider well-formed programs.

(RVT Bool)
$$\frac{}{\vdash_{\mathbf{R}} \texttt{boolean}}$$

(RVT Object)
$$\frac{}{\vdash_{\mathbf{R}} \texttt{Object}}$$

(RVT Relation)
$$\frac{}{\vdash_{\mathbf{R}} \texttt{Relation}}$$

(RVT Set)
$$\frac{\vdash_{\mathbf{R}} n}{\vdash_{\mathbf{R}} \texttt{set<}n\texttt{>}}$$

(RVT Class)
$$\frac{\begin{array}{l} c \neq \texttt{Object} \\ c, \mathscr{C}(c)_{\mathsf{super}} \in \mathsf{dom}(\mathscr{C}) \\ \forall f \in \mathsf{dom}(\mathscr{F}_c) : c \vdash_{\mathbf{R}} f \\ \forall m \in \mathsf{dom}(\mathscr{M}_c) : c \vdash_{\mathbf{R}} m \end{array}}{\vdash_{\mathbf{R}} c}$$

(RVT Relationship)
$$\frac{\begin{array}{l} r \neq \texttt{Relation} \qquad r' = \mathscr{R}(r)_{\mathsf{super}} \\ r, r' \in \mathsf{dom}(\mathscr{R}) \\ \forall f \in \mathsf{dom}(\mathscr{F}_r) : r \vdash_{\mathbf{R}} f \\ \forall m \in \mathsf{dom}(\mathscr{M}_r) : r \vdash_{\mathbf{R}} m \\ \vdash_{\mathbf{R}} \mathscr{R}(r)_{\mathsf{from}} \leq \mathscr{R}(r')_{\mathsf{from}} \\ \vdash_{\mathbf{R}} \mathscr{R}(r)_{\mathsf{to}} \leq \mathscr{R}(r')_{\mathsf{to}} \end{array}}{\vdash_{\mathbf{R}} r}$$

---

(RProgram)
$$\frac{\begin{array}{l} \mathscr{P} = (\mathscr{C}, \mathscr{R}, s) \\ \forall n \in \mathsf{dom}(\mathscr{C}) \cup \mathsf{dom}(\mathscr{R}) : \ \vdash_{\mathbf{R}} n \\ \forall n_1, n_2 : \ \vdash_{\mathbf{R}} n_1 \leq n_2 \text{ and } \vdash_{\mathbf{R}} n_2 \leq n_1 \Rightarrow n_1 = n_2 \\ \emptyset \vdash_{\mathbf{R}} s \end{array}}{\vdash_{\mathbf{R}} \diamond}$$

Figure 2.9: Well-formed types and programs

## 2.5 Semantics

The semantics defines the dynamic behaviour of a program. With this formal model of execution, we may go on to show that the type system imposes enough restrictions to enforce certain properties: that is, that well-typed programs are also well-behaved.

**Evaluation contexts** We use evaluation contexts to specify the evaluation order of expressions and statements [77]. Evaluation contexts for RelJ are given in Figure 2.10. There are two kinds of context: expression contexts, denoted $\mathscr{E}_{\mathrm{e}}$, control evaluation order of sub-expressions inside expressions; statement contexts, denoted $\mathscr{E}_{\mathrm{s}}$, control evaluation of sub-expressions inside statements. $\mathscr{E}$ ranges over both kinds of context.

Each context contains exactly one hole, written $\bullet$, which represents the position of the sub-expression to be executed first. In the grammar, the unique hole is normally found in a sub-context: for example, the grammar contains $\mathscr{E}_{\mathrm{e}}\texttt{.}m(e)$ which specifies that the receiver of a method is evaluated first. Only when the receiver has been reduced to a value does evaluation of the parameter begin, hence $v\texttt{.}m(\mathscr{E}_{\mathrm{e}})$. The evaluation order is largely as expected: expressions and statements are evaluated left-to-right. Execution never occurs under `for` or `if`,

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

Figure 2.10: Grammar for RelJ evaluation contexts

or in the tail of a sequence of statements.

A context is made into an expression or statement by substituting the hole with some expression. We write $\mathscr{E}_e[e]$ when expression $e$ is substituted for the hole in $\mathscr{E}_e$. Thus, expression contexts may be regarded as endofunctions on expressions and statement contexts as functions from expressions to statements.

$$\mathscr{E}_e \in \text{ExpContext} : \text{Expression} \rightarrow \text{Expression}$$
$$\mathscr{E}_s \in \text{StatContext} : \text{Expression} \rightarrow \text{Statement}$$

**Variable substitution** Substitution of variables in program syntax is written $e[x'/x]$ for the replacement of all variables $x$ in $e$ with $x'$, and similarly with statements, $s[x'/x]$. We do not need to consider binding: substitution is a straight-forward replacement.

**Errors** In common with many class-based object-oriented languages, RelJ yields an error when asked to dereference `null`. In order to avoid complicating the presentation with incidental details, error catching is not permitted: when an error is thrown, it is immediately propagated to the top level and execution is

halted. To accomplish this, the set of errors, ranged over by $w$, is defined (adding Error to the evaluation contexts' domains):

$$w \in \mathsf{Error} ::= \mathsf{NullPtrError} \mid \mathscr{E}_{\mathsf{e}}[w] \mid \mathscr{E}_{\mathsf{s}}[w]$$

Thus, any expression or statement is an error when it contains an error.

**Heaps and objects** Classes and relationships are instantiated during execution, either with the new operator in the case of classes, or when objects are placed in a relationship.

Objects are finite maps from field names to field values, annotated with the instance's *dynamic type* — the name of the class or relationship that was instantiated to create the object. We write an instance of class $c$, with fields $f_1, f_2, \ldots, f_i$ each having value $v_1, v_2, \ldots, v_i$ as follows:

$$o ::= \langle\!\langle c \| f_1 : v_1, f_2 : v_2, \ldots, f_i : v_i \rangle\!\rangle$$

The heap stores a program's objects, each at a unique address. Addresses, ranged over by $\iota$, are infinite — we do not model limited memory or garbage collection.

Heaps are ranged over by $\sigma$, which is a finite map from addresses to objects:

$$\sigma \in \mathsf{Heap} ::= \{\iota_1 \mapsto o_1, \iota_2 \mapsto o_2, \ldots, \iota_i \mapsto o_{i \geq 0}\}$$

Relationship instances are similar to class instances, but are also annotated with references to the objects they relate. An $r$-instance relating $\iota_1$ and $\iota_2$ is therefore written:

$$\langle\!\langle r, \iota_1, \iota_2 \| f_1 : v_1, f_2 : v_2, \ldots, f_i : v_i \rangle\!\rangle$$

For both stores and objects, we use the usual map notation for access and update. In the case of objects, these apply only to their field-value map:

$$o(f_2) = v_2 \qquad o[f_4 \mapsto v'] \qquad \sigma(\iota) = o \qquad \sigma[\iota \mapsto o']$$

As before, we use subscript notation to access an object's dynamic type:

$$o = \langle\!\langle c \| \ldots \rangle\!\rangle \text{ if and only if } o_{\mathsf{dyntyp}} = c$$

**Relationship store** The relationship store holds the pairs of addresses related through each relationship and, for each such pair, the address of the relationship instance that exists between the two objects. It therefore acts as abstraction of the reference structures discussed in Section 1.2 — rather than requiring the participating objects to hold references to the relationship instance, its presence is simply recorded in $\rho$. It maps a pair of instances, $\iota_1$ and $\iota_2$, and a relationship name, $r$, to the address of the $r$-instance that relates $\iota_1$ and $\iota_2$:

$$\rho(r, \iota_1, \iota_2) = \iota'$$

$$
\begin{aligned}
\iota &\in \mathsf{Address} \\
o \in \mathsf{Object} &::= \langle\!\langle c \| f_1 : v_1, \ldots, f_i : v_{i\geq 0} \rangle\!\rangle \mid \langle\!\langle r, \iota_1, \iota_2 \| f_1 : v_1, \ldots, f_i : v_{i\geq 0} \rangle\!\rangle \\
\sigma \in \mathsf{Heap} &::= \{\iota_1 \mapsto o_1, \ldots, \iota_i \mapsto o_{i\geq 0}\} \\
\rho \in \mathsf{RelStore} &::= \{(r_1, \iota_1, \iota_1') \mapsto \iota_1'', \ldots, (r_i, \iota_i, \iota_i') \mapsto \iota_{i\geq 0}''\} \\
\gamma \in \mathsf{LocalStore} &::= \{x_1 \mapsto v_1, \ldots, x_i \mapsto v_{i\geq 0}\} \\
w \in \mathsf{Error} &::= \mathsf{NullPtrError} \mid \mathscr{E}_\mathrm{e}[w] \mid \mathscr{E}_\mathrm{s}[w] \\
\mathsf{Config} &::= \langle \sigma, \rho, \gamma, R \rangle \mid \langle \sigma, \rho, \gamma, w \rangle
\end{aligned}
$$

Figure 2.11: The sets and meta-variables supporting RelJ's semantics

As a map, it enforces our invariant that each relationship occurs only once between each pair of addresses. The store may be updated or extended in the usual way. Unlike objects and heaps, it may also be shrunk:

$$\rho' = \rho \setminus \{(r, \iota_1, \iota_2) \mapsto \iota'\}$$

**Local variables store** A local variable store maps variable names to their values, and is written $\gamma$:

$$\gamma \in \mathsf{LocalStore} ::= \{x_1 \mapsto v_1, x_2 \mapsto v_2, \ldots, x_n \mapsto v_n\}$$

The local variable store models a program stack, except it has no framing structure. Instead, variable names are freshened upon entry to new scopes; this is discussed further when the semantics of expressions is defined.

**Configurations** The semantics is given by transitions between *configurations*, each of which represents the state of a program at some point in its execution. Each configuration consists the three stores discussed above, as well as the term which is to be executed.

$$\langle \sigma, \rho, \gamma, e \rangle \qquad\qquad \langle \sigma, \rho, \gamma, s \rangle$$

Instead of a term, a configuration may contain an error, $w$, in which case this is known as an *error configuration*.

**Well-formedness** The rules of Figure 2.12 describe what it means for these structures — objects, heaps, relationship stores and local variable stores — to be well-formed by applying the typing relation from the previous section to their component values.

(RTYP COBJECT) checks a class instance for well-formedness by ensuring all its fields have values which conform to the fields' declared types. (RTYP ROBJECT)

(RTyp CObject)

$$\frac{\begin{array}{l}\mathsf{dom}(\mathscr{F}_c^+) = \{f_1,\ldots,f_i\}\\ \forall j \in 1..i : \Gamma \vdash_{\mathbf{R}} v_j : \mathscr{F}_c^+(f_j)\end{array}}{\Gamma \vdash_{\mathbf{R}} \langle\!\langle c \| f_1 : v_1, \ldots, f_i : v_i \rangle\!\rangle}$$

(RTyp Heap)

$$\frac{\begin{array}{l}\forall \iota \in \mathsf{dom}(\sigma)\\ \quad \Gamma \vdash_{\mathbf{R}} \sigma(\iota) \qquad \Gamma(\iota) = \sigma(\iota)_{\mathsf{dyntyp}}\end{array}}{\Gamma \vdash_{\mathbf{R}} \sigma}$$

(RTyp RObject)

$$\frac{\begin{array}{l}\mathsf{dom}(\mathscr{F}_r^+) = \{f_1,\ldots,f_i\}\\ \forall j \in 1..i : \Gamma \vdash_{\mathbf{R}} v_j : \mathscr{F}_n^+(f_j)\\ \Gamma \vdash_{\mathbf{R}} \iota_1 : \mathscr{R}(r)_{\mathsf{from}}\\ \Gamma \vdash_{\mathbf{R}} \iota_2 : \mathscr{R}(r)_{\mathsf{to}}\end{array}}{\Gamma \vdash_{\mathbf{R}} \langle\!\langle r, \iota_1, \iota_2 \| f_1 : v_1, \ldots, f_i : v_i \rangle\!\rangle}$$

(RTyp RelStore)

$$\frac{\begin{array}{l}\forall (r, \iota_1, \iota_2) \in \mathsf{dom}(\rho)\\ \quad \Gamma \vdash_{\mathbf{R}} \iota_1 : \mathscr{R}(r)_{\mathsf{from}}\\ \quad \Gamma \vdash_{\mathbf{R}} \iota_2 : \mathscr{R}(r)_{\mathsf{to}}\\ \quad \Gamma \vdash_{\mathbf{R}} \rho(r, \iota_1, \iota_2) : r\end{array}}{\Gamma \vdash_{\mathbf{R}} \rho}$$

(RTyp Locals)

$$\frac{\forall x \in \mathsf{dom}(\gamma) : \Gamma \vdash_{\mathbf{R}} \gamma(x) : \Gamma(x)}{\Gamma \vdash_{\mathbf{R}} \gamma}$$

(RTyp ExpConfig)

$$\frac{\begin{array}{l}\mathsf{dom}(\Gamma) = \mathsf{dom}(\sigma) \cup \mathsf{dom}(\gamma)\\ \Gamma \vdash_{\mathbf{R}} \sigma \qquad \Gamma \vdash_{\mathbf{R}} \rho \qquad \Gamma \vdash_{\mathbf{R}} \gamma \qquad \Gamma \vdash_{\mathbf{R}} e : t\end{array}}{\Gamma \vdash_{\mathbf{R}} \langle \sigma, \rho, \gamma, e \rangle : t}$$

(RTyp StatConfig)

$$\frac{\begin{array}{l}\mathsf{dom}(\Gamma) = \mathsf{dom}(\sigma) \cup \mathsf{dom}(\gamma)\\ \Gamma \vdash_{\mathbf{R}} \sigma \qquad \Gamma \vdash_{\mathbf{R}} \rho \qquad \Gamma \vdash_{\mathbf{R}} \gamma \qquad \Gamma \vdash_{\mathbf{R}} s\end{array}}{\Gamma \vdash_{\mathbf{R}} \langle \sigma, \rho, \gamma, s \rangle}$$

| **Judgement** | **Rule** | |
|---|---|---|
| $\Gamma \vdash_{\mathbf{R}} o$ | (RTyp CObject) | The values of fields in class instance $o$ agree with their declarations in $o$'s class. |
| $\Gamma \vdash_{\mathbf{R}} o$ | (RTyp RObject) | As above for relationship instance $o$, and the types of the addresses related by $o$ agree with $o$'s relationship declaration. |
| $\Gamma \vdash_{\mathbf{R}} \sigma$ | (RTyp Heap) | The objects in the heap $\sigma$ are well-formed, and their types agree with those assigned to their addresses in environment $\Gamma$. |
| $\Gamma \vdash_{\mathbf{R}} \rho$ | (RTyp RelStore) | The entries in the relationship store, $\rho$, agree with the program's relationship declarations and with $\Gamma$. |
| $\Gamma \vdash_{\mathbf{R}} \gamma$ | (RTyp Locals) | The values of variables in the local variable store, $\gamma$, agree with their types in $\Gamma$. |
| $\Gamma \vdash_{\mathbf{R}} \langle \sigma, \rho, \gamma, e \rangle : t$ | (RTyp ExpConfig) | The components of the configuration are well-formed, and $\Gamma$ only types addresses and variables that exist in $\sigma$ and $\gamma$. |
| $\Gamma \vdash_{\mathbf{R}} \langle \sigma, \rho, \gamma, s \rangle$ | (RTyp StatConfig) | As above, for statement configurations. |

Figure 2.12: Rules specifying well-formedness conditions for structures used in the semantics, with corresponding informal description.

performs a similar function for relationship instances with additional checks to ensure that the source and destination addresses have types appropriate to the relationship's declaration. (RTYP HEAP) maps these conditions across all the objects in the heap, as well as ensuring that every heap address has an entry in $\Gamma$ appropriate to its object's dynamic type. (RTYP RELSTORE) ensures the addresses in each entry of the relationship store have types appropriate to each relationship's declaration. (RTYP LOCALS) ensures every variable's value in the local variable store, $\gamma$, can be typed according to the variable's entry in the environment, $\Gamma$.

Finally, configurations are well-formed with respect to some environment, $\Gamma$, if $\Gamma$ has an entry for every address of the heap, $\sigma$, and for every variable in the local variable store, $\gamma$, and no more. This condition ensures, for example, that no address is typable (see (RTYP ADDR) of Figure 2.6) which does not exist in the heap: that is, there are no dangling pointers. The remaining conditions ensure that the heap, the local variable store, the relationship store, $\rho$, and the configuration's term are well-typed. If the term is an expression, its type is assigned to the entire configuration.

**Execution relation** We now define the execution relation, $\leadsto_{\mathbf{R}}$, which relates starting configurations to configurations that result from one step in program execution.

$$\leadsto_{\mathbf{R}} \subset \mathsf{Config} \times \mathsf{Config}$$

The relation is constructed from a collection of axioms, which specify the behaviour of the smallest executable expressions and statements, and is then closed under (RRED CONTEXT), which is shown at the top of Figure 2.13 and which allows sub-terms to be reduced inside larger terms.

It then remains to define the base cases for construction of the $\leadsto_{\mathbf{R}}$ relation. The rules governing expression execution are given in Figure 2.13 underneath (RRED CONTEXT). The `empty` value literal is translated into $\emptyset$ by (RRED EMPTY). Variables' values are taken directly from the local variable store, $\gamma$, by (RRED VAR). Variable assignment is performed by updating $\gamma$, in (RRED VARASS). (RRED EQ) and (RRED NEQ) compare two values for equality; the type system ensures this is restricted to heap addresses for which, as a countable set, we have a good notion of equality.

New objects are allocated by (RRED NEW), which uses the $\mathsf{new}(c)$ auxilliary function, given in Figure 2.14, which creates an object with all the appropriate fields for an object of type $c$, including those inherited from superclasses. Each field is assigned an initial value by the $\mathsf{initial}$ function, which returns a suitable value based on each field's type. Once this object has been created, it is given a fresh address in the heap, $\sigma$, which is updated accordingly.

The addition and removal of elements from a set is handled by (RRED ADD) and (RRED SUB). Again, we rely on the type system to ensure that the only values encountered here are sets of addresses.

Field access, in (RRED FLD), looks up the receiver address in the heap, then

(RRED CONTEXT)

$$\frac{\langle \sigma, \rho, \gamma, R \rangle \rightsquigarrow_{\mathbf{R}} \langle \sigma', \rho', \gamma', R' \rangle}{\langle \sigma, \rho, \gamma, \mathscr{E}[R] \rangle \rightsquigarrow_{\mathbf{R}} \langle \sigma', \rho', \gamma', \mathscr{E}[R'] \rangle}$$

(RRED EMPTY) $\langle \sigma, \rho, \gamma, \texttt{empty} \rangle \rightsquigarrow_{\mathbf{R}} \langle \sigma, \rho, \gamma, \emptyset \rangle$

(RRED VAR) $\langle \sigma, \rho, \gamma, x \rangle \rightsquigarrow_{\mathbf{R}} \langle \sigma, \rho, \gamma, \gamma(x) \rangle$

(RRED VARASS) $\langle \sigma, \rho, \gamma, x \texttt{ = } v \rangle \rightsquigarrow_{\mathbf{R}} \langle \sigma, \rho, \gamma[x \mapsto v], v \rangle$

(RRED EQ) $\langle \sigma, \rho, \gamma, v \texttt{ == } v \rangle \rightsquigarrow_{\mathbf{R}} \langle \sigma, \rho, \gamma, \texttt{true} \rangle$

(RRED NEQ) $\langle \sigma, \rho, \gamma, v \texttt{ == } v' \rangle \rightsquigarrow_{\mathbf{R}} \langle \sigma, \rho, \gamma, \texttt{false} \rangle$ where $v \neq v'$

(RRED NEW) $\langle \sigma, \rho, \gamma, \texttt{new } c\texttt{()} \rangle \rightsquigarrow_{\mathbf{R}} \langle \sigma[\iota \mapsto \mathsf{new}(c)], \rho, \gamma, \iota \rangle$ where $\iota \notin \mathsf{dom}(\sigma)$

(RRED ADD) $\langle \sigma, \rho, \gamma, v \texttt{ + } \iota \rangle \rightsquigarrow_{\mathbf{R}} \langle \sigma, \rho, \gamma, v \cup \{\iota\} \rangle$

(RRED SUB) $\langle \sigma, \rho, \gamma, v \texttt{ - } \iota \rangle \rightsquigarrow_{\mathbf{R}} \langle \sigma, \rho, \gamma, v \setminus \{\iota\} \rangle$

(RRED FLD) $\langle \sigma, \rho, \gamma, \iota\texttt{.}f \rangle \rightsquigarrow_{\mathbf{R}} \langle \sigma, \rho, \gamma, \sigma(\iota)(f) \rangle$

(RRED FLDASS) $\langle \sigma, \rho, \gamma, \iota\texttt{.}f \texttt{ = } v \rangle \rightsquigarrow_{\mathbf{R}} \langle \sigma[\iota \mapsto \sigma(\iota)[f \mapsto v]], \rho, \gamma, v \rangle$

(RRED CALL) $\langle \sigma, \rho, \gamma_1, \iota\texttt{.}m\texttt{(}v\texttt{)} \rangle \rightsquigarrow_{\mathbf{R}} \langle \sigma, \rho, \gamma_2, \texttt{\{ } s_2 \texttt{ return } e_2\texttt{; \}} \rangle$

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

$\mathscr{M}_n^+(m) = (x, \mathscr{L}, t_1, t_2, \texttt{\{ } s_1 \texttt{ return } e_1\texttt{; \}})$

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

$x', x'_{\texttt{this}}, x'_1, \ldots, x'_i \notin \mathsf{dom}(\gamma_1)$

$\gamma_2 = \gamma_1[x' \mapsto v][x'_{\texttt{this}} \mapsto \iota][x'_{1..i} \mapsto \mathsf{initial}(\mathscr{L}(x_{1..i}))]$

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

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

(RRED RETURN) $\langle \sigma, \rho, \gamma, \texttt{\{ return } v\texttt{; \}} \rangle \rightsquigarrow_{\mathbf{R}} \langle \sigma, \rho, \gamma, v \rangle$

(RRED RELOBJ) $\langle \sigma, \rho, \gamma, \iota\texttt{.}r \rangle \rightsquigarrow_{\mathbf{R}} \langle \sigma, \rho, \gamma, \{\iota' \mid \exists \iota'' : \rho(r, \iota, \iota') = \iota''\} \rangle$

(RRED RELINST) $\langle \sigma, \rho, \gamma, \iota\texttt{:}r \rangle \rightsquigarrow_{\mathbf{R}} \langle \sigma, \rho, \gamma, \{\iota'' \mid \exists \iota' : \rho(r, \iota, \iota') = \iota''\} \rangle$

(RRED FROM) $\langle \sigma, \rho, \gamma, \iota\texttt{.from} \rangle \rightsquigarrow_{\mathbf{R}} \langle \sigma, \rho, \gamma, \iota' \rangle$ where $\sigma(\iota) = \langle\!\langle _, \iota', _ \| _ \rangle\!\rangle$

(RRED TO) $\langle \sigma, \rho, \gamma, \iota\texttt{.to} \rangle \rightsquigarrow_{\mathbf{R}} \langle \sigma, \rho, \gamma, \iota' \rangle$ where $\sigma(\iota) = \langle\!\langle _, _, \iota' \| _ \rangle\!\rangle$

(RRED RELADD) $\langle \sigma_1, \rho_1, \gamma, r\texttt{.add(}\iota_1\texttt{,}\iota_2\texttt{)} \rangle \rightsquigarrow_{\mathbf{R}} \langle \sigma_2, \rho_2, \gamma, \iota_3 \rangle$

where $(\sigma_2, \rho_2) = \mathsf{addRel}(r, \iota_1, \iota_2, \sigma_1, \rho_1)$

$\iota_3 = \rho_2(r, \iota_1, \iota_2)$

(RRED RELREM1) $\langle \sigma, \rho_1, \gamma, r\texttt{.rem(}\iota_1\texttt{,}\iota_2\texttt{)} \rangle \rightsquigarrow_{\mathbf{R}} \langle \sigma, \rho_2, \gamma, \rho_1(r, \iota_1, \iota_2) \rangle$

where $(r, \iota_1, \iota_2) \in \mathsf{dom}(\rho_1)$

$\rho_2 = \rho_1 \setminus \{(r, \iota_1, \iota_2) \mapsto \rho_1(r, \iota_1, \iota_2)\}$

(RRED RELREM2) $\langle \sigma, \rho, \gamma, r\texttt{.rem(}\iota_1\texttt{,}\iota_2\texttt{)} \rangle \rightsquigarrow_{\mathbf{R}} \langle \sigma, \rho, \gamma, \texttt{null} \rangle$

where $(r, \iota_1, \iota_2) \notin \mathsf{dom}(\rho)$

Figure 2.13: Transition relation for RelJ expressions

$$\mathsf{new}(c) = \langle\!\langle c \| f_1 : \mathsf{initial}(\mathscr{F}_c^+(f_1)), \ldots, f_i : \mathsf{initial}(\mathscr{F}_c^+(f_i))^{\,f_i \in \mathsf{dom}(\mathscr{F}_c^+)} \rangle\!\rangle$$

$$\mathsf{new}(r, \iota_1, \iota_2) = \langle\!\langle r, \iota_1, \iota_2 \| f_1 : \mathsf{initial}(\mathscr{F}_r^+(f_1)), \ldots, f_i : \mathsf{initial}(\mathscr{F}_r^+(f_i))^{\,f_i \in \mathsf{dom}(\mathscr{F}_r^+)} \rangle\!\rangle$$

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

$$\mathsf{addRel}(r, \iota_1, \iota_2, \sigma, \rho) = \begin{cases} (\sigma, \rho) & \text{if } (r, \iota_1, \iota_2) \in \mathsf{dom}(\rho) \\ (\sigma[\iota \mapsto \mathsf{new}(r, \iota_1, \iota_2)], \rho[(r, \iota_1, \iota_2) \mapsto \iota]) & \text{otherwise} \end{cases}$$

$$\text{where } \iota \notin \mathsf{dom}(\sigma)$$

Figure 2.14: Functions auxiliary to the semantics

looks up the accessed field in the resulting object. Similarly, (RRED FLDASS) implements field update by looking up the object in the heap, updating it, then updating the heap so that the object's address maps to the newly updated object.

(RRED CALL) implements method call. The receiver object is first found in the heap in order to establish its dynamic type, $n$. The method's definition is then determined from $\mathscr{M}_n^+$. Fresh variables — variables not present in the locals store — are then found to act as `this`, as the method's formal parameter, and as the method's local variables. We substitute these fresh variable names for those used in the method's body. By freshening variables in this way, there is no need to keep an explicit stack [30]. Finally, the local variable store is updated so that the variable acting as the method's parameter obtains the value given as the method call's argument; the variable taking the place of `this` receives the address of the call's receiver; and local variables receive appropriate initial values dependent on their declared type. The result of this execution step is the method body from $\mathscr{M}_n^+$, with variables freshened: recall that method bodies are expressions, and that the evaluation contexts of Figure 2.10 permit execution inside method bodies.

Once execution of a method body has completed, only the enclosing block and `return` statement will remain: it will have the form `{ return` $v$`; }`. This value is unwrapped by (RRED RETURN), eliminating the block created by (RRED CALL). For example, a method invocation used in a variable assignment may execute in the following way:

$$\begin{array}{lll} x \texttt{ = } \iota\texttt{.}m(v) \rightsquigarrow_{\mathrm{R}} x \texttt{ = \{ } s \texttt{ return } e\texttt{; \}} & & \text{(RRED CALL)} \\ \rightsquigarrow_{\mathrm{R}} \ldots \rightsquigarrow_{\mathrm{R}} x \texttt{ = \{ return } v'\texttt{; \}} & & \text{Execution inside method body} \\ \rightsquigarrow_{\mathrm{R}} x \texttt{ = } v' & & \text{(RRED RETURN)} \end{array}$$

(RRED STAT) $\langle \sigma, \rho, \gamma, v\, ;\, s \rangle \rightsquigarrow_{\mathbf{R}} \langle \sigma, \rho, \gamma, s \rangle$

(RRED CONDT) $\langle \sigma, \rho, \gamma, \texttt{if (true) } \{s_1\} \texttt{ else } \{s_2\}; s_3 \rangle \rightsquigarrow_{\mathbf{R}} \langle \sigma, \rho, \gamma, s_1\, s_3 \rangle$

(RRED CONDF) $\langle \sigma, \rho, \gamma, \texttt{if (false) } \{s_1\} \texttt{ else } \{s_2\}; s_3 \rangle \rightsquigarrow_{\mathbf{R}} \langle \sigma, \rho, \gamma, s_2\, s_3 \rangle$

(RRED FORI) $\langle \sigma, \rho, \gamma, \texttt{for } (n\, x : \emptyset)\ \{s_1\}; s_2 \rangle \rightsquigarrow_{\mathbf{R}} \langle \sigma, \rho, \gamma, s_2 \rangle$

(RRED FOR2) $\langle \sigma, \rho, \gamma_1, \texttt{for } (n\, x : v)\ \{s_1\}; s_2 \rangle \rightsquigarrow_{\mathbf{R}}$

$$\langle \sigma, \rho, \gamma_2, s_1' \texttt{ for } (n\, x : (v \setminus \{\iota\}))\ \{s_1\}; s_2 \rangle$$

where
$$\iota \in v$$
$$x' \notin \mathsf{dom}(\gamma_1)$$
$$\gamma_2 = \gamma_1[x' \mapsto \iota]$$
$$s_1' = s_1[x'/x]$$

Figure 2.15: Transition relation for RelJ statements

The remaining rules of Figure 2.13 specify the manipulation of relationships. (RRED RELOBJ) and (RRED RELINST) express relationship access. Both build a set of addresses based on the contents of $\rho$ for a given receiver, $\iota$ and relationship $r$: (RRED RELOBJ) implements the '`.`' operator, and returns the addresses related to $\iota$ in $\rho$; (RRED RELINST) implements the '`:`' operator, and returns the relating instances in $\rho$.

The (RRED FROM) and (RRED TO) rules simply extract the source and destination references from the relationship instance retrieved from the heap. Notice that there are no corresponding update rules.

Relationship addition is handled by (RRED RELADD). The resulting configuration has a new heap and relationship store, which are obtained from the auxilliary function addRel. addRel's definition is given in Figure 2.14. It takes a heap, $\sigma$, relationship store, $\rho$, a pair of addresses, $\iota_1$ and $\iota_2$, and the relationship through which the addresses should be related, $r$. If $\iota_1$ and $\iota_2$ are already related through $r$ — that is, $\rho(r, \iota_1, \iota_2)$ is defined — then addRel returns the heap and relationship store unchanged. Otherwise, a new instance of $r$ is added to $\sigma$, and $\rho$ is updated such that $\rho(r, \iota_1, \iota_2)$ maps to the new $r$-instance's address in $\sigma$. Notice that there is a version of new that accepts two addresses alongside a relationship name, which creates $r$-objects just as new($c$) creates $c$-objects.

Relationship removal has two rules: (RRED RELREMI) governs the result of relationship removal when the addresses given are related in $\rho$ — the appropriate entry in $\rho$ is simply deleted, the heap is left unchanged, and the address of the inactivated $r$-instance is returned. When the addresses are not related through the given relationship then, under (RRED RELREM2), no action is taken and `null` is returned.

Figure 2.15 gives the semantics for statements. The rules are largely standard: (RRED STAT) throws away values encountered in statement context and (RRED CONDT) and (RRED CONDF) select a branch of an `if`-statement based on

| | |
|---|---|
| (RRED ADDN) | $\langle \sigma, \rho, \gamma, v \texttt{ + null}\rangle \rightsquigarrow_{\mathrm{R}} \langle \sigma, \rho, \gamma, \mathsf{NullPtrError}\rangle$ |
| (RRED SUBN) | $\langle \sigma, \rho, \gamma, v \texttt{ - null}\rangle \rightsquigarrow_{\mathrm{R}} \langle \sigma, \rho, \gamma, \mathsf{NullPtrError}\rangle$ |
| (RRED FLDN) | $\langle \sigma, \rho, \gamma, \texttt{null}.f\rangle \rightsquigarrow_{\mathrm{R}} \langle \sigma, \rho, \gamma, \mathsf{NullPtrError}\rangle$ |
| (RRED FLDASSN) | $\langle \sigma, \rho, \gamma, \texttt{null}.f \texttt{ = } v\rangle \rightsquigarrow_{\mathrm{R}} \langle \sigma, \rho, \gamma, \mathsf{NullPtrError}\rangle$ |
| (RRED CALLN) | $\langle \sigma, \rho, \gamma, \texttt{null}.m(v)\rangle \rightsquigarrow_{\mathrm{R}} \langle \sigma, \rho, \gamma, \mathsf{NullPtrError}\rangle$ |
| (RRED RELINSTN) | $\langle \sigma, \rho, \gamma, \texttt{null:}r\rangle \rightsquigarrow_{\mathrm{R}} \langle \sigma, \rho, \gamma, \mathsf{NullPtrError}\rangle$ |
| (RRED RELOBJN) | $\langle \sigma, \rho, \gamma, \texttt{null}.r\rangle \rightsquigarrow_{\mathrm{R}} \langle \sigma, \rho, \gamma, \mathsf{NullPtrError}\rangle$ |
| (RRED FROMN) | $\langle \sigma, \rho, \gamma, \texttt{null.from}\rangle \rightsquigarrow_{\mathrm{R}} \langle \sigma, \rho, \gamma, \mathsf{NullPtrError}\rangle$ |
| (RRED TON) | $\langle \sigma, \rho, \gamma, \texttt{null.to}\rangle \rightsquigarrow_{\mathrm{R}} \langle \sigma, \rho, \gamma, \mathsf{NullPtrError}\rangle$ |
| (RRED RELADDN) | $\left.\begin{array}{l}\langle \sigma, \rho, \gamma, r\texttt{.add(}\iota\texttt{,null)}\rangle \\ \langle \sigma, \rho, \gamma, r\texttt{.add(null,}\iota\texttt{)}\rangle \\ \langle \sigma, \rho, \gamma, r\texttt{.add(null,null)}\rangle\end{array}\right\} \rightsquigarrow_{\mathrm{R}} \langle \sigma, \rho, \gamma, \mathsf{NullPtrError}\rangle$ |
| (RRED RELREMN) | $\left.\begin{array}{l}\langle \sigma, \rho, \gamma, r\texttt{.rem(}\iota\texttt{,null)}\rangle \\ \langle \sigma, \rho, \gamma, r\texttt{.rem(null,}\iota\texttt{)}\rangle \\ \langle \sigma, \rho, \gamma, r\texttt{.rem(null,null)}\rangle\end{array}\right\} \rightsquigarrow_{\mathrm{R}} \langle \sigma, \rho, \gamma, \mathsf{NullPtrError}\rangle$ |

Figure 2.16: Error transitions for RelJ

the value of the test condition. Execution of the `for` iterator is specified by (RRED FOR1) and (RRED FOR2). In the former, iterations over empty sets are discarded. In the latter case, the iterator variable is freshened in the loop's body, and the locals store is updated so that this freshened variable is bound to an element of the target set. The resulting statement starts with the (freshened) body statement and is followed by the original `for` statement, with its target set missing the element selected earlier.

Finally, the axioms in Figure 2.16 specify the conditions under which a null-pointer error is generated. Such an error will result from any attempt: to access or update a field on `null`; to invoke a method on `null`; to add `null` to or subtract `null` from a set; to relate or un-relate `null` through a relationship; to access the related or relating instances of `null`; or to access the source or destination pointers of `null`.

## 2.6 Soundness

In this section we outline proofs of two key safety properties for RelJ:

**Type preservation** We wish to ensure that types of terms are preserved as they are executed. By checking that any execution step from a well-typed term

results in a new well-typed term, we can be assured that the requirements imposed by the typing rules are enforced throughout execution.

**Progress** If a term is a value, then we consider execution to have completed. Otherwise, the progress theorem specifies that a well-typed term may make an execution step to a new state — at least one of the rules defining the semantics must be applicable.

Both of these properties hold for RelJ, so RelJ is *sound*. A term will always execute, by a series of steps, to a well-defined error state or to a value whose type matches that assigned to the original term.

The remainder of this section summarizes the key steps of the soundness proof.

### 2.6.1 Properties of subtyping

The subtyping relation essentially gives permission for values of one type to be viewed at another type, so it is of primary importance that types cannot 'break out' of their type class such that they become the target of operations which they cannot perform: for example, the ability to observe a `boolean` at reference type would be unsound, as a boolean value cannot satisfy method invocations. The following lemma specifies, amongst other properties, that such a situation does not arise.

**Lemma 2.2.** *The subtyping relation,* $\vdash_{\mathbf{R}} t \leq t'$, *is the disjoint union of three join-semilattices: one for reference types, one for set types, and one (singleton) for booleans. Each is closed under subtyping, with greatest elements* `Object`, `set<Object>` *and* `boolean` *respectively.*

*The relationship types form a sub-tree in the reference-type portion of the relation, with* `Relation` *as its greatest element. Relationship types are downwards-closed under subtyping.*

*Proof.* The properties (including reflexivity, transitivity and anti-symmetry) are shown by induction: they are closed under the rules generating the subtyping relation. □

The disjointness of the three portions of the subtyping relation ensures that subtype polymorphism does not allow a value to be viewed at an inappropriate type. That the portions are join-semilattices ensures that non-empty least upper bounds exist in all three portions of the subtyping relation, as required by the syntax-directed typing rules of Figure 2.7.

It is also important that fields and methods cannot become unavailable down the inheritance hierarchy within the type class of reference types. Again, this ensures that an object is never asked for a method or field it cannot provide:

**Lemma 2.3.** *Field and method declarations respect subtyping:*

$$\vdash_{\mathbf{R}} n \leq n' \text{ implies } \begin{cases} \mathscr{F}^{+}_{n'} \subseteq \mathscr{F}^{+}_{n} \\ m \in dom(\mathscr{M}^{+}_{n'}) \text{ implies } m \in dom(\mathscr{M}^{+}_{n}) \\ \qquad \text{and } \vdash_{\mathbf{R}} \mathscr{M}^{+}_{n'}(m)_{arg} \leq \mathscr{M}^{+}_{n}(m)_{arg} \\ \qquad \text{and } \vdash_{\mathbf{R}} \mathscr{M}^{+}_{n}(m)_{ret} \leq \mathscr{M}^{+}_{n'}(m)_{ret} \end{cases}$$

*Proof.* Again, proof is by induction on the rules defining the subtype relation. □

### 2.6.2 Preserving well-formedness

During execution, new addresses will be added to the heap and, as required by (RTYP EXPCONFIG) and (RTYP STATCONFIG), to the typing environment if it is to remain well-formed with respect to that heap. We must ensure, then, that typing of expressions is preserved by such extension.

**Lemma 2.4** (Environment weakening)**.** *Typing of expressions and statements is preserved by the addition of new variable and address bindings to the typing environment:*

$$\left.\begin{matrix} \Gamma \vdash_{\mathbf{R}} e : t \\ \Gamma \vdash_{\mathbf{R}} s \end{matrix}\right\} \text{ and } x \notin dom(\Gamma) \text{ implies } \begin{cases} \Gamma[x \mapsto t'] \vdash_{\mathbf{R}} e : t \\ \Gamma[x \mapsto t'] \vdash_{\mathbf{R}} s \end{cases}$$

*Similarly for the addition of address bindings.*

*Proof.* By induction on the rules defining the typing relation. □

Recalling the rules of Figure 2.12, the well-formedness of heaps, relationship stores and local variable stores is predicated upon their constituent values being well-typed. As a result, the weakening property is available for those well-formedness relations too:

**Lemma 2.5.** *Well-formedness of objects, heaps, relationship stores, and local variable stores is preserved by environment weakening:*

$$\left.\begin{matrix} \Gamma \vdash_{\mathbf{R}} o \\ \Gamma \vdash_{\mathbf{R}} \sigma \\ \Gamma \vdash_{\mathbf{R}} \rho \\ \Gamma \vdash_{\mathbf{R}} \gamma \end{matrix}\right\} \text{ and } x \notin dom(\Gamma) \text{ implies } \begin{cases} \Gamma[x \mapsto t'] \vdash_{\mathbf{R}} o \\ \Gamma[x \mapsto t'] \vdash_{\mathbf{R}} \sigma \\ \Gamma[x \mapsto t'] \vdash_{\mathbf{R}} \rho \\ \Gamma[x \mapsto t'] \vdash_{\mathbf{R}} \gamma \end{cases}$$

*Similarly for the addition of address bindings to the environment.*

*Proof.* By inspection of the rules in Figure 2.12 and the application of Lemma 2.4 (Environment Weakening). □

During each execution step, objects may be updated or new objects may be allocated. Updates to the heap may be expressed by disjoint union:

$$\sigma[\iota \mapsto o] = \begin{cases} \sigma \uplus \{\iota \mapsto o\} & \iota \notin \mathsf{dom}(\sigma) \\ \sigma' \uplus \{\iota \mapsto o\} & \text{otherwise, where } \sigma = \sigma' \uplus \{\iota \mapsto o'\} \end{cases}$$

It is therefore useful to note that the indirection provided by typing addresses from an environment, $\Gamma$, allows us to isolate parts of the various stores that have changed in order to re-establish well-formedness conditions.

**Lemma 2.6.** *Well-formedness conditions are compositional. For disjoint heaps, where $\mathsf{dom}(\sigma) \cap \mathsf{dom}(\sigma') = \emptyset$:*

$$\begin{array}{l} \Gamma \vdash_{\mathbf{R}} \sigma \textit{ and } \Gamma \vdash_{\mathbf{R}} \sigma' \textit{ iff } \Gamma \vdash_{\mathbf{R}} \sigma \uplus \sigma' \\ \Gamma \vdash_{\mathbf{R}} \rho \textit{ and } \Gamma \vdash_{\mathbf{R}} \rho' \textit{ iff } \Gamma \vdash_{\mathbf{R}} \rho \uplus \rho' \\ \Gamma \vdash_{\mathbf{R}} \gamma \textit{ and } \Gamma \vdash_{\mathbf{R}} \gamma' \textit{ iff } \Gamma \vdash_{\mathbf{R}} \gamma \uplus \gamma' \end{array}$$

*Proof.* By inspection of the rules in Figure 2.12. □

To establish that the operations performed by the semantics upon the stores preserve their well formedness, it remains then to check that the objects with which the heap will be extended are well-formed:

**Lemma 2.7.** *The singleton heap portions containing new or updated objects are well-formed. For $\iota \notin \mathsf{dom}(\Gamma)$:*

$$\Gamma[\iota \mapsto c] \vdash_{\mathbf{R}} \{\iota \mapsto \mathsf{new}(c)\}$$
$$\Gamma \vdash_{\mathbf{R}} \iota_1 : \mathscr{R}(r)_{\mathsf{from}} \textit{ and } \Gamma \vdash_{\mathbf{R}} \iota_2 : \mathscr{R}(r)_{\mathsf{to}} \textit{ implies } \Gamma[\iota \mapsto r] \vdash_{\mathbf{R}} \{\iota \mapsto \mathsf{new}(r, \iota_1, \iota_2)\}$$

*And for $\iota \in \mathsf{dom}(\Gamma)$:*

$$\Gamma \vdash_{\mathbf{R}} \{\iota \mapsto o\} \textit{ and } o_{\mathsf{dyntyp}} = n \textit{ and } \Gamma \vdash_{\mathbf{R}} v : \mathscr{F}_n(f) \textit{ implies } \Gamma \vdash_{\mathbf{R}} \{\iota \mapsto o[f \mapsto v]\}$$

*Proof.* By application of (RTyp CObject), (RTyp RObject) and (RTyp Heap). □

Therefore, we can show that object allocation and object update result in well-formed stores. Similarly, the addition of new entries in the relationship store and the allocation of new local variables preserve well-formedness.

### 2.6.3 Preserving types

It then remains to show that the manipulations performed on terms by the semantics preserve typing. The fundamental operation in the semantics is the subtitution of sub-terms in contexts with other sub-terms:

**Lemma 2.8** (Soundness of context substitution)**.** *Substitution of a well-typed term in a context with another well-typed term preserves (or specializes) typing. Formally, if* $\Gamma \vdash_{\mathbf{R}} e_1 : t_1$ *and* $\Gamma \vdash_{\mathbf{R}} e_2 : t_2$ *and* $\vdash_{\mathbf{R}} t_2 \leq t_1$ *then:*

$$\Gamma \vdash_{\mathbf{R}} \mathscr{E}_{\mathrm{e}}[e_1] : t_3 \textit{ implies } \Gamma \vdash_{\mathbf{R}} \mathscr{E}_{\mathrm{e}}[e_2] : t_3$$
$$\Gamma \vdash_{\mathbf{R}} \mathscr{E}_{\mathrm{s}}[e_1] \textit{ implies } \Gamma \vdash_{\mathbf{R}} \mathscr{E}_{\mathrm{s}}[e_2]$$

*Proof.* By induction on the rules specifying $\Gamma \vdash_{\mathbf{R}} \mathscr{E}_{\mathrm{e}}[e_1] : t_1$ and $\Gamma \vdash_{\mathbf{R}} \mathscr{E}_{\mathrm{s}}[e_2]$. □

The lemma above demonstrates the soundness of (RRED CONTEXT), thus it remains to show that the small steps generated by the other reduction rules preserve types.

**Theorem 2.9** (Subject Reduction)**.**

$$\Gamma \vdash_{\mathbf{R}} \langle \sigma, \rho, \gamma, e \rangle : t \textit{ and } \langle \sigma, \rho, \gamma, e \rangle \rightsquigarrow_{\mathbf{R}} \langle \sigma', \rho', \gamma', e' \rangle \textit{ implies}$$
$$\textit{there exists } \Gamma' \supseteq \Gamma \textit{ such that } \Gamma' \vdash_{\mathbf{R}} \langle \sigma', \rho', \gamma', e' \rangle : t$$

$$\Gamma \vdash_{\mathbf{R}} \langle \sigma, \rho, \gamma, s \rangle \textit{ and } \langle \sigma, \rho, \gamma, s \rangle \rightsquigarrow_{\mathbf{R}} \langle \sigma', \rho', \gamma', s' \rangle$$
$$\textit{implies there exists } \Gamma' \supseteq \Gamma \textit{ such that } \Gamma' \vdash_{\mathbf{R}} \langle \sigma', \rho', \gamma', s' \rangle$$

*Proof.* By induction on the rules generating $\Gamma \vdash_{\mathbf{R}} e : t$ from (RTYP EXPCONFIG). In general, each case proceeds by examining two subcases:

i. the execution step arises from (RRED CONTEXT), in which case the result follows by induction and from Lemma 2.8; or

ii. the execution step arises from one of the other, axiomatic reduction rules, in which case well-formedness of the stores is established by Lemmas 2.6 and 2.7, and the preservation of type must be checked by inspection of the relevant rule.

In both cases, the environment weakening of Lemma 2.4 must be used to accomodate new addresses and variables from the inductive step in (i), or the composition of heaps in (ii). □

### 2.6.4 Progress

In order to drive the induction in the progress proof, it is important that sub-terms of well-typed terms are themselves well-typed:

**Lemma 2.10** (Context sub-expressions are typable)**.**

$$\left.\begin{array}{l}\Gamma \vdash_{\mathbf{R}} \mathscr{E}_{\mathrm{e}}[e] : t \\ \Gamma \vdash_{\mathbf{R}} \mathscr{E}_{\mathrm{s}}[e]\end{array}\right\} \textit{ implies } \exists t' : \Gamma \vdash_{\mathbf{R}} e : t'$$

(RAbstraction)

$$\frac{\forall (r, \iota_1, \iota_2) \in \mathsf{dom}(\rho) : \sigma(\rho(r, \iota_1, \iota_2)) = \langle\!\langle r, \iota_1, \iota_2 \| \ldots \rangle\!\rangle}{\sigma \vdash_{\mathbf{R}} \rho}$$

Figure 2.17: Relationship abstraction relation specifying consistency between relationship store $\rho$ and heap $\sigma$

*Proof.* By induction on the rules defining the $\Gamma \vdash_{\mathbf{R}} e : t$ and $\Gamma \vdash_{\mathbf{R}} s$ relations. □

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

**Theorem 2.11** (Progress)**.** *If a configuration is well-typed, it is either terminal (a value or an empty statement) or it may make an execution step to a new configuration, which may be an error configuration.*

$$\begin{aligned} \Gamma \vdash_{\mathbf{R}} \langle \sigma, \rho, \gamma, e \rangle : t \textit{ and } e \notin \mathsf{Value} \textit{ implies } & \langle \sigma, \rho, \gamma, e \rangle \rightsquigarrow_{\mathbf{R}} \langle \sigma', \rho', \gamma', e' \rangle \\ \textit{or } & \langle \sigma, \rho, \gamma, e \rangle \rightsquigarrow_{\mathbf{R}} \langle \sigma', \rho', \gamma', w \rangle \end{aligned}$$

$$\begin{aligned} \Gamma \vdash_{\mathbf{R}} \langle \sigma, \rho, \gamma, s \rangle \textit{ and } s \neq \epsilon \textit{ implies } & \langle \sigma, \rho, \gamma, s \rangle \rightsquigarrow_{\mathbf{R}} \langle \sigma', \rho', \gamma', s' \rangle \\ \textit{or } & \langle \sigma, \rho, \gamma, s \rangle \rightsquigarrow_{\mathbf{R}} \langle \sigma, \rho, \gamma, w \rangle \end{aligned}$$

Theorems 2.9 and 2.11 demonstrate that a well-formed program never gets stuck and always executes either to a terminal configuration or to an error configuration — well-typed programs do not go wrong.

## 2.7 Correctness

The type preservation and progress results guarantee, amongst other properties, that an expression of some type will evaluate to a value of that type, or will diverge, or will terminate in a well-defined error state — there will be no attempt to invoke methods upon a value that cannot provide them (other than `null`, for which we have an error state).

Whilst the well-formedness conditions of Figure 2.12 ensure that the objects involved in entries from relationship store $\rho$ have the correct type, we do not check that $\rho$-entries are consistent with the heap, $\sigma$, other than that types match and the objects exist. We have already discussed how the relationship represents an abstraction of the heap: it remains to show that the abstraction is faithful. To do so, we add a new property, called *relationship abstraction*, which is defined in Figure 2.17 and ensures that $\rho$ will give consistent answers when evaluating the results for relationship access operators. When a heap $\sigma$ is abstracted by a relationship store $\rho$, written $\sigma \vdash_{\mathbf{R}} \rho$, (RAbstraction) requires that the relationship

and related addresses in $\rho$ map to an object in the heap whose dynamic type, source and destination addresses match those used to access the heap object in the first place. As a result, the following conditions are enforced:

$$\rho(r, \iota_1, \iota_2).\texttt{from} \rightsquigarrow^*_{\mathbf{R}} \iota_1$$
$$\rho(r, \iota_1, \iota_2).\texttt{to} \rightsquigarrow^*_{\mathbf{R}} \iota_2$$

Referring to the relationship encodings of Section 1.2, this informally means that references from participating objects to the relationship object — not present in RelJ as they are replaced by $\rho$ — are consistent with the references in the opposite direction — the `to` and `from` fields of relationship instances.

Owing to the possibility of inactive relationship instances in the heap, there may be many relationship stores that are consistent with any given heap. Furthermore, we have the following property, just as for well-formedness conditions:

**Lemma 2.12.** *Relationship abstraction is compositional:*

$$\sigma \vdash_{\mathbf{R}} \rho \uplus \rho' \textit{ iff } \sigma \vdash_{\mathbf{R}} \rho \textit{ and } \sigma \vdash_{\mathbf{R}} \rho'$$

*Proof.* By trivial inspection of the rule (RABSTRACTION): the relation is defined pointwise on $\rho$. □

The store may be extended without affecting the validity of entries in the relationship store:

**Lemma 2.13.** *Relationship abstraction is preserved by store extension and by field update:*

$$\sigma \vdash_{\mathbf{R}} \rho \textit{ and } \iota \notin \mathsf{dom}(\sigma) \textit{ implies } \sigma[\iota \mapsto o] \vdash_{\mathbf{R}} \rho$$
$$\sigma \vdash_{\mathbf{R}} \rho \textit{ and } \sigma(\iota)_{\substack{\mathsf{dyntyp,}\\ \mathsf{from,to}}} = o_{\substack{\mathsf{dyntyp,}\\ \mathsf{from,to}}} \textit{ implies } \sigma[\iota \mapsto o] \vdash_{\mathbf{R}} \rho$$

*Proof.* By inspection of (RABSTRACTION). □

The removal of entries from $\rho$ must therefore preserve relationship abstraction. By inspection, we observe that addRel only creates entries in $\rho$ and objects in $\sigma$ that satisfy the conditions of (RABSTRACTION). It is then easy to see that this abstraction is preserved by the operational semantics:

**Theorem 2.14** (Abstraction preservation). *Heap and relationship store updates during program execution preserve relationship abstraction. For all transitions from some term, R:*

$$\langle \sigma, \rho, \gamma, R \rangle \rightsquigarrow_{\mathbf{R}} \begin{cases} \langle \sigma', \rho', \gamma', R' \rangle \\ \langle \sigma', \rho', \gamma', w \rangle \end{cases}$$

*it is the case that:*

$$\Gamma \vdash_{\mathbf{R}} \langle \sigma, \rho, \gamma, e \rangle : t \textit{ and } \sigma \vdash_{\mathbf{R}} \rho \textit{ and implies } \sigma' \vdash_{\mathbf{R}} \rho'$$

*Proof.* By induction on the rules generating the $\rightsquigarrow_{\mathbf{R}}$ relation. □

As a result, we may be assured that queries on the relationship store when evaluating $e.r$ and $e{:}r$ will provide sound *and* correct results. Put differently, this guarantees that the consistency of structures representing relationships are always preserved, unlike when programmers are allowed (or obliged) to alter those structures manually.

## 2.8 Conclusion

In this chapter we presented RelJ, which demonstrates how Java can be extended with the ability to declare relationships alongside classes. By offering a formal description of RelJ's type system and semantics, we are able to guarantee the safety of program execution in the presence of such extensions. In the formal semantics, we introduced the relationship store, which acts as an abstract representation of the complex reference structures otherwise necessary to encode relationships. In doing so, we are able to guarantee that the language always maintains the consistency of those structures.

In the following chapter, we outline some variants of RelJ that fill more of the design space outlined at the beginning of this chapter, including a different model of relationship inheritance. In Chapter 4, we introduce heap query as a means to replace the relationship store — in a sense, the abstraction is done 'on the fly' — and in Chapter 5 we show how RelJ's relationship features can be translated into query.

Chapter 3

# Extending RelJ

In this chapter we present several extensions or modifications to the relationship model upon which RelJ is based. First, we give an alternative model of inheritance more suited to relationships which, we argue, is more intuitive than the Java-style inheritance model from the previous chapter. The addition of multiplicity constraints is also explored, after which we summarize the modification of the relationship model to accommodate multi-relationships, bidirectional relationships and relationships of higher arities.

## 3.1 An alternative model of relationship inheritance

In the previous chapter, it was observed that RelJ's behaviour can be unintuitive when objects are related both through a relationship and its sub-relationships. This section presents an alternative semantics for relationship inheritance, which we refer to as *shared inheritance*.

### 3.1.1 The problem

Recall the relationship definitions from the previous chapter:

```
relationship Attends extends Relation (Student, Course) {
   int mark;
   ...
}
relationship ReluctantlyAttends extends Attends
                        (LazyStudent, HardCourse) {
   ...
}
```

Unexpected behaviour was observed in the following situation:

```
Attends.add(bob, rocketScience);
ReluctantlyAttends.add(bob, rocketScience);
...
set<Course> bobsCourses = bob.Attends;
```

According to the semantics of Figure 2.13, bobsCourses has just one entry for rocketScience despite having two entries for the (bob,rocketScience) pair in the relationship store — one for Attends and one for ReluctantlyAttends. Consider the following code, in which bob's marks are printed:

```
for (Attends a : bob:Attends) {
   System.println("Bob got " + a.mark + " for " + a.to.code);
}
```

Whilst the semantics we assigned to bob:Attends in the previous chapter prevents a mark for rocketScience from appearing twice, the output only reflects the mark from the Attends-instance. The ReluctantlyAttends-instance between bob and rocketScience is distinct from the Attends-instance, such that updates to the mark through ReluctantlyAttends will not alter the output from the above code — relationship inheritance is not reflected by relationship state.

### 3.1.2 Proposed solution

In the RelJ semantics as they stand, the fields inherited by a relationship are implemented in exactly the same way as those the relationship declares. Instead, we propose that inherited fields should be implemented by an instance of the relationship in which they were declared, so that this instance may be *shared* among instances of sub-relationships.

For example, in the above situation, there would be only one instance of the mark field — that in the Attends-instance relating bob and rocketScience. The programmer may still ask for the mark field of the ReluctantlyAttends instance, but the request is passed on to the Attends instance which holds the field. Similarly, any update of the mark field can be passed on, so that the reported mark is consistent regardless of which relationship instance is accessed.

Instance sharing means that we must ensure the presence of an Attends-instance for every ReluctantlyAttends-instance if a full set of (possibly inherited) fields is to be available: if a ReluctantlyAttends-instance were allowed to exist alone, then there would be no storage for its expected mark field.

By enforcing this invariant, there are no longer competing entries in the relationship store, and sub-relationships are always represented in accesses to super-relationships. For example, recall the triangular inheritance hierarchy from the previous chapter:

```
relationship R extends Relation (Object, Object) { ... }
relationship S extends R (Object, Object) { ... }
relationship T extends R (Object, Object) { ... }
```

When we make the following addition:

```
S.add(a, b);
```

it must be the case that an instance of R exists between a and b in order to implement the fields declared in R and inherited by S. If it is also the case that a

and b are related though T, then we no longer have to decide whether to return the S-instance, the T-instance or both when evaluating:

```
set<R> rinsts = a.R;
```

We may simply return the R-instance for a and b, which must exist and which is shared by the S- and T-instances relating a and b.

Whilst the idea of small objects cooperating to produce an instance of a single type bears some similarity to virtual multiple inheritance of C++ [18], it is not possible for C++'s sub-instances to be allocated separately: the implementation relies on a strict memory layout. The inheritance scheme given here also resembles Drossopoulou et al.'s Fickle [31], which allows objects to move — with some limitations — between classes with a common supertype just as we allow a relationship instance to take part in several sub-relationship instances simultaneously. Like C++, however, Fickle objects are not divided into smaller, separately-allocated parts. Albano et al.'s roles [3] also allow the run-time extension of objects with new state and behaviour. They rely on a delegation mechanism — more complicated than that described above — to ensure that the appropriate role methods are used to answer method invocations.

We now define $\mathsf{RelJ}^{\mathsf{s}}$, which implements this new model of relationship inheritance.

### 3.1.3 Modified semantics

The typing of expressions and statements is unchanged in $\mathsf{RelJ}^{\mathsf{s}}$: only the way in which relationships are instantiated and field accesses are carried out is to be altered. The goal of these new semantics is to preserve the following invariants:

**Invariant 1** (Single-instance). *There is only one instance of $r$ between any to addresses $\iota_1$ and $\iota_2$.*

This invariant is already enforced by RelJ as the relationship store, $\rho$, is a map.

**Invariant 2** (Super-relationship). *If $\iota_1$ and $\iota_2$ are related by an instance of $r$, then there is an instance of every super-relationship of $r$ also between $\iota_1$ and $\iota_2$.*

**Extended object model** Class instances remain unchanged in $\mathsf{RelJ}^{\mathsf{s}}$, but we extend relationship instances to accommodate the sharing of super-relationship implementations:

$$o = \langle\!\langle r, \iota_1, \iota_2, \iota \| f_1 : v_1, \ldots, f_i : v_i \rangle\!\rangle \qquad\qquad o_{\mathsf{super}} = \iota$$

The fourth member, $\iota$, is referred to as $o$'s *super-instance address*. That is, $\iota$ is the address of an instance of $r$'s super-relationship, $r'$, and services all requests for fields declared in $r'$ on behalf of $o$. This super-instance address may be `null` where the instance is of `Relation`, as it has no super-relationship. As shown

$$\frac{\Gamma \vdash_{\mathbf{R}} \iota_1 : \texttt{Object} \qquad \Gamma \vdash_{\mathbf{R}} \iota_2 : \texttt{Object}}{\Gamma \vdash_{\mathbf{R}} \langle\!\langle \texttt{Relation}, \iota_1, \iota_2, \texttt{null} \| \rangle\!\rangle} \text{ (RTYP ROBJECTI}^{\mathsf{s}}\text{)}$$

$$\frac{\begin{array}{l} \mathsf{dom}(\mathscr{F}_r) = \{f_1, \ldots, f_i\} \\ \forall j \in 1..i : \Gamma \vdash_{\mathbf{R}} v_j : \mathscr{F}_r(f_j) \\ \Gamma \vdash_{\mathbf{R}} \iota_1 : \mathscr{R}(r)_{\mathsf{from}} \qquad \Gamma \vdash_{\mathbf{R}} \iota_2 : \mathscr{R}(r)_{\mathsf{to}} \\ \Gamma(\iota) = \mathscr{R}(r)_{\mathsf{super}} \end{array}}{\Gamma \vdash_{\mathbf{R}} \langle\!\langle r, \iota_1, \iota_2, \iota \| f_1 : v_1, \ldots, f_i : v_i \rangle\!\rangle} \text{ (RTYP ROBJECT2}^{\mathsf{s}}\text{)}$$

Figure 3.1: Modified well-formedness relation for relationship instances

(RRED RELADD$^{\mathsf{s}}$) $\langle \sigma_1, \rho_1, \gamma, r\texttt{.add(}\iota_1\texttt{,}\iota_2\texttt{)} \rangle \rightsquigarrow_{\mathbf{R}} \langle \sigma_2, \rho_2, \gamma, \iota_3 \rangle$
where $(\sigma_2, \rho_2) = \mathsf{addRel}^{\mathsf{s}}(r, \iota_1, \iota_2, \sigma_1, \rho_1)$
$\iota_3 = \rho_2(r, \iota_1, \iota_2)$

(RRED RELREMI$^{\mathsf{s}}$) $\langle \sigma, \rho_1, \gamma, r\texttt{.rem(}\iota_1\texttt{,}\iota_2\texttt{)} \rangle \rightsquigarrow_{\mathbf{R}} \langle \sigma, \rho_2, \gamma, \rho_1(r, \iota_1, \iota_2) \rangle$
where $(r, \iota_1, \iota_2) \in \mathsf{dom}(\rho_1)$
$\rho_2 = \rho_1 \setminus \{(r', \iota_1, \iota_2) \mapsto \iota' \mid \vdash_{\mathbf{R}} r' \leq r\}$

Figure 3.2: Updated semantics for RelJ$^{\mathsf{s}}$'s relationship manipulation operators

above, our subscript notation is extended to provide shorthand access to this extra annotation.

**Object typing** Figure 3.1 updates the well-formedness relation for relationship instances, $\Gamma \vdash_{\mathbf{R}} o$. Two rules, (RTYP ROBJECTI$^{\mathsf{s}}$) and (RTYP ROBJECT2$^{\mathsf{s}}$) replace the original (RTYP ROBJECT) rule from the previous chapter. The former checks instances of `Relation`, for which we require only that the source and destination addresses are of reference type. (RTYP ROBJECT2$^{\mathsf{s}}$) performs the checks familiar from the original rule, but now specifies that only those fields directly declared in $r$ are present in $r$-instances. An additional condition ensures that the super-instance address has the type of $r$'s super-relationship, so that we may be assured that the super-instance will be able to satisfy requests for inherited fields and methods that will be passed on to it. Here, we require an equal type rather than a subtype in order to guarantee progress up the inheritance hierarchy when following super-instance pointers. Use of subtyping (via the typing relation) would admit various undesirable structures: for example, an object would be allowed to be its own super-instance.

The other rules from Figure 2.12 remain in place for RelJ$^{\mathsf{s}}$.

**Updated reduction relation** Modified rules for defining $\rightsquigarrow_{\mathbf{R}}$ are given in Figure 3.2 and replace their counterparts from the semantics given in Figure 2.13.

$$\mathsf{new^s}(r, \iota_1, \iota_2, \iota_{\text{sup}}) \mathrel{\widehat{=}} \langle\!\langle r, \iota_1, \iota_2, \iota_{\text{sup}} \| f_1 : \mathsf{initial}_P(\mathscr{F}_r(f_1)), \ldots, f_i : \mathsf{initial}(\mathscr{F}_r(f_i))^{\,f_i \in \mathsf{dom}(\mathscr{F}_r)} \rangle\!\rangle$$

$$\mathsf{addRel^s}(r, \iota_1, \iota_2, \sigma, \rho) \mathrel{\widehat{=}} \begin{cases} (\sigma, \rho) & \text{if } (r, \iota_1, \iota_2) \in \mathsf{dom}(\rho) \\ (\sigma'[\iota \mapsto o_{\text{new}}], \rho'[(r, \iota_1, \iota_2) \mapsto \iota]) & \text{otherwise, } \iota \notin \mathsf{dom}(\sigma) \end{cases}$$

where $\mathsf{addRel^s}$ is used recursively in the construction of $\sigma'$ and $\rho'$:

$$(\sigma', \rho') = \begin{cases} (\sigma, \rho) & \text{if } r = \texttt{Relation} \\ \mathsf{addRel^s}(r', \iota_1, \iota_2, \sigma, \rho) & \text{otherwise} \end{cases}$$

$$o_{\text{new}} = \begin{cases} \mathsf{new^s}(r, \iota_1, \iota_2, \texttt{null}) & \text{if } r = \texttt{Relation} \\ \mathsf{new^s}(r, \iota_1, \iota_2, \rho'(r', \iota_1, \iota_2)) & \text{otherwise} \end{cases}$$

$$r' \mathrel{\widehat{=}} \mathscr{R}(r)_{\mathsf{super}}$$

Figure 3.3: Updated auxilliary functions for $\mathsf{RelJ^s}$

(RRED RELADD$^{\mathsf{s}}$) has only been changed to reflect a new version of the $\mathsf{addRel}$ auxilliary function, $\mathsf{addRel^s}$, which is shown in Figure 3.3. Just as before, it takes a relationship, $r$, two addresses to be related through $r$, $\iota_1$ and $\iota_2$, and a heap and relationship store, $\sigma$ and $\rho$ respectively. If $\rho$ indicates that $\iota_1$ and $\iota_2$ are already related through $r$, then no action is taken. However, if the relationship does not already exist, then $\mathsf{addRel^s}$ first recursively ensures that $r$'s super-relationship, $r'$, exists between $\iota_1$ and $\iota_2$ in order to maintain the super-relationship invariant. This may in turn lead to the creation of new relationships all the way up $r$'s inheritance hierarchy. If $r$ is `Relation`, then this recursion does not occur as there are no more super-relationships. Only once $\iota_1$ and $\iota_2$ are related by all of $r$'s super-relationships does $\mathsf{addRel^s}$ add an $r$-instance, $o_{\text{new}}$, between $\iota_1$ and $\iota_2$ using a modified version of the $\mathsf{new}$ auxilliary function, $\mathsf{new^s}$, which takes a fourth argument for the new $r$-instance's super-instance address. For `Relation`, there is no super-relationship, so its super-instance address is `null`; otherwise, $o_{\text{new}}$'s super-instance address is read from $\rho$. Notice also that $\mathsf{new^s}$ constructs $r$-instances only with fields immediately declared in $r$, using $\mathscr{F}_r$ rather than the inheritance-respecting field map, $\mathscr{F}_r^+$.

Relationships are inactivated by (RRED RELREMI$^{\mathsf{s}}$): when inactivating the relation of $\iota_1$ to $\iota_2$ through relationship $r$, the relation of $\iota_1$ to $\iota_2$ through any sub-relationship of $r$ must also be removed from the relationship store. If sub-relationships were not removed in this way, the super-relationship invariant for those entries that survive may be violated.

Finally, there are modified versions of field access and update in Figure 3.4, which are defined in terms of a new operation that returns the instance contain-

$$\sigma \uparrow_\iota f \mathrel{\widehat{=}} \begin{cases} \iota & \text{if } f \in \mathsf{dom}(\sigma(\iota)) \\ \sigma \uparrow_{\iota'} f & \text{otherwise, where } \iota' = \sigma(\iota)_{\mathsf{super}} \end{cases}$$

(RRED FLD$^\mathsf{s}$) $\langle \sigma, \rho, \gamma, \iota \,.\, f \rangle \rightsquigarrow_\mathbf{R} \langle \sigma, \rho, \gamma, \sigma(\sigma \uparrow_\iota f)(f) \rangle$

(RRED FLDASS$^\mathsf{s}$) $\langle \sigma, \rho, \gamma, \iota \,.\, f \ = \ v \rangle \rightsquigarrow_\mathbf{R} \langle \sigma[\iota' \mapsto \sigma(\iota')[f \mapsto v]], \rho, \gamma, v \rangle$
where $\iota' = \sigma \uparrow_\iota f$

Figure 3.4: Updated semantics for field access and update in RelJ$^\mathsf{s}$

ing field $f$ on behalf of the instance at $\iota$ in heap $\sigma$:

$$\sigma \uparrow_\iota f = \iota'$$

The instance at $\iota'$ will — if the heap is well-formed — be an instance either of a class which inherited field $f$, or will be an instance of the relationship in which $f$ was originally declared. For class instances, $\sigma \uparrow_\iota f$ is equivalent simply to $\iota$.

(RRED FLD$^\mathsf{s}$) reads field $f$ on address $\iota$ by finding the instance containing $f$, which may be at $\iota$ itself, or one of its (transitive) super-instances, and then by looking up $f$'s value as usual. Similarly, (RRED FLDASS$^\mathsf{s}$) updates the instance containing $f$ with $f$'s new value.

No modification is required to the semantics of the relationship access operators, given by (RRED RELOBJ) and (RRED RELINST) in Figure 2.13. When accessing relationship $r$, they already return only instances of precisely $r$ rather than admitting instances of sub-relationships. The super-relationship invariant now ensures that the results are representative of all active relationships, as the presence of any sub-relationships of $r$ now implies the presence of a corresponding $r$-instance.

### 3.1.4 Soundness

The soundness proof proceeds much as before: upon relationship access, the RelJ$^\mathsf{s}$ semantics returns exactly those instances that would have been returned according to the original RelJ semantics. It must be shown that new$^\mathsf{s}$ returns well-formed relationship instances and that addRel$^\mathsf{s}$ preserves the well-formedness of heaps and relationship stores, but this is straightforward. Otherwise, it remains to show that:

**Lemma 3.1.** *All inherited fields are available for each object, possibly through a super-instance:*

$$\Gamma \vdash_\mathbf{R} \langle \sigma, \rho, \gamma, \iota \rangle : n \text{ implies for all } f \in \mathsf{dom}(\mathscr{F}^+_n) \text{ that } f \in \mathsf{dom}(\sigma \uparrow_\iota f)$$

$$\frac{\begin{array}{l}\forall (r,\iota_1,\iota_2) \in \mathsf{dom}(\rho): \\ \quad r = \texttt{Relation} \text{ implies } \sigma(\rho(r,\iota_1,\iota_2)) = \langle\!\langle r,\iota_1,\iota_2,\texttt{null} \| \ldots \rangle\!\rangle \\ \quad r \neq \texttt{Relation} \text{ implies } \sigma(\rho(r,\iota_1,\iota_2)) = \langle\!\langle r,\iota_1,\iota_2,\rho(\mathscr{R}(r)_{\mathsf{super}},\iota_1,\iota_2) \| \ldots \rangle\!\rangle\end{array}}{\sigma \vdash_{\mathbf{R}} \rho} \quad (\text{RABSTRACTION}^{\mathsf{s}})$$

Figure 3.5: Updated abstraction relation for ReIJ$^{\mathsf{s}}$

*The well-formedness condition for the instance returned by $\sigma \uparrow_\iota f$ ensures that each field's value is well-typed.*

*Proof.* That the configuration is well-typed implies that $\iota \in \mathsf{dom}(\sigma)$ and that its dynamic type is a sub-type of $n$. With (RTYP ROBJECT2$^{\mathsf{s}}$), the result follows by induction on the number of recursive steps required to evaluate $\sigma \uparrow_\iota f$. □

The full proof of soundness is given in the RelJ technical report [13].

### 3.1.5 Correctness

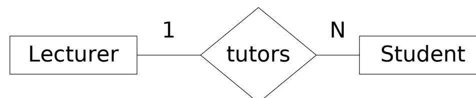
Previously the relationship abstraction relation, $\sigma \vdash_{\mathbf{R}} \rho$ implied that $\rho$ and $\sigma$ gave consistent results for each relationship instance's source and destination addresses. We give a new version of the rule in Figure 3.5, where the consistency requirements are also applied to super-instance pointers. As before, the rule ensures that any access of the relationship store, $\rho$, for relationship $r$ and addresses $\iota_1$ and $\iota_2$ always yields an $r$-instance whose participating addresses are $\iota_1$ and $\iota_2$. The new rule (RABSTRACTION$^{\mathsf{s}}$) further checks that the entry in $\rho$ for $r$'s super-relationship matches the super-instance address of the $r$-instance in $\sigma$, except for `Relation`-instances which have no super-instance. This ensures that following super-instance references does not result in a jump from an active instance to an inactive instance. The converse may not be true, of course, as a sub-relationship may have been inactivated, but its super-relationship may remain in $\rho$; for example, `bob` may become interested in taking `rocketScience` such that he no longer attends it reluctantly:

```
ReluctantlyAttends.add(bob, rocketScience);
ReluctantlyAttends.rem(bob, rocketScience);
```

The first line implicitly establishes an entry in $\rho$ for `Attends` between `bob` and `rocketScience`, as well as an entry for `ReluctantlyAttends`. The second line only removes the `ReluctantlyAttends` entry, leaving the entry for `Attends` active. Thus, asking for a field declared in `Attends` from the (inactivated) `ReluctantlyAttends`-instance involves following a super-instance pointer from an inactive instance to an active one.

## 3.2 Relationships with multiplicities

As we saw in Section 1.1.3, both UML class diagrams and ER diagrams permit associations/relationships to be annotated with multiplicities/cardinalities, which specify how many of one participant is related to each instance of the other participant. ER diagrams distinguish 'one' from 'many': for example each student has only one lecturer as their tutor, but each lecturer my act as tutor for several students:




UML *multiplicities* can be more expressive. 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 are available in UML, including 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 that can represent UML's more flexible multiplicities, and a more restricted, statically checked approach that only attempts to express the one/many granularity of ER diagrams' cardinalities.

### 3.2.1 Dynamic versus static solutions

The use of a run-time checks at every relationship addition would allow us to preserve multiplicities' upper-bounds by limiting the number of related objects. When, say, too many courses are added to the `Attends` relationship for a particular student, an exception could be raised:

```
relationship Attends (many Student, 0..2 Course)
   { int mark; }
...
Attends.add(alice, programming);
Attends.add(alice, semantics);
Attends.add(alice, types);       // Exception!
```

Complementary checks at relationship removal can ensure that lower-bounds are respected, whilst the addition of relationship constructors with implicit checks at the end of initialization can ensure that relationships start with an acceptable number of participants.

$$mc \in \mathsf{MultConstraint} ::= \mathtt{one} \mid \mathtt{many}$$
$$\mathsf{RelDecl} ::= \mathtt{relationship}\ r\ \mathtt{extends}\ r'\ (mc_1\, n_1,\ mc_2\, n_2)\ \{\ \mathsf{FieldDecl}^*, \mathsf{MethDecl}^*\ \}$$

Figure 3.6: Relationship declaration grammar for $\mathsf{RelJ}^{\mathbf{m}}$ and $\mathsf{RelJ}^{\mathbf{sm}}$

Our preference, however, is for a static approach to the expression of multiplicities. While less flexible, a compiler need not generate constraint-checking code for relationship additions, removals and constructions, and we provide more robust guarantees that the multiplicity constraints are satisfied.

### 3.2.2 Relationship declarations

$\mathsf{RelJ}^{\mathbf{m}}$ is the extension of RelJ with multiplicity declarations. Instead of the UML-style '0..$n$' declarations which must be checked at run-time, $\mathsf{RelJ}^{\mathbf{m}}$ only distinguishes between 'one' and 'many' multiplicities.

```
relationship Teaches extends Relation
                          (one Lecturer, many Course) {
   int studentRating;
}
```

According to this declaration, a lecturer may teach many courses, but each course has only one lecturer. We do not require that every `Course` has at least one `Lecturer` associated with it, so one is in fact equivalent to '0..1' in UML notation. If it were precisely '1', construction of each `Course` would require a `Lecturer` parameter in order to immediately satisfy the constraint.

The modified grammar for relationship declarations is given in Figure 3.6, and the definition of the relationship map, $\mathscr{R}$, is updated to reflect these declarations:

$$\mathscr{R}(\mathtt{Teaches}) = (\mathtt{Relation}, \mathtt{one\ Lecturer}, \mathtt{many\ Course}, \mathscr{F}_{\mathtt{Teaches}}, \mathscr{M}_{\mathtt{Teaches}})$$

We name the participants' multiplicity constraints fmul and tmul so that they may be obtained from $\mathscr{R}$ as follows:

$$\mathscr{R}(\mathtt{Teaches})_{\mathsf{fmul}} = \mathtt{one} \qquad \mathscr{R}(\mathtt{Teaches})_{\mathsf{tmul}} = \mathtt{many}$$

### 3.2.3 $\mathsf{RelJ}^{\mathrm{m}}$: Multiplicities with simple inheritance

No change to the typing rules or semantics is necessary other than to update the definition of addRel, which is given in Figure 3.7. Where one of the participating types is annotated 'one', $\mathsf{addRel}^{\mathbf{m}}$ removes any existing entries from the

$$\mathsf{addRel}^{\mathbf{m}}(r,\iota_1,\iota_2,\sigma,\rho) = \begin{cases} (\sigma,\rho) & \text{if } (r,\iota_1,\iota_2) \in \mathsf{dom}(\rho) \\ (\sigma[\iota \mapsto o_{\text{new}}], \rho[(r,\iota_1,\iota_2) \mapsto \iota]) & \text{otherwise, } \iota \notin \mathsf{dom}(\sigma) \end{cases}$$

$$\text{where } o_{\text{new}} = \mathsf{new}(r,\iota_1,\iota_2)$$
$$\rho' = \rho \setminus \{(r,\iota',\iota_2) \mapsto \iota'' \mid \mathscr{R}(r)_{\mathsf{fmul}} = \text{one and } \rho(r,\iota',\iota_2) = \iota''\}$$
$$\setminus \{(r,\iota_1,\iota') \mapsto \iota'' \mid \mathscr{R}(r)_{\mathsf{tmul}} = \text{one and } \rho(r,\iota_1,\iota') = \iota''\}$$

Figure 3.7: Definition of addRel for $\mathsf{RelJ}^{\mathbf{m}}$ that maintains multiplicity constraints

(RVT RELATIONSHIP$^{\mathbf{sm}}$)

$$\frac{\begin{array}{ll} r \neq \mathtt{Relation} \quad r' = \mathscr{R}(r)_{\mathsf{super}} & r, r' \in \mathsf{dom}(\mathscr{R}) \\ \forall f \in \mathsf{dom}(\mathscr{F}_r) : r \vdash_{\mathbf{R}} f & \forall m \in \mathsf{dom}(\mathscr{M}_r) : r \vdash_{\mathbf{R}} m \\ \vdash_{\mathbf{R}} \mathscr{R}(r)_{\mathsf{from}} \leq \mathscr{R}(r')_{\mathsf{from}} & \vdash_{\mathbf{R}} \mathscr{R}(r)_{\mathsf{to}} \leq \mathscr{R}(r')_{\mathsf{to}} \\ \mathscr{R}(r')_{\mathsf{fmul}} = \text{one implies } \mathscr{R}(r)_{\mathsf{fmul}} = \text{one} & \mathscr{R}(r')_{\mathsf{tmul}} = \text{one implies } \mathscr{R}(r)_{\mathsf{tmul}} = \text{one} \end{array}}{\vdash_{\mathbf{R}} r}$$

Figure 3.8: Updated rule for the specification of well-declared relationships in $\mathsf{RelJ}^{\mathbf{sm}}$, with new conditions highlighted

relationship store that will conflict with the new entry. If the source type is annotated one, then all relationships through $r$ to the new destination address are deleted; if the destination type is annotated one, then entries linking the new source address to any other address through $r$ are removed.

### 3.2.4 $\mathsf{RelJ}^{\mathbf{sm}}$: Multiplicities with shared inheritance

The situation is more complicated when introducing multiplicity constraints to $\mathsf{RelJ}^{\mathbf{s}}$, the version of RelJ with the shared inheritance model of Section 3.1.

**Type system** Firstly, the super-relationship invariant requires that multiplicity constraints can only become *more* restrictive down the inheritance hierarchy, otherwise sub-relationships may exist where super-relationships may not and the invariant becomes unsatisfiable. Therefore, a relationship with a one annotation on one of its participants may only be extended by a new relationship if it preserves that annotation. This requirement is formalized in Figure 3.8 by a new version of (RVT RELATIONSHIP$^{\mathbf{sm}}$), which specifies the conditions under which a relationship is well-declared — the new conditions are highlighted.

**Semantics** Just as for $\mathsf{RelJ}^{\mathbf{m}}$, the only change required to the semantics is to the definition of addRel. The new version, $\mathsf{addRel}^{\mathbf{sm}}$, is defined in Figure 3.9 as a combination of its counterparts in $\mathsf{RelJ}^{\mathbf{s}}$ and $\mathsf{RelJ}^{\mathbf{m}}$. Like the former, the function recurses up the inheritance tree to ensure super-relationships are in place before

$$\mathsf{addRel^{sm}}(r,\iota_1,\iota_2,\sigma,\rho) \mathrel{\widehat{=}} \begin{cases} (\sigma,\rho) & \text{if } (r,\iota_1,\iota_2) \in \mathsf{dom}(\rho) \\ (\sigma'[\iota \mapsto o_{\text{new}}], \rho''[(r,\iota_1,\iota_2) \mapsto \iota]) & \text{otherwise, } \iota \notin \mathsf{dom}(\sigma) \end{cases}$$

where:

$$(\rho',\sigma') \mathrel{\widehat{=}} \begin{cases} (\sigma,\rho) & \text{if } r = \texttt{Relation} \\ \mathsf{addRel^{sm}}(r',\iota_1,\iota_2,\sigma,\rho) & \text{otherwise} \end{cases}$$

$$o_{\text{new}} \mathrel{\widehat{=}} \begin{cases} \mathsf{new^s}(r,\iota_1,\iota_2,\texttt{null}) & \text{if } r = \texttt{Relation} \\ \mathsf{new^s}(r,\iota_1,\iota_2,\rho'(r',\iota_1,\iota_2)) & \text{otherwise} \end{cases}$$

$$r' \mathrel{\widehat{=}} \mathscr{R}(r)_{\mathsf{super}}$$

$$\begin{aligned} \rho'' \mathrel{\widehat{=}} \rho' \backslash \{(r'',\iota',\iota_2) \mapsto \iota'' \mid \mathscr{R}(r)_{\mathsf{fmul}} = \text{one and} \\ \vdash_{\mathbf{R}} r'' \leq r \text{ and } \rho'(r'',\iota',\iota_2) = \iota''\} \\ \backslash \{(r'',\iota_1,\iota') \mapsto \iota'' \mid \mathscr{R}(r)_{\mathsf{tmul}} = \text{one and} \\ \vdash_{\mathbf{R}} r'' \leq r \text{ and } \rho'(r'',\iota_1,\iota') = \iota''\} \end{aligned}$$

Figure 3.9: New definition of addRel for $\mathsf{RelJ^{sm}}$ which implements multiplicities in the presence of the revised inheritance model for relationships

the new relationship instance is created, whilst relationship entries are cleared away if they conflict with the new instance created at each recursive step. However, where $\mathsf{addRel^{sm}}$ must clear a relationship entry to protect the multiplicity invariant, it must also clear entries for sub-relationships in order to protect the super-relationship invariant just as for 'standard' relationship removal in $\mathsf{RelJ^{s}}$.

The combination of recusion up the inheritance tree with the clearance of relationships down the inheritance tree can yield surprising — if potentially intuitive — results. Consider the following relationship definitions, where a course can only be taught by a single lecturer, and where lecturers enjoy teaching hard courses, but may teach them slowly:

```
relationship Teaches extends Relation
                  (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();
```

Suppose that `charlie` `ExcitedlyTeaches` `rocketScience`, then by the super-relationship invariant, `charlie` also `Teaches` `rocketScience`:

```
ExcitedlyTeaches.add(charlie, rocketScience);
  // implies: Teaches.add(charlie, rocketScience);
```

Now suppose that deirdre is to teach rocketScience slowly:

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

Again by the super-relationship invariant, deirdre must also be in the Teaches relationship with rocketScience via Teaches. However, in order to respect the multiplicity constraint on Teaches, charlie and deirdre cannot *both* teach rocketScience. In that case, $\mathsf{addRel}^{\mathbf{sm}}$ removes the (charlie, rocketScience) pair from Teaches. Furthermore, the super-relationship invariant does not allow charlie to be in ExcitedlyTeaches with rocketScience once he has been removed from Teaches — therefore, he is also removed from ExcitedlyTeaches.

Whilst this behaviour appears rather convoluted, we argue that — at least for this example — this behaviour is what the programmer would expect in the absence of exceptions. Further experience will undoubtedly be required before any such design can be fixed; indeed, any implementation may need to remain flexible so that the programmer can adapt the model according to requirements.

**Soundness and correctness** The proof of soundness proceeds mostly as before; it must of course be shown that $\mathsf{addRel}^{\mathbf{sm}}$ preserves the well-formedness of the heap and relationship stores, but this is clear when viewed as a combination of relationship deletion (compare with (RRED RELREMI) in Figure 3.2) and addition as per $\mathsf{addRel}^{\mathbf{s}}$ in Figure 3.3: $\mathsf{RelJ}^{\mathbf{sm}}$ can be easily compiled to $\mathsf{RelJ}^{\mathbf{s}}$. That the multiplicity invariants are maintained is not required for soundness, but the following property is easy to show:

**Lemma 3.2** (Preservation of multiplicity invariant). *Take $\phi, \psi$ to be the following properties, which express that a relationship store, $\rho$, respects the multiplicity invariants:*

$$\phi(\rho) \mathrel{\widehat{=}} \forall (r, \iota_1, \iota_2) \in \rho : \mathscr{R}(r)_{fmul} = one \text{ implies } \forall (r, \iota', \iota_2) \in dom(\rho) : \iota' = \iota_1$$
$$\psi(\rho) \mathrel{\widehat{=}} \forall (r, \iota_1, \iota_2) \in \rho : \mathscr{R}(r)_{tmul} = one \text{ implies } \forall (r, \iota_1, \iota') \in dom(\rho) : \iota' = \iota_2$$

*Then, wherever $\langle \sigma, \rho, \gamma, R \rangle \rightsquigarrow_{\mathbf{R}} \langle \sigma', \rho', \gamma', R' \rangle$ for well-typed configurations, then:*

$$\phi(\rho) \text{ implies } \phi(\rho') \qquad\qquad \psi(\rho) \text{ implies } \psi(\rho')$$

*Proof.* It suffices to show that these constraints are preserved by the relationship deletions in the definition of $\mathsf{addRel}^{\mathbf{sm}}$. □

The definitions of relationship abstraction are unchanged with the addition of multiplicity constraints, as is the related proof.

## 3.3 Other extensions

We now give informal summaries of other extensions to RelJ, and demonstrate them by example.

### 3.3.1 Multi-relationships

Multi-relationships allow pairs of objects to be related more than once through each relationship. For example, a student may be related to an examination more than once, where the examination has had to be re-taken; a mark will exist for each re-take. RelJ only permits one instance of each relationship between any pair of objects, but may be easily modified to permit multi-relationships.

First, the relationship store, $\rho$, must be modified so that more than one $r$-instance may exist between each pair of participating objects:

$$\rho ::= \{(r_1, \iota_1, \iota'_1) \mapsto \{\iota_{1,1}, \iota_{1,2}, \ldots \iota_{1,i\geq 0}\}, \ldots, (r_j, \iota_j, \iota'_j) \mapsto \{\iota_{j,1}, \ldots \iota_{j\geq 0, i\geq 0}\}\}$$

This change in the relationship store may be reflected for most operations in the obvious way. In the original RelJ semantics, `alice:Attends` evaluates to:

$$\{\iota' \mid \exists \iota'' : \rho(\texttt{Attends}, \iota_{\texttt{alice}}, \iota'') = \iota'\}$$

However, in the presence of a multi-relationship model the result becomes:

$$\bigcup \{\rho(\texttt{Attends}, \iota_{\texttt{alice}}, \iota') \mid (\texttt{Attends}, \iota_{\texttt{alice}}, \iota') \in \mathsf{dom}(\rho)\}$$

where $\iota_{\texttt{alice}}$ is the address of the `Student`-instance stored in the variable `alice`.

Finally, notice that addRel need no longer be idempotent; a new entry in $\rho$ may be made for every relationship addition, rather than for those where the requested relationship does not already exist. The modification required is slight, so we do not give a full redefinition of addRel.

The considerations above apply to both RelJ and RelJ$^{\mathsf{s}}$. However, whilst the impact on RelJ, with its simple inheritance model, is slight, the existence of more than one relationship instance between any pair of participating addresses makes it difficult to select an appropriate super-instance when using the shared inheritance model. For example, suppose that `alice` is in the `Attends` relationship with `semantics` twice: if `alice` is added to the `ReluctantlyAttends` relationship with `semantics`, then it is unclear which of the two existing `Attends`-instances should receive forwarded field requests on behalf of the new instance of `ReluctantlyAttends`. This ambiguity must either be resolved with new operations for 'extending' specific relationship instances, or by specifying that new relationship instances always imply a new line of ancestor-instances, eliminating sharing of super-relationship fields altogether.

### 3.3.2 Bidirectional relationships

Relationships in RelJ may only be accessed with a reference to the source object: the fact that `semantics` is in the set returned by `alice.Attends` may not be determined with a reference to `semantics` alone.

Whilst the semantic stuctures may remain unchanged, the addition of bidirectional relationships to RelJ requires a different syntax for relationship access which distinguishes the source and destination (or forwards and backward traversal) of the relationship. As we shall see, the problem of specifying which relationship participants we have, and which we wish to access, is a problem common to $n$-ary as well as bidirectional relationships. Therefore, we treat bidirectional binary relationships as a special case, and move directly to the consideration of $n$-ary relationships.

### 3.3.3 $n$-ary relationships

RelJ is restricted to binary relationships. Here we sketch a version with $n$-ary relationships, which can be derived relatively easily from RelJ's model of binary relationships but which still pose some interesting design problems.

So that participants of a relationship may be named in syntax, relationship declarations must give names to the participating types, denoting their rôle in the relationship.[1] For example, the binary `Attends` relationship's definition becomes:

```
relationship Attends extends Relation
       (Student student, Course course)  {
  ...
}
```

Of course, in this case, the rôles could have been determined by type, but this is certainly not always the case, particularly when the types are comparable by the subtype relation. For example, the following `Assesses` relationship specifies which tutors are responsible for assessing a student:

```
relationship Assesses extends Relation
        ( Student student,
          Tutor   supervisor,
          Tutor   examiner )
{ ... }
```

Two `Tutors` are involved, one as the `supervisor`, and one as the `examiner`.

The syntax for manipulating relationships can be extended in the obvious way. For example:

[1] The term 'rôle' is overloaded in the database and object-oriented language literature [9, 52]. Whilst our meaning overlaps with that in other works, our rôles are just labels for participants in a relationship, and we do not imbue objects in those rôles with any special properties.

```
Tutor lucy = new Tutor();    Tutor matthew = new Tutor();
...
Assesses.add( student = bob,
              supervisor = lucy,
              examiner = matthew );
```

Whilst the relationship store of RelJ is a map from relationship names to pairs of addresses, support for $n$-ary addresses requires that it map relationship names to a further map from rôle names to addresses. Thus, after evaluation of the `Attends.add` line above, the following might be true:

$$\rho(\texttt{Assesses}) = \{\texttt{student} \mapsto \iota_{\texttt{bob}}, \texttt{supervisor} \mapsto \iota_{\texttt{lucy}}, \texttt{examiner} \mapsto \iota_{\texttt{matthew}}\}$$

where $\iota_x$ is the store location stored in variable $x$. The replacement of pairs in $\rho$ for these maps must be echoed in relationship instances, where a rôle-participant map will replace the pair of addresses used to provide values for `from` and `to` in RelJ. Thus, an instance's participants begin to look increasingly like fields, but remain immutable during the lifetime of the instance. Just as for `from` and `to`, we make the participants' rôles available as pseudo-fields:

```
Assesses a = Assesses.add( ... );
Tutor t = a.examiner;
```

There is no longer an assumption of directionality for $n$-ary relationships, so we require a more expressive syntax for relationship access. Inspired by XPath [72], we suggest a projection notation. For example, if we wish to obtain the set of `Assesses`-instances that apply to `alice` as a student, we may issue the following:

```
Assesses[student == alice]
```

Similarly, the first step towards determining all those `Students` supervised by `lucy` and examined by `matthew` is to find all such `Attends`-instances:

```
Attends[supervisor == lucy && examiner == matthew]
```

By adding the ability to map field requests over a set of objects to obtain the corresponding set of results, as found in C$\omega$ [11] for example, we may provide a similar shortcut for obtaining participating instances without explicitly iterating over the set returned by relationship access:

```
set<Tutor> examiners = Attends[student == alice].examiner;
```

We stop short of suggesting a full query language for relationships here; query is explored in much greater detail in the following chapters.

## 3.4 Conclusion

In this chapter, we specified *shared inheritance* which provides a model of inheritance more suited to relationships than that used for classes. A scheme for

the support of multiplicity constraints was also sketched, and its interactions with other language features discussed. Finally, some adjustments to the model were presented which may be of particular relevance to software and database modelling languages: multi-relationships, bidirectional relationships and relationships of arbitrary arity.

This series of extensions demonstrates that RelJ's model of relationships is reasonably flexible: new relationship features may be obtained without extensive changes to the formalization. This flexibility is important so that experience in software development can rapidly inform the choice of features made available in languages with relationships.

CHAPTER 4

# Q$\varsigma$: An object calculus with heap query

We now turn our attention to object-based languages, and to heap query. We have already seen how access to RelJ's relationships relies on a very limited query of the relationship store, which is in turn an abstraction of the structures in the heap that would otherwise be required to encode relationships. However, a more flexible approach is required in an object-based world where objects are not so rigidly classified. We therefore move towards a more powerful query mechanism as a means to recover relationships from the heap — in the presence of query, a single reference may represent a relationship without the complex encodings from Section 1.2.

Whilst object-based calculi have been used by researchers to explore the fundamentals of object-oriented programming — perhaps most famously by Abadi and Cardelli [1] — object-based languages are popular in practice, particularly for scripting. For example, most web browsers will execute Javascript [33], which allows the construction of arbitrary objects without the constraints imposed by a class-based type system. Development time is reduced but, in the absence of a type system, the possibility of run-time error is increased.

In this chapter, we give a strongly-typed object calculus based on the $\varsigma$-calculi of Abadi and Cardelli [1]. We equip the calculus with heap query, which gives rise to a number of interesting typing issues, and demonstrate soundness.

## 4.1 Core language

First, we describe the core object calculus on which we build our extensions. Our object calculus is based on the calculus presented by Gordon, Hankin and Lassen [37], itself based on the **imp**$\varsigma$ calculus of Abadi and Cardelli [1]. To this basis, we add recursive types in the spirit of $\mathbf{FOb}_{1<:\mu}$ [1, Chapter 9]. A similar starting point was used in work by Clarke to explore the fundamentals of Ownership Types [23].

### 4.1.1 Terms

The core calculus, the grammar for which is given in Figure 4.1, contains an object-based portion and, for convenience, let-binding, $\lambda$-abstraction and application, which may be encoded using only the object-based terms.[1] We assume familiarity with this latter set of features, and only introduce the object-based portion of the calculus in detail. Types for the calculus, denoted $A$ and $B$, are discussed later.

An object is constructed by giving definitions to each of its methods. A method definition has the following components:

$$m ::= \varsigma(x : A)a$$

The body of the method — the term to be executed upon method invocation — is denoted $a$. The *self*-binder, $\varsigma$, specifes the variable, $x$, to be bound in the body with type $A$ to the identity of the object upon which the method is invoked.

Objects are finite, unordered collections of labelled methods, where each object has at most one definition for each label, and where at least one label is defined.[2]

$$[l_1 = \varsigma(x_1 : B_1)b_1, l_2 = \varsigma(x_2 : B_2)b_2, \ldots, l_n = \varsigma(x_n : B_n)b_n]$$
$$[l_i = \varsigma(x_i : B_i)b_i \,^{i \in 1..n}]$$

$l$ ranges over the countably infinite set of label names. For brevity, we index the parts of an object definition by some interval in the naturals — the definitions above are equivalent.

When an object is constructed, it is assigned a unique identity, which we will later use as the object's location in a program store. We use $\iota$ to range over the countable, infinite set of these identities. This set of identities is also included in the set of values, ranged over by $v$, which is, informally, the set of terms which do not execute any further and thus form the result of some computation. This set also includes $\lambda$-abstractions. Where the method body for a label $l$ is a value $v$ that does not include the self-parameter (in a free position) then we write $l = v$ instead of $l = \varsigma(x : A)v$.[3] Thus, whilst fields and methods are disjoint in the class-based world, they are only distinguished by notation in this calculus. Indeed, it is entirely possible that an update of label $l$ will give it a method body which is a non-value term.

An object's methods may be invoked using the '.' operator. For example, the

[1] In the encoding of $\lambda$-abstraction/application, provision of subtyping for function types requires variance annotations for soundness, which are not included in the object-based portion. [1, Section 8.7] We include them first-class, however, so we are able to provide sound subtyping for function types.

[2] Unlike Gordon, Hankin and Lassen [37], we *do* identify objects up to reordering of labels.

[3] As discussed later, self types must match for all labels in an object. Therefore, only one label need have a typed self-parameter for its type to be determined.

following constructs a *button* object, binds it to the *button* variable and 'clicks' the button by selecting its *click* label. We assume an appropriate type *Button* for the button object:

$$\mathsf{let}\ button : Button = [click = \varsigma(x : Button) \ldots \text{click code} \ldots]\ \mathsf{in}\ button.click$$

For clarity in examples, such as the above, we will routinely use lower-case italics for variables and labels, instead of relying on $x$ and $l$. Similarly for types, to which we give an upper-case initial.

If we wish to re-implement the *click* method, we can update its definition. In the following example, we construct a button with methods *click*, $l_1$ and $l_2$. The button reacts to clicks by invoking its own $l_1$ method. During the assignment to $button_2$, however, we update the button so that it receives a new implementation of its *click* method that invokes $l_2$ instead:

$$\begin{aligned}\mathsf{let}\ button_1 : Button = [&click = \varsigma(x : Button)x.l_1,\\ &l_1 = \varsigma(x : Button)a_1,\\ &l_2 = \varsigma(x : Button)a_2]\ \mathsf{in}\end{aligned}$$
$$\mathsf{let}\ button_2 : Button = button_1 \Leftarrow \varsigma(x : Button)x.l_2\ \mathsf{in}\ button_2.click$$

Method update may neither add nor remove methods. Update modifies the target object directly; no copying is involved. Thus, $button_1$ and $button_2$ both contain the address, $\iota$, of the single button — the invocation of $button_2.click$ could just as easily have been written $button_1.click$.

To make copies of an existing object, the result of a term may be cloned. For example, we may modify the final let clause from the example above:

$$\mathsf{let}\ button_2 : Button = \mathsf{clone}(button_1) \Leftarrow \varsigma(x : Button)x.l_2\ \mathsf{in}\ button_2.click$$

In this example, $button_2$'s *click* method is implemented using $l_2$, whilst $button_1$'s continues to use $l_1$.

### 4.1.2 Types

The intention of the type system is to prevent a label being invoked on an object which has no method associated with that label. Therefore, the intention of *object types* is to specify which labels are available for invocation, and the type of the term that will be returned upon such an invocation. We allow *width subtyping*, so that an object's type need not mention all of the labels defined within it. Therefore, labels may be hidden from the outside world, for example to protect an invariant. Width subtyping also allows objects to be extended with additional methods during software maintenance, but unmodified code may continue to view the extended object through its unextended type as long as all of the required methods are still provided. Where $A$ is a subtype of $B$, we write $A <: B$.

In general, an object type is a finite, unordered collection of mappings from

| | |
|---|---|
| $x \in \mathsf{var}$ | variable names |
| $l \in \mathsf{label}$ | label names |
| $\iota \in \mathsf{loc}$ | store locations |
| $A, B \in \mathsf{type}$ | types, see Figure 4.2 |
| $m \in \mathsf{meth} ::= \varsigma(x : A)b$ | method |
| $o \in \mathsf{obj} ::= [l_i = m_{i \in 1..n}]$ | object ($l_i$ distinct) |
| $a, b \in \mathsf{term} ::= \iota$ | location |
| $\mid x$ | variable |
| $\mid o$ | object |
| $\mid \lambda(x : A)a \mid a(b)$ | $\lambda$-abstraction/application |
| $\mid \mathsf{let}\ x : A = a\ \mathsf{in}\ b$ | let |
| $\mid a.l$ | selection |
| $\mid a.l \Leftarrow m$ | update |
| $\mid \mathsf{clone}(a)$ | clone |
| $\mid \mathsf{fold}(A, a)$ | type folding |
| $\mid \mathsf{unfold}(a)$ | type unfolding |

Figure 4.1: Grammar describing terms for the core calculus underlying **Qς**

label names to types. Object types must have at least one label defined. We have a distinguished, largest type, written $\mathsf{Top}$, which acts as the object type with no labels — that is, no methods may be invoked on a term of type $\mathsf{Top}$.

In our *button* example above, some appropriate types (in increasing subtype order) for the object assigned to $button_1$ would be:

$$[click : Click, l_1 : Click, l_2 : Click] <: [l_1 : Click] <: \mathsf{Top}$$

Of course, only the first type allows the update and invocation of methods as in the *button* example.

Many interesting examples require some form of recursion in the type system [1, 60]. For example, suppose we extend our button with a method, *id*, that returns the button's identity:

$$\begin{aligned} \mathsf{let}\ button : \mathsf{ButtonA} = [&click = \ldots, \\ &id = \varsigma(self : \mathsf{ButtonB})self]\ \mathsf{in}\ button.id.id \end{aligned}$$

It seems desirable that $\mathsf{ButtonA}$ and $\mathsf{ButtonB}$ — both set in sans-serif as we are discussing specific types, and no longer meta-variables — should all be the same type. We quickly run into trouble, however: $\mathsf{ButtonA}$ must be at least $[id : \ldots]$ to allow the first invocation of *id*. If the second call is to be allowed, the $\mathsf{ButtonA}$

$$
\begin{array}{rll}
X, Y \in \mathsf{tyvar} & & \text{type variables} \\
O \in \mathsf{otype} ::= & [l_i : A_{i \in 1..n}] & \text{object type } (l_i \text{ distinct}) \\
A, B \in \mathsf{type} ::= & X & \\
& \mid O & \text{object type} \\
& \mid \mathsf{Top} & \text{greatest type} \\
& \mid A \to B & \text{function type} \\
& \mid \mu(X).A & \text{recursive type}
\end{array}
$$

Figure 4.2: Grammar describing types for the core calculus underlying **Qς**

must be at least $[id : [id : \ldots]]$. This situation is clearly unsatisfactory, so we introduce recursion. Recursive types are written:

$$\mu(X).A$$

where $X$ ranges over a countably infinite set of type variables, and may occur in type $A$. This type may be *unfolded* by substituting $\mu(X).A$ for every (free) occurrence of $X$ in $A$. This substitution is written:

$$A\{\!\{\mu(X).A/X\}\!\}$$

By introducing fold and unfold into our term language to witness the isomorphism between a recursive type's folded and unfolded forms, we may write our example:

$$
\begin{array}{l}
\mathsf{let}\ \mathit{button} : \mathsf{ButtonA} = [\mathit{click} = \ldots, \\
\qquad\qquad\qquad\qquad \mathit{id} = \varsigma(\mathit{self} : \mathsf{ButtonA})\mathsf{fold}(\mathsf{ButtonB}, \mathit{self})] \\
\mathsf{in}\ \mathsf{unfold}(\mathit{button}.\mathit{id}).\mathit{id}
\end{array}
$$

where our Button types are, for some appropriate *click* return type, *Click*:

$$
\begin{array}{lcl}
\mathsf{ButtonA} & \widehat{=} & [\mathit{click} : \mathit{Click}, \mathit{id} : \mathsf{ButtonB}] \\
\mathsf{ButtonB} & \widehat{=} & \mu(X).[\mathit{click} : \mathit{Click}, \mathit{id} : X]
\end{array}
$$

To make *id* available for its second invocation, the result of the first *id* invocation must be unfolded to take its type from the folded type, ButtonB, to the unfolded type, ButtonA. The use of fold in the object definition, on the other hand, marks the transition of the type in the other direction, from ButtonA to ButtonB.

We summarize the grammar for the types so far described in Figure 4.2.

### 4.1.3 Adding lists

As for RelJ, it is reasonable to expect that more than one object may satisfy any given query. We therefore provide lists in which the results of such queries may be stored.

We therefore extend the core calculus with lists. Lists are built in the usual way using constructors nil, for the empty list, and cons. Lists are given the type list of $A$, where $A$ is a type valid for the list's members:

$$\mathsf{cons}(\iota_1, \mathsf{cons}(\iota_2, \mathsf{nil})) : \mathsf{list\ of\ Top}$$

Lists are typed covariantly, as they are passed by value. For the consumption of lists, the match operator is provided:

$$\mathsf{match}\ a : \mathsf{list\ of}\ A\ \mathsf{with\ cons}(x, y) \Rightarrow b \mid \mathsf{nil} \Rightarrow b'$$

This acts as both destructor and empty-test: it executes the term $b$ with $x$ and $y$ bound to the head and tail of the input list when it is non-empty; it executes $b'$ when the list is nil.

With these constructs, we are able to iterate over lists. In the following example, we use the *button* definition from the previous section to write a method *clickAll* for a window object. This method fetches a list of buttons from the window, clicks each in turn and then minimizes the window:

$$\begin{array}{l}
\mathsf{let}\ \mathit{window} : \mathsf{Window} = \\
\quad [\mathit{getButtons} = \varsigma(w : \mathsf{Window}) \ldots \text{supplies list of } \mathsf{ButtonA} \text{ objects} \ldots, \\
\quad\ \ \mathit{minimize} = \varsigma(w : \mathsf{Window}) \ldots \text{minimizes the window} \ldots, \\
\quad\ \ \mathit{clickAll} = \varsigma(w : \mathsf{Window}) w.\mathit{clickList}(w.\mathit{getButtons}), \\
\quad\ \ \mathit{clickList} = \varsigma(w : \mathsf{Window}) \\
\qquad\qquad \lambda(\mathit{bs} : \mathsf{list\ of\ ButtonA}) \mathsf{match}\ \mathit{bs} : \mathsf{list\ of\ ButtonA\ with} \\
\qquad\qquad\quad \mathsf{cons}(b, \mathit{bs}') \Rightarrow \mathsf{let}\ x : \mathsf{Top} = b.\mathit{click}\ \mathsf{in}\ w.\mathit{clickList}(\mathit{bs}') \mid \\
\qquad\qquad\quad \mathsf{nil} \Rightarrow w.\mathit{minimize}]
\end{array}$$

where:

$$\begin{array}{l}
\mathsf{Window} \mathrel{\widehat{=}} [\mathit{getButtons} : \mathsf{list\ of\ ButtonA}, \mathit{minimize} : \mathsf{Top}, \\
\qquad\qquad\qquad\qquad \mathit{clickAll} : \mathsf{Top}, \mathit{clickList} : \mathsf{list\ of\ ButtonA} \to \mathsf{Top}]
\end{array}$$

The 'self' parameter provides the recursion necessary to iterate over the list provided by *getButtons*. Notice also that the *clickList* method returns a $\lambda$-abstraction so that it may be considered to be a method that takes an argument, as at the recursive call on the penultimate line.

| Binding form | binds | in |
|---|---|---|
| Terms: | | |
| $\lambda(x:A)a$ | $x$ | $a$ |
| $\varsigma(x:A)a$ | $x$ | $a$ |
| let $x:A=a$ in $b$ | $x$ | $b$ |
| match $a:A$ with cons$(x,y) \Rightarrow b$ \| nil $\Rightarrow b'$ | $x,y$ | $b$ |
| Types: | | |
| $\mu(X).A$ | $X$ | $A$ |

Figure 4.3: Binding forms in **Q**ς

### 4.1.4 Definitions

The set of free type and term variables in a term $a$ is written $\mathsf{FV}(a)$, and in a type $A$ as $\mathsf{FV}(A)$. The capture-free substitution of a term $a$ for term variable $x$ in another term $b$ is written $b\{\!\{a/x\}\!\}$, and similarly for types, $B\{\!\{A/X\}\!\}$. The effect of **Q**ς's various binding constructs is summarized in Figure 4.3, and gives rise to the obvious definition for substitution.

**Definition 4.1** (Closed terms)**.** *As usual, a term, a, is* closed *when* $\mathsf{FV}(a)=\emptyset$.

**Definition 4.2** (Fresh names)**.** *A term variable $x$ is* fresh *for a term a, written $x \# a$ iff $x \notin \mathsf{FV}(a)$. Similarly for type variables X, and types A.*

## 4.2 Heap query

Heap query is a means by which objects may be selected from all those that exist at some point in program execution, according to whether or not they satisfy some property. Specifically, to give a value to the comprehension:

$$\{\, o \in \mathsf{Object} \mid \psi(o) \,\} \subseteq \text{existing objects}$$

The goal is then to decide upon an appropriate class of properties, of which $\psi$ is a member, and to ensure that the mechanism and result of such queries respect both the type system and programmer intuition.

### 4.2.1 Expressivity of query

When deciding upon a class of properties determinable by our query mechanism, we must return to our original goal — the provision of relationships in an object-based language. As discussed in Section 1.2, the fundamental form of inter-object relationship is a reference, upon which a network of back-pointers and collection objects may be built to represent $n$-ary, many-to-many, two-way relationships. Instead, we argue that heap query may take the strain of such book-keeping,

allowing the programmer to use a single reference to represent membership of a variety of different relationships.

In the following term, we create two objects assigned to $x$ and $y$ respectively, then relate them by constructing an object that references them both:

$$\begin{array}{l} \text{let } x : A = o_1 \text{ in} \\ \quad \text{let } y : B = o_2 \text{ in} \\ \quad\quad \text{let } r : [l_a : A,\ l_b : B] = [l_a = x,\ l_b = y] \text{ in } \ldots \end{array}$$

The objects $o_1$ and $o_2$ are unrelated until the object bound to $r$ is constructed, at which point they are related by that object. Their rôles in the object are specified by the labels $l_a$ and $l_b$ respectively. In order to determine which objects are thus related to $o_1$, for example, we can constuct the following set of objects that relate $o_1$ to something else, where $\iota_1$ and $\iota_2$ are the object identifiers for $o_1$ and $o_2$ respectively:

$$\{ \iota \mid \exists A', \iota' : \iota \text{ is the object id for } [l_a = \iota_1, l_b = \iota', \ldots] \text{ and } \iota \text{ exists in the heap} \}$$

Thus, we obtain all objects such as that bound to $r$, where $o_1$ is assocated with label $l_a$ and some other object is assocated with $l_b$. If we were to extend the term above with the following:

$$\begin{array}{l} \text{let } z : B = o_3 \text{ in} \\ \quad \text{let } r' : [l_a : A,\ l_b : B] = [l_a = x,\ l_b = z] \text{ in } \ldots \end{array}$$

then both $r$ and $r'$ would be included in the result.

In the following sections we describe how such queries shall be expressed in **Qς**, how query results are determined, and how these mechanisms interact with the structural type system of object-based calculi.

### 4.2.2 Query syntax

A query of the heap requests objects which have certain labels with specified values. For example, if we were to want all buttons that are blue, then we might issue the following query:

$$\begin{array}{l} \text{let } \mathit{blue} : \mathit{Colour} = [\ldots] \text{ in} \\ \quad\quad \ldots \\ \text{let } \mathit{blueButtons} : \text{list of } A = \langle \mathit{getColour} = \mathit{blue} \rangle \end{array}$$

where we intend $\langle \mathit{getColour} = \mathit{blue} \rangle$ to return all the objects in the heap whose *getColour* label returns the value bound to the *blue* variable. Immediately, we observe two challenges posed by such a query in a typed setting:

1. The result of the query is bound to the variable *blueButtons* with type

$$
\begin{array}{rll}
lv \in \mathsf{listval} ::= & \mathsf{nil} \mid \mathsf{cons}(v, lv) & \\
Q \in \mathsf{querytype} ::= & O \mid \mu(X).O & \text{queryable types} \\
qv \in \mathsf{queryval} ::= & \iota \mid \mathsf{fold}(A, qv) & \text{queryable values} \\
v \in \mathsf{value} ::= & \lambda(x:A)a \mid qv \mid lv & \text{values} \\
a, b \in \mathsf{term} ::= & \ldots & \\
& \mid \langle l_i : Q_i = a_{i \in 1..n}, l_{j \in n+1..m} : A_j = ?\rangle & \text{object query } (l_i, l_j \text{ distinct})
\end{array}
$$

Figure 4.4: Additions to the **Q**ς grammar for heap query

list of $A$, but it is not obvious which types may be represented by $A$. In this case, we know that *blue*, the value for which we are querying, is of type *Colour*, so $A \equiv [getColour : Colour]$ seems like a reasonable type. However, we may wish to use more complicated terms to provide values for queries, rather than restricting ourselves to term variables. In that case, type inference will be required. As usual when faced with such a difficulty, we counter by annotating labels with types, just as we do for ς-, $\lambda$- and let-bound variables:

$$\langle getColour : Colour = blue \rangle$$

Now, the query's type may be read directly from its definition.

2. Furthermore, if we were to want to invoke *click* on all blue buttons in the system, then the type system would not allow it. We could extend the query so that *click* is requested with a value equal to *click*'s implementation, but this breaks the most fundamental encapsulation provided to object-oriented programmers and forces us to test the equality of functions.

   Instead, we provide *wildcards* in object queries, which guarantee that the specified label will be present in each object of the result, and with the specified type:

   $$\langle getColour : Colour = blue, click : Click = ? \rangle$$

   The result will be the list of blue buttons upon which the type system will allow us to safely select the *click* label, but without requiring the programmer to know the implementation of the *click* method itself.

We give the additional portions of the **Q**ς grammar that support heap query in Figure 4.4. For convenience, the full grammar is given in Figure 4.5.

### 4.2.3 Completeness of result

Consider the following term, in which *DialogWindow* <: *Window*:

$$\begin{array}{l} \mathsf{let}\ wnd : DialogWindow = \ldots\ \mathsf{in} \\ \quad \mathsf{let}\ yesBtn : Button = [parent : Window = wnd, \ldots]\ \mathsf{in} \\ \quad\quad \mathsf{let}\ noBtn : DialogButton = [parent : DialogWindow = wnd, \ldots]\ \mathsf{in}\ \ldots \end{array}$$

Here, a *DialogWindow* is created, the intention being that a *DialogWindow* is a specialization of *Window* which acts as a dialogue box. Two buttons are then created: the first is typed *Button*; the second as *DialogButton*, which reflects the fact that a *DialogButton* may only be contained in a *DialogWindow*. Given that a *DialogWindow* is a *Window*, however, both buttons are contained inside that assigned to *wnd*, as specified by the buttons' *parent* label.

When querying which buttons are contained in the window represented by *wnd*, then, it is reasonable to pose the following query:

$$\langle parent : Window = wnd, click : Click = ?\rangle$$

As we have already established, this query will return a result of type:

$$\mathsf{list\ of}\ [parent : Window, click : \mathsf{Top}]$$

We would expect such a query to return both *noBtn* and *yesBtn*; after all, both have *parent* and *click* labels at a type compatible with *Window* and, in the case of *parent*, the value requested by the query. However, whilst the type of *yesBtn* is equal to that of the list's element type, the type of *noBtn* is unrelated to the element type owing to the absence of depth subtyping. Clearly, *noBtn* may not be soundly returned by such a query. Conversely, *yesBtn* may not be returned in the result of the following query, as its *parent* label is typed *Window*:

$$\langle parent : DialogWindow = wnd, click : Click = ?\rangle$$

Thus, a query's type annotations do more than simplify type-checking: they form further constraints on returned objects just as a query's values and labels do.

Where the programmer considers the type annotations to be part of a query's constraint set, the results obtained are complete. However, this may seem like a convenient choice of perception. If one were to regard the annotations simply as a typing aid, the problem may be attacked by the standard approaches to obtaining sound depth-subtyping. One such approach involves annotating labels with their variance: '+' for a covariant label; and '−' for contravariance [60].

$$A <: B \text{ implies } [l^+ : A] <: [l^+ : B]$$
$$A <: B \text{ implies } [l^- : B] <: [l^- : A]$$

| | |
|---|---|
| $l \in \mathsf{label}$ | |
| $\iota \in \mathsf{loc}$ | |
| $X, Y \in \mathsf{tyvar}$ | |
| $O \in \mathsf{otype} ::= [l_i : A_{i \in 1..n}]$ | object type ($l_i$ distinct) |
| $Q \in \mathsf{querytype} ::= O \mid \mu(X).O$ | queryable types |
| $A, B \in \mathsf{type} ::= \mathsf{Top} \mid X \mid O \mid A \to B \mid \mu(X).A \mid \mathsf{list\ of}\ A$ | |
| $lv \in \mathsf{listval} ::= \mathsf{nil} \mid \mathsf{cons}(v, lv)$ | |
| $qv \in \mathsf{queryval} ::= \iota \mid \mathsf{fold}(A, qv)$ | queryable values |
| $v \in \mathsf{value} ::= \lambda(x : A)a \mid qv \mid lv$ | values |
| $m \in \mathsf{meth} ::= \varsigma(x : A)b$ | method |
| $o \in \mathsf{obj} ::= [l_i = m_{i \in 1..n}]$ | object ($l_i$ distinct) |
| $a, b \in \mathsf{term} ::= \iota$ | location |
| $\mid x$ | variable |
| $\mid o$ | object |
| $\mid \lambda(x : A)a \mid a(b)$ | $\lambda$-abstraction/application |
| $\mid \mathsf{let}\ x : A = a\ \mathsf{in}\ b$ | let |
| $\mid a.l$ | selection |
| $\mid a.l \Leftarrow m$ | update |
| $\mid \mathsf{clone}(a)$ | clone |
| $\mid \mathsf{fold}(A, a)$ | type folding |
| $\mid \mathsf{unfold}(a)$ | type unfolding |
| $\mid \mathsf{nil} \mid \mathsf{cons}(a, b)$ | list constructors |
| $\mid (\mathsf{match}\ a : A\ \mathsf{with}$ | |
| $\mathsf{cons}(x, x') \Rightarrow b \mid \mathsf{nil} \Rightarrow b')$ | list destructor |
| $\mid \langle l_i : Q_i = a_{i \in 1..n}, l_{j \in n+1..m} : A_j = ?\rangle$ | object query ($l_i, l_j$ distinct) |

Figure 4.5: **Q**$\varsigma$ grammar

The type system precludes potentially unsound operations, specifically that a covariant label cannot be updated, whilst a contravariant label cannot be read. A full treatment of variance annotations for the $\varsigma$-calculus is given by Abadi and Cardelli [1]. Similar features are also being adopted for mainstream languages, where variance with respect to the parameters of generic types is allowed in Java [38] and has been proposed for C# [34]. Whether these features are well-understood by programmers remains an open question.

(Env Empty)
$$\frac{}{\vdash_{\mathbf{Q}\varsigma} \epsilon \ \mathsf{ok}}$$

(Env Var)
$$\frac{E \vdash_{\mathbf{Q}\varsigma} A \qquad x \notin \mathsf{dom}(E)}{\vdash_{\mathbf{Q}\varsigma} E, x : A \ \mathsf{ok}}$$

(Env TypVar)
$$\frac{E \vdash_{\mathbf{Q}\varsigma} A \qquad X \notin \mathsf{dom}(E)}{\vdash_{\mathbf{Q}\varsigma} E, X <: A \ \mathsf{ok}}$$

(Env Loc)
$$\frac{E \vdash_{\mathbf{Q}\varsigma} [l_i : A_{i \in 1..n}] \qquad \iota \notin \mathsf{dom}(E)}{\vdash_{\mathbf{Q}\varsigma} E, \iota : [l_i : A_{i \in 1..n}] \ \mathsf{ok}}$$

Figure 4.6: Rules specifying well-formed environments, written $\vdash_{\mathbf{Q}\varsigma} E \ \mathsf{ok}$

## 4.3 Type system

The type system formalizes the features discussed earlier in this chapter by giving rules which may later be subjected to formal proof.

**Typing environments** Typing is performed in the presence of a typing environment, which assigns types to term variables and object addresses, and gives bounds for $\mu$-bound type variables.

| | | |
|---|---|---|
| $E \in \mathsf{Environment} ::=$ | $\epsilon$ | empty environment |
| | $\mid E, x : A$ | term variable type |
| | $\mid E, \iota : A$ | store location type |
| | $\mid E, X <: A$ | type variable bound |

When an environment $E$ is well-formed, we write $\vdash_{\mathbf{Q}\varsigma} E \ \mathsf{ok}$. Such environments are contained in the least set that is closed under the rules given in Figure 4.6, which make use of the judgement that a type $A$ is valid in environment $E$, written $E \vdash_{\mathbf{Q}\varsigma} A$ and defined by the rules in Figure 4.7.

(Env Empty) specifies that the empty environment is well-formed whilst the (Env Var) and (Env Loc) rules allow term variables and addresses to be assigned any well-formed type as long as they have not been previously defined in the environment. (Env TypVar) imposes similar restrictions on the bounds of type variables.

**Well-formed types** $\mathsf{Top}$ is a valid type in any valid environment, as specified by (IsTypTop) in Figure 4.7. Object types, function types and list types are well-formed according to (IsTypObj), (IsTypFun) and (IsTypList) respectively, only when their constituent types are well-formed. In the case of object types, recall that we only consider finite object types where the labels are distinct. (IsTypVar) permits type variables as valid types when they have a bound specified in the environment, which must itself be well-formed. (IsTypRec) checks the validity

(ISTYPTOP)
$$\frac{\vdash_{\mathbf{Q}\varsigma} E \text{ ok}}{E \vdash_{\mathbf{Q}\varsigma} \mathsf{Top}}$$

(ISTYPOBJ)
$$\frac{E \vdash_{\mathbf{Q}\varsigma} B_{i\in 1..n}}{E \vdash_{\mathbf{Q}\varsigma} [l_i : B_{i\in 1..n}]}$$

(ISTYPVAR)
$$\frac{\vdash_{\mathbf{Q}\varsigma} E', X <: A, E'' \text{ ok}}{E', X <: A, E'' \vdash_{\mathbf{Q}\varsigma} X}$$

(ISTYPFUN)
$$\frac{E \vdash_{\mathbf{Q}\varsigma} A \qquad E \vdash_{\mathbf{Q}\varsigma} B}{E \vdash_{\mathbf{Q}\varsigma} A \to B}$$

(ISTYPLIST)
$$\frac{E \vdash_{\mathbf{Q}\varsigma} A}{E \vdash_{\mathbf{Q}\varsigma} \text{list of } A}$$

(ISTYPREC)
$$\frac{E, X <: \mathsf{Top} \vdash_{\mathbf{Q}\varsigma} B \qquad X \notin \mathsf{dom}(E)}{E \vdash_{\mathbf{Q}\varsigma} \mu(X).B}$$

Figure 4.7: Rules specifying well-formed types,written $E \vdash_{\mathbf{Q}\varsigma} A$.

of the body of recursive types in the presence of a bound on the type's $\mu$-bound variable to determine the validity of a recursive type.

Types in environments are checked for validity from right to left and environment ordering is important: a type later in the environment may mention type variables which acquire their bound earlier on. For example, it is the case that $\vdash_{\mathbf{Q}\varsigma} X <: \mathsf{Top}, \iota : X$ ok but not that $\vdash_{\mathbf{Q}\varsigma} \iota : X, X <: \mathsf{Top}$ ok. In the latter case, the derivation requires $\epsilon \vdash_{\mathbf{Q}\varsigma} X$, which does not hold in the absence of a bound for $X$ in the environment. Similarly, mutually-bounding variables are quite sensibly precluded: $\vdash_{\mathbf{Q}\varsigma} X <: Y, Y <: X$ ok does not hold.

Whilst the rules defining well-formed environments and well-formed types are mutually-recursive, they are decidable: each ENV-rule is an axiom or shrinks the environment and each ISTYP-rule, with the exception of (ISTYPREC) shrinks the type or uses environment well-formedness. Types have only a finite number of recursive bindings and no rule grows a type, so a derivation ending in (ISTYPREC) will always reach a sub-derivation without a $\mu$-binding in outer position.

**Subtyping** When $A$ is a subtype of $B$ in the presence of an environment $E$, we write $E \vdash_{\mathbf{Q}\varsigma} A <: B$ for the least relation closed under the rules of Figure 4.8.

(SUB TOP) establishes all types as subtypes of Top, whilst (SUB OBJECT) allows an object type to be a subtype of any other object type which is a prefix. Recall that we identify objects and object types up to reordering of their labels, so we can consider only prefixes without imposing any real restriction. (SUB LIST) and (SUB FUN) specify that list types are covariant in their element type, while function types are contravariant in their argument type and covariant in their result type. (SUB VAR) propagates subtype relationships for type variables from the environment into the subtyping relation, while (SUB REC) allows two recursive types to be subtypes if their bodies are subtypes in the presence of bounds for their $\mu$-bound type variables [1, 60]. Finally, (SUB REFL) and (SUB TRANS) provide the

(SUB TOP)

$$\frac{E \vdash_{\mathbf{Q}\varsigma} A}{E \vdash_{\mathbf{Q}\varsigma} A <: \mathsf{Top}}$$

(SUB OBJECT)

$$\frac{\forall i \in 1..n+m : E \vdash_{\mathbf{Q}\varsigma} A_i \qquad m \geq 0}{E \vdash_{\mathbf{Q}\varsigma} [l_i : A_{i \in 1..n+m}] <: [l_i : A_{i \in 1..n}]}$$

(SUB LIST)

$$\frac{E \vdash_{\mathbf{Q}\varsigma} A <: B}{E \vdash_{\mathbf{Q}\varsigma} \mathsf{list\ of}\ A <: \mathsf{list\ of}\ B}$$

(SUB FUN)

$$\frac{E \vdash_{\mathbf{Q}\varsigma} A' <: A \qquad E \vdash_{\mathbf{Q}\varsigma} B <: B'}{E \vdash_{\mathbf{Q}\varsigma} A \to B <: A' \to B'}$$

(SUB VAR)

$$\frac{\vdash_{\mathbf{Q}\varsigma} E', X <: A, E'' \ \mathsf{ok}}{E', X <: A, E'' \vdash_{\mathbf{Q}\varsigma} X <: A}$$

(SUB REC)

$$\frac{E \vdash_{\mathbf{Q}\varsigma} \mu(X).A \qquad E \vdash_{\mathbf{Q}\varsigma} \mu(Y).B \qquad E, Y <: \mathsf{Top}, X <: Y \vdash_{\mathbf{Q}\varsigma} A <: B}{E \vdash_{\mathbf{Q}\varsigma} \mu(X).A <: \mu(Y).B}$$

(SUB REFL)

$$\frac{E \vdash_{\mathbf{Q}\varsigma} A}{E \vdash_{\mathbf{Q}\varsigma} A <: A}$$

(SUB TRANS)

$$\frac{E \vdash_{\mathbf{Q}\varsigma} A <: B \qquad E \vdash_{\mathbf{Q}\varsigma} B <: C}{E \vdash_{\mathbf{Q}\varsigma} A <: C}$$

Figure 4.8: Rules defining the subtyping relation, $E \vdash_{\mathbf{Q}\varsigma} A <: B$

reflexive and transitive closure of the relation established by the preceding rules.

Notice that only well-formed types take part in the subtype relation.

**Typing relation** The type system assigns a type $A$ to term $a$ in the presence of typing environment $E$, written $E \vdash_{\mathbf{Q}\varsigma} a : A$. This relation is the least such closed under the rules of Figure 4.9.

Subtyping is by subsumption, expressed by (TYP SUB), so that subtyping is available at all points in a type derivation in comparison with RelJ where subtyping may be used only at specific locations, for example at method invocation. This comes at the cost of non-structural rules for typing.

Term variables and object addresses are typed directly from the environment in (TYP VAR) and (TYP VAL) respectively. Empty lists may be typed with any well-formed list type in (TYP NIL), whilst (TYP CONS) ensures that the type of the head of a list matches that of its tail.

The typing of object construction by (TYP OBJECT) ensures that method bodies are well-typed when the self type declaration is transferred into the environment. The types given as self types in method declarations must all match, and also match the derived type for the object.

(TYP SELECT) types label selection simply by projection from the object type, while (TYP UPDATE) ensures that new method bodies are well-typed similarly to those in (TYP OBJECT). Notice that the self type declared in the new method body may not match either the originally derived type for the target object or the

(TYP SUB)

$$\frac{E \vdash_{\mathbf{Q}\varsigma} A <: B \qquad E \vdash_{\mathbf{Q}\varsigma} a : A}{E \vdash_{\mathbf{Q}\varsigma} a : B}$$

(TYP VAR)

$$\frac{\vdash_{\mathbf{Q}\varsigma} E', x : A, E'' \text{ ok}}{E', x : A, E'' \vdash_{\mathbf{Q}\varsigma} x : A}$$

(TYP VAL)

$$\frac{\vdash_{\mathbf{Q}\varsigma} E', \iota : [l_i : A_{i \in 1..n}], E' \text{ ok}}{E', \iota : [l_i : A_{i \in 1..n}], E'' \vdash_{\mathbf{Q}\varsigma} \iota : [l_i : A_{i \in 1..n}]}$$

(TYP NIL)

$$\frac{E \vdash_{\mathbf{Q}\varsigma} A}{E \vdash_{\mathbf{Q}\varsigma} \text{nil} : \text{list of } A}$$

(TYP CONS)

$$\frac{E \vdash_{\mathbf{Q}\varsigma} a : A \qquad E \vdash_{\mathbf{Q}\varsigma} b : \text{list of } A}{E \vdash_{\mathbf{Q}\varsigma} \text{cons}(a, b) : \text{list of } A}$$

(TYP OBJECT)

$$\frac{E, x_i : A \vdash_{\mathbf{Q}\varsigma} b_i : A_{i \in 1..n} \qquad A \equiv [l_i : A_{i \in 1..n}] \qquad x_{i \in 1..n} \notin \text{dom}(E)}{E \vdash_{\mathbf{Q}\varsigma} [l_i = \varsigma(x_i : A) b_{i \in 1..n}] : A}$$

(TYP SELECT)

$$\frac{E \vdash_{\mathbf{Q}\varsigma} a : [l_i : A_{i \in 1..n}] \qquad j \in 1..n}{E \vdash_{\mathbf{Q}\varsigma} a.l_j : A_j}$$

(TYP UPDATE)

$$\frac{E \vdash_{\mathbf{Q}\varsigma} a : A \qquad E, x : A \vdash_{\mathbf{Q}\varsigma} b : A_j \qquad x \notin \text{dom}(E) \qquad j \in 1..n \qquad A \equiv [l_i : A_{i \in 1..n}]}{E \vdash_{\mathbf{Q}\varsigma} a.l_j \Leftarrow \varsigma(x : A) b : A}$$

(TYP CLONE)

$$\frac{E \vdash_{\mathbf{Q}\varsigma} a : [l_i : A_{i \in 1..n}]}{E \vdash_{\mathbf{Q}\varsigma} \text{clone}(a) : [l_i : A_{i \in 1..n}]}$$

(TYP LET)

$$\frac{E \vdash_{\mathbf{Q}\varsigma} a : A \qquad E, x : A \vdash_{\mathbf{Q}\varsigma} b : B \qquad x \notin \text{dom}(E)}{E \vdash_{\mathbf{Q}\varsigma} \text{let } x : A = a \text{ in } b : B}$$

(TYP LAM)

$$\frac{E, x : A \vdash_{\mathbf{Q}\varsigma} b : B}{E \vdash_{\mathbf{Q}\varsigma} \lambda(x : A) b : A \to B}$$

(TYP APP)

$$\frac{E \vdash_{\mathbf{Q}\varsigma} b : A \to B \qquad E \vdash_{\mathbf{Q}\varsigma} a : A}{E \vdash_{\mathbf{Q}\varsigma} b(a) : B}$$

(TYP MATCH)

$$\frac{x, x' \notin \text{dom}(E) \qquad E \vdash_{\mathbf{Q}\varsigma} a : \text{list of } A \qquad E, x : A, x' : \text{list of } A \vdash_{\mathbf{Q}\varsigma} b : B \qquad E \vdash_{\mathbf{Q}\varsigma} b' : B}{E \vdash_{\mathbf{Q}\varsigma} \text{match } a : \text{list of } A \text{ with cons}(x, x') \Rightarrow b \mid \text{nil} \Rightarrow b' : B}$$

(TYP FOLD)

$$\frac{E \vdash_{\mathbf{Q}\varsigma} b : B\{\!\{A/X\}\!\} \qquad A \equiv \mu(X).B}{E \vdash_{\mathbf{Q}\varsigma} \text{fold}(A, b) : A}$$

(TYP UNFOLD)

$$\frac{E \vdash_{\mathbf{Q}\varsigma} a : A \qquad A \equiv \mu(X).B}{E \vdash_{\mathbf{Q}\varsigma} \text{unfold}(a) : B\{\!\{A/X\}\!\}}$$

(TYP QUERY)

$$\frac{E \vdash_{\mathbf{Q}\varsigma} a_i : Q_i \qquad E \vdash_{\mathbf{Q}\varsigma} A_j \qquad \text{for all } i \in 1..n, j \in n+1..m}{E \vdash_{\mathbf{Q}\varsigma} \langle l_i : Q_i = a_{i \in 1..n}, l_j : A_{j \in n+1..m} = ? \rangle : \text{list of } [l_i : Q_{i \in 1..n}, l_j : A_{j \in n+1..m}]}$$

Figure 4.9: Rules defining the typing relation $E \vdash_{\mathbf{Q}\varsigma} a : A$

method's originally declared self type; we require only that it matches a derivable type for the update's target which, in the presence of subsumption, will be a supertype of that originally derived for the object. (TYP CLONE) types object copies in the obvious way.

Typing for the let-bindings and $\lambda$-calculus portions is entirely standard, and is given by (TYP LET), (TYP LAM) and (TYP APP). (TYP MATCH) types list destruction in a similar way as for let.

Type folding and unfolding is also standard [1, 60]: the folding of a term to a recursive type $\mu(X).B$ is valid as long as the term can be typed to the unfolded type, $B\{\!\{\mu(X).B/X\}\!\}$, and *vice versa* for type unfolding.

Finally, we type heap queries with (TYP QUERY). We check that the query's sub-terms are typable according to the query's type annotations, and that the types assigned to the query's wildcards are valid. The query's type is then a list of object types conforming to those declared in the body of the query. We restrict non-wildcard type annotations to the set of 'queryable types', which guarantees that the values for which we must query will be object addresses (possibly inside some folds), and not, for example, $\lambda$-expressions.

**Principal types** It is clear that, in the presence of subsumption, the types assigned to terms are not unique. However, it is conceptually useful to have a canonical, or 'best', type; that is, the type that allows the greatest number of labels to be selected from a term. Such a type for a term, $a$, where it exists and is unique, is known as $a$'s *principal type*. **Qς** does not have principal types in general — for example, the empty list constructor nil may take any well-formed list type. However, it is useful to note that objects and addresses do have principal types:

**Theorem 4.1** (Principal types)**.** *The* principal type *of a term is the unique type which is a subtype of all other types assignable to the term. Formally, the principal type $A_P$ is the unique type that satisfies:*

$$E \vdash_{\mathbf{Q}\varsigma} a : A_P \text{ and, for all } A,\ E \vdash_{\mathbf{Q}\varsigma} a : A \text{ implies } E \vdash_{\mathbf{Q}\varsigma} A_P <: A$$

*In **Qς**, objects and object locations have principal types.*

*Proof.* The proof is based on the observation that (TYP OBJECT) and (TYP VAL) uniquely determine the types assigned to objects and locations, whilst (TYP SUB) may only assign supertypes of those types previously derived.

In (TYP OBJECT), objects are assigned the types extracted from their labels' self types, which must all match according to the same rule. In (TYP VAL), object addresses, $\iota$, are typed from the environment which, by the environment well-formedness conditions, must assign a type to $\iota$ uniquely. The result then follows from an inductive step for (TYP SUB), and from the transitivity of subtyping specified in (SUB TRANS). □

This property is particularly useful when considering the typing constraints previously discussed for heap query: without a canonical type for an object, it becomes less clear whether these constraints are satisfied.

## 4.4 Semantics

We specify the execution of **Q**ς programs as a transition relation between program states or *configurations*. A configuration consists of a term and an object store:

$$c \in \mathsf{Configuration} ::= (a, \sigma) \qquad \text{configuration}$$
$$\sigma \in \mathsf{Store} ::= \{\iota_i \mapsto o_{i \in 1..n}\} \qquad \text{object store } (\iota_i \text{ distinct})$$

The store, $\sigma$, maps addresses, $\iota$, to the object located at that address, $o$. Each address may be mapped to no more than one object and there are infinitely many addresses, though each store is a finite map.

Store update is written $\sigma[\iota \mapsto o]$, which represents the update (or addition) of location $\iota$ in the store, which will subsequently map to object $o$. All other store locations are unchanged.

We can use the typing relation from the previous section to enforce some expectations about the store. (TYP STORE) in Figure 4.10 ensures that each location in the store has a type in the environment and that the object is well-typed with respect to that environment. In particular, all free variables and store locations mentioned in any object must have an entry in the environment. When a store, $\sigma$, is well-formed in the presence of an environment, $E$, we write $E \vdash_{\mathbf{Q}\varsigma} \sigma$.

A configuration may also be well-formed and well-typed, in which case we write $E \vdash_{\mathbf{Q}\varsigma} (a, \sigma) : A$. This relation is defined by the rule (TYP CONFIG), which checks that a configuration's store is well-formed, its term well-typed, and that the domains of the environment, $E$, and store, $\sigma$, are identical. This final condition ensures that only addresses in the store have entries in the environment, so that there can be no dangling pointers — if there were, the containing object in the store would not be type-correct whilst checking store well-formedness. As the store's domain contains only addresses, this also implies that there are no term or type variables in the environment and thus in the configuration's term or in any terms in object methods. All such terms must therefore be closed.

Execution is represented by a transition relation, $\rightsquigarrow_{\mathbf{Q}\varsigma}$, which expresses a single execution step between configurations:

$$\rightsquigarrow_{\mathbf{Q}\varsigma} \subseteq \mathsf{Configuration} \times \mathsf{Configuration}$$

This relation is defined by the axioms in Figure 4.12, which make use of evaluation contexts [77] to frame execution order. Just as for RelJ, evaluation contexts contain a single hole, denoted $\bullet$, which marks the position of the sub-term to be executed. A context, $\mathscr{E}$, is instantiated to a term, $a$, by the substitution of a sub-term, $b$, for the context's hole: this is written $a = \mathscr{E}[b]$. During execution

(Typ Store)

$$\dfrac{\begin{array}{l}\forall \iota \in \mathsf{dom}(\sigma): \\ \quad \sigma(\iota) = [l_i = \varsigma(x_i : A)b_{i \in 1..n}] \qquad A \equiv [l_i : A_{i \in 1..n}] \\ \quad E \vdash_{\mathbf{Q}\varsigma} \sigma(\iota) : A \qquad E \equiv E', \iota : A, E''\end{array}}{E \vdash_{\mathbf{Q}\varsigma} \sigma}$$

(Typ Config)

$$\dfrac{E \vdash_{\mathbf{Q}\varsigma} \sigma \qquad E \vdash_{\mathbf{Q}\varsigma} a : A \qquad \mathsf{dom}(E) = \mathsf{dom}(\sigma)}{E \vdash_{\mathbf{Q}\varsigma} (a, \sigma) : A}$$

Figure 4.10: Rules defining well-formed stores, and well-typed configurations

| | |
|---|---|
| $\mathscr{E} \in \mathsf{context} ::= \bullet$ | hole |
| $\mid \mathscr{E}(b) \mid v(\mathscr{E})$ | $\lambda$-application |
| $\mid \mathsf{let}\ x : A = \mathscr{E}\ \mathsf{in}\ b$ | let |
| $\mid \mathsf{fold}(A, \mathscr{E})$ | type fold |
| $\mid \mathsf{unfold}(\mathscr{E})$ | type unfold |
| $\mid \mathscr{E}.l$ | select |
| $\mid \mathscr{E}.l \Leftarrow \varsigma(x : A)b$ | update |
| $\mid \mathsf{clone}(\mathscr{E})$ | clone |
| $\mid \mathsf{cons}(\mathscr{E}, a) \mid \mathsf{cons}(v, \mathscr{E})$ | list |
| $\mid (\mathsf{match}\ \mathscr{E} : A\ \mathsf{with}\ \mathsf{cons}(x, x') \Rightarrow b \mid \mathsf{nil} \Rightarrow b')$ | list match |
| $\mid \langle l_i : A_i = v_{i \in 1..n}, l_{n+1} : A_{n+1} = \mathscr{E}, l_j : A_j = a_{j \in n+2..m}, l_k : A_{k \in m+1..p} = ?\rangle$ | query |

Figure 4.11: Evaluation contexts

of a term, one can imagine the decomposition of the term into a context and a reducible expression (or *redex*). The redex is transformed by one of the rules of Figure 4.12 into a new sub-term, and the result of that execution step shall be the new sub-term wrapped in the original context — the continuation.

The definition of the evaulation contexts for **Q**ς is given in Figure 4.11. The $\lambda$-calculus evaluation order is standard for a call-by-value semantics. let-bound expressions are fully evaluated before the term in which the binding occurs. We fully evaluate terms that are to be folded or unfolded, cloned, used for label selection, for method update, or for list construction or destruction. We also

evaluate all terms in queries to values, from left to right. Notice that no execution occurs underneath let-bindings, inside method bodies at update, or underneath a match — in particular it is useful to note that no variables are bound in sub-terms in reduction position.

We now describe the semantics of the redexes themselves, given by the rules in Figure 4.12. Object definitions are simply copied to the store in (RED OBJECT), having been allocated a new store location. (RED CLONE) is similar, but copies the object from the store location which is to be cloned, rather than from the executed term. (RED SELECT) looks up the appropriate object's method body in the store — the new configuration's term is the method body where the self parameter has been substituted for the receiving location. Similarly, (RED UPDATE) locates the object to be updated, and updates the target location so that the new method replaces that previously found at the updated label. Notice that the self type is re-written to match those of methods already in the object, thus the new object remains typable according to (TYP OBJECT) as all self types match, and the method remains typable as its bound variable's type becomes only more permissive.

Reduction of let-bindings in (RED LET) and $\lambda$-applications in (RED APP) is standard, as is the reduction of a fold/unfold pair in (RED FOLD), where an unfold cancels out an immediately enclosed fold.

(RED MATCH CONS) and (RED MATCH NIL) correspond to the two cases of list destruction. In the former, a match with a non-nil list reduces to the first branch where the head and tail of the list are substituted for the variables in the pattern; in the latter, the term is reduced to the second branch without any substitutions.

Finally, (RED QUERY) evaluates heap queries by returning a list of all the objects in the store whose types match that required by the query's type annotations and whose values match those given in the query's non-wildcard labels. This list is constructed using a *deterministic list comprehension*:

**Definition 4.3** (Deterministic list comprehension). *A deterministic list comprehension is written*

$$\langle\!\langle z \mid \psi(z) \rangle\!\rangle$$

*and returns a list containing exactly the elements of the set comprehension* $\{z \mid \psi(z)\}$ *in some order. This order is preserved regardless of the context in which the comprehension is used. This may be accomplished with the use of some total order; for example, store locations may be sorted by their order of allocation.*[4]

Maintaining a deterministic semantics in this way — as opposed to using a set comprehension — simplifies the later soundness proof and, given that our study is not one of non-determinism, does not preclude any useful or interesting execution paths. It would be of little importance to a programmer in practice.

[4]Such use of a total order is more powerful than we require — the ordering need not be consistent between different comprehensions.

(Red Object)

$$\frac{\sigma' = \sigma[\iota \mapsto o] \quad \iota \notin \mathrm{dom}(\sigma)}{(\mathscr{E}[o], \sigma) \rightsquigarrow_{\mathbf{Q}\varsigma} (\mathscr{E}[\iota], \sigma')}$$

(Red Select)

$$\frac{\sigma(\iota) = [l_i = \varsigma(x_i : A)b_{i \in 1..n}] \quad j \in 1..n}{(\mathscr{E}[\iota.l_j], \sigma) \rightsquigarrow_{\mathbf{Q}\varsigma} (\mathscr{E}[b_j \{\!\{\iota / x_j\}\!\}], \sigma)}$$

(Red Update)

$$\frac{\sigma(\iota) = [l_i = \varsigma(x_i : B)b_{i \in 1..n}] \quad j \in 1..n \quad \sigma' = \sigma[\iota \mapsto [l_i = \varsigma(x_i : B)b_{i \in 1..j-1}, l_j = \varsigma(x : B)b, l_i = \varsigma(x_i : B)b_{i \in j+1..n}]]}{(\mathscr{E}[\iota.l_j \Leftarrow \varsigma(x : A)b], \sigma) \rightsquigarrow_{\mathbf{Q}\varsigma} (\mathscr{E}[\iota], \sigma')}$$

(Red Clone)

$$\frac{\sigma(\iota) = o \quad \sigma' = \sigma[\iota' \mapsto o] \quad \iota' \notin \mathrm{dom}(\sigma)}{(\mathscr{E}[\mathrm{clone}(\iota)], \sigma) \rightsquigarrow_{\mathbf{Q}\varsigma} (\mathscr{E}[\iota'], \sigma')}$$

(Red Let)

$$\frac{}{(\mathscr{E}[\mathrm{let}\ x : A = v\ \mathrm{in}\ b], \sigma) \rightsquigarrow_{\mathbf{Q}\varsigma} (\mathscr{E}[b \{\!\{v / x\}\!\}], \sigma)}$$

(Red Fold)

$$\frac{}{(\mathscr{E}[\mathrm{unfold}(\mathrm{fold}(A, v))], \sigma) \rightsquigarrow_{\mathbf{Q}\varsigma} (\mathscr{E}[v], \sigma)}$$

(Red App)

$$\frac{}{(\mathscr{E}[(\lambda(x : A)b)(v)], \sigma) \rightsquigarrow_{\mathbf{Q}\varsigma} (\mathscr{E}[b \{\!\{v / x\}\!\}], \sigma)}$$

(Red Match Nil)

$$\frac{}{(\mathscr{E}[\mathrm{match\ nil} : A\ \mathrm{with\ cons}(x, x') \Rightarrow b \mid \mathrm{nil} \Rightarrow b'], \sigma) \rightsquigarrow_{\mathbf{Q}\varsigma} (\mathscr{E}[b'], \sigma)}$$

(Red Match Cons)

$$\frac{}{(\mathscr{E}[\mathrm{match\ cons}(v, lv) : A\ \mathrm{with\ cons}(x, x') \Rightarrow b \mid \mathrm{nil} \Rightarrow b'], \sigma) \rightsquigarrow_{\mathbf{Q}\varsigma} (\mathscr{E}[b \{\!\{v / x\}\!\} \{\!\{lv / x'\}\!\}], \sigma)}$$

(Red Query)

$$\frac{v \equiv \langle\!\langle \iota \mid \exists E : E \vdash_{\mathbf{Q}\varsigma} (\iota, \sigma) : [l_i : A_{i \in 1..m}] \text{ and } \sigma(\iota) = [l_i = qv_{i \in 1..n}, \ldots] \rangle\!\rangle}{(\mathscr{E}[\langle l_i : A_i = qv_{i \in 1..n}, l_j : A_{j \in n+1..m} = ?\rangle], \sigma) \rightsquigarrow_{\mathbf{Q}\varsigma} (v, \sigma)}$$

Figure 4.12: Substitution-based, small-step reduction semantics

(RED QUERY SYNTACTIC)

$$\frac{v = \langle\!\langle \iota \mid \sigma(\iota) = [l_k = \varsigma(x_k : A)qv_{k\in 1..m}, \ldots] \text{ and } A \equiv [l_i : A_{i\in 1..m}, \ldots]\rangle\!\rangle}{(\mathscr{E}[\langle l_i : A_i = qv_{i\in 1..n}, l_j : A_{j\in n+1..m} = ?\rangle], \sigma) \rightsquigarrow_{\mathbf{Q}\varsigma} (v, \sigma)}$$

Figure 4.13: A syntax-directed implementation of (RED QUERY)

(RED QUERY)'s use of the type system, and quantification over the environment, renders a direct implementation impractical. An alternative approach would involve the annotation of objects with their principal types, in order to avoid the invocation of the type system. This is precisely the approach taken for class-based languages, both in theory and in practice, where these annotations are usually referred to as the object's *dynamic type*. Whilst objects are not explicitly annotated with their principal types, recall that object methods are annotated with a self-type. We can therefore re-cast (RED QUERY) as a syntactic examination of each object, as shown in Figure 4.13.

To establish that these two approaches are equivalent, it suffices to show the following:

**Theorem 4.2.** *In a well-typed configuration:*

$$\exists E : E \vdash_{\mathbf{Q}\varsigma} (\iota, \sigma) : [l_i : A_{i\in 1..m}] \quad \textit{iff} \quad \sigma(\iota) = [l_i = \varsigma(x_i : A)a_{i\in 1..m}, \ldots] \textit{ and } A \equiv [l_i : A_{i\in 1..m}, \ldots]$$

*Proof.* Suppose that $E \vdash_{\mathbf{Q}\varsigma} (\iota, \sigma) : [l_i : A_{i\in 1..m}]$ for some environment $E$. Then from (TYP CONFIG), $E \vdash_{\mathbf{Q}\varsigma} \sigma$ and $E \vdash_{\mathbf{Q}\varsigma} \iota : [l_i : A_{i\in 1..m}]$. The latter implies that $\iota \in \mathsf{dom}(E)$ which, with (TYP CONFIG), implies that $\iota \in \mathsf{dom}(\sigma)$. (TYP STORE) requires that $\sigma(\iota) = [l_i = \varsigma(x_i : A)b_{i\in 1..n}]$ and $A \equiv [l_i : A_{i\in 1..n}]$. The original typing could have arisen either from (TYP VAL) or (TYP SUB), so $m = n$ or $m < n$. Therefore, $\sigma(\iota) = [l_i = \varsigma(x_i : A)a_{i\in 1..m}, \ldots]$ and $A \equiv [l_i : A_{i\in 1..m}, \ldots]$ as required.

From right-to-left, the property follows simply from the well-formedness of the configuration, hence the store, and the conditions of (TYP STORE). □

## 4.5 Soundness

In this section, we summarize the main properties of $\mathbf{Q}\varsigma$ required for soundness. As for RelJ, we focus on two key properties: that no type-correct program will get 'stuck' — except in a well-defined error state — and that types are preserved during program execution. $\mathbf{Q}\varsigma$ has both of these properties, and is thus sound. For the interested reader, we summarize the important steps in the proof of these properties in the remainder of this section.

### 4.5.1 Properties of well-formed environments and types

The definitions of type validity and environment well-formedness are mutually-referencing. Therefore, their properties must be established by induction over both sets of rules. The store is manipulated by execution, and the environments must keep up if the conditions set by (TYP CONFIG) are to be satisfied. Additionally, during typing, we add type variable bounds and address and term variable bindings to the environment. We therefore establish some properties of well-formed environments and types under addition, removal and reordering of environment elements:

**Lemma 4.3** (Well-formed environments and valid types). *We establish the following properties of well-formed environments and valid types:*

i. *Type validity is preserved by the extension of the environment as long as the new environment is also well-formed (specifically, as long as the extension mentions no invalid types or duplicates of variables already bound by the original environment):*

$$E_1, E_2 \vdash_{\mathbf{Q}\varsigma} A \text{ and } \vdash_{\mathbf{Q}\varsigma} E_1, E_3, E_2 \text{ ok implies } E_1, E_3, E_2 \vdash_{\mathbf{Q}\varsigma} A$$

ii. *Similarly, type validity and environment well-formedness are preserved by the removal of unused elements from the environment:*

$$E_1, E_2, E_3 \vdash_{\mathbf{Q}\varsigma} A \text{ and } \mathsf{dom}(E_2) \# E_3, A \text{ implies } E_1, E_3 \vdash_{\mathbf{Q}\varsigma} A$$
$$\vdash_{\mathbf{Q}\varsigma} E_1, E_2, E_3 \text{ ok and } \mathsf{dom}(E_2) \# E_3 \text{ implies } \vdash_{\mathbf{Q}\varsigma} E_1, E_3 \text{ ok}$$

*Clearly, term variable and address bindings may be removed immediately, as these are always fresh for environments and types.*

*Finally, as a corollary of the above properties:*

iii. *Environments remain well-formed and types remain valid under any reordering of the environment that respects dependencies:*

$$\vdash_{\mathbf{Q}\varsigma} E_1, E_2, E_3 \text{ ok and } \mathsf{dom}(E_2) \# E_3 \text{ implies } \vdash_{\mathbf{Q}\varsigma} E_1, E_3, E_2 \text{ ok}$$
$$E_1, E_2, E_3 \vdash_{\mathbf{Q}\varsigma} A \text{ and } \mathsf{dom}(E_2) \# E_3 \text{ implies } E_1, E_3, E_2 \vdash_{\mathbf{Q}\varsigma} A$$

*Proof.* The proofs proceed by ensuring the required property is closed under the ISTYP and ENV rules of Figures 4.6 and 4.7. As the $\vdash_{\mathbf{Q}\varsigma} E$ ok and $E \vdash_{\mathbf{Q}\varsigma} A$ relations are the least such relations closed under the rules, then they must be contained in the properties above. □

We must also check that types remain valid under substitutions resulting from traversal of the isomorphism between a recursive type and its unfolding.

**Lemma 4.4** (Validity of recursive type foldings/unfoldings)**.** *If the folding of a recursive type is a valid type, so is its unfolding and* vice versa*:*

$$E \vdash_{\mathbf{Q}\varsigma} \mu(X).B \textit{ iff } E \vdash_{\mathbf{Q}\varsigma} B\{\!\{\mu(X).B/X\}\!\}$$

*Proof.* By induction on the TYP rules of Figure 4.7:

**($\Longrightarrow$)** We observe that where $X$ is not a free variable in $B$, then the result is immediate. Where $X \in \mathsf{FV}(B)$, we show the following property is closed under the rules defining $E \vdash_{\mathbf{Q}\varsigma} A$:

$$E \vdash_{\mathbf{Q}\varsigma} A \text{ and } (A \equiv A'\{\!\{\mu(X).B/X\}\!\} \text{ and } X \in \mathsf{FV}(A') \text{ implies } E \vdash_{\mathbf{Q}\varsigma} \mu(X).B)$$

**($\Longleftarrow$)** Rather than using the property above as the inductive hypothesis, we instead show the following is closed under the rules defining type validity:

$$E, X <: \mathsf{Top} \vdash_{\mathbf{Q}\varsigma} A \text{ and } E \vdash_{\mathbf{Q}\varsigma} B \text{ implies } E \vdash_{\mathbf{Q}\varsigma} A\{\!\{B/X\}\!\}$$

The desired result then follows simply by observing that $E \vdash_{\mathbf{Q}\varsigma} \mu(X).B$ must derive from $E, X <: \mathsf{Top} \vdash_{\mathbf{Q}\varsigma} B$, to which the above property may then be applied. □

### 4.5.2 Properties of typed expressions

We are able to establish that typing is preserved by the reorderings and weakenings similar to those given in Lemma 4.3, and omit their definitions here. However, we add an additional operation — that of bound weakening:

**Lemma 4.5** (Weakening of term variable bindings)**.** *The type bound to a term variable in the environment may be replaced by a subtype without impacting on the validity of typing:*

$$E, x : A, E' \vdash_{\mathbf{Q}\varsigma} b : B \textit{ and } E \vdash_{\mathbf{Q}\varsigma} A' <: A \textit{ implies } E, x : A', E' \vdash_{\mathbf{Q}\varsigma} b : B$$

*Proof.* By trivial induction on the rules defining the typing relation. □

We now demonstrate that types are preserved by operations used in the definition of the semantics in Figure 4.12. As a substitution-based semantics, the following property is of particular importance:

**Lemma 4.6** (Types preserved by substitution)**.** *Substitution of a term for a variable, both with matching types, preserves typing:*

$$E, x : B \vdash_{\mathbf{Q}\varsigma} a : A \textit{ and } E \vdash_{\mathbf{Q}\varsigma} b : B \textit{ implies } E \vdash_{\mathbf{Q}\varsigma} a\{\!\{b/x\}\!\} : A$$

*Proof.* The proof follows routinely by ensuring that the following property is closed under the Typ rules of Figure 4.9:

$$E \vdash_{\mathbf{Q}\varsigma} a : A \text{ and } (E \equiv E', x : B \text{ and } E' \vdash_{\mathbf{Q}\varsigma} b : B \text{ implies } E' \vdash_{\mathbf{Q}\varsigma} a\{\!\{b/x\}\!\} : A)$$

□

Our semantics is also predicated upon evaluation contexts to determine evaluation order and to frame the execution of sub-terms within larger terms. We therefore establish the following properties that percolate the properties of typing to sub-terms in contexts:

**Lemma 4.7** (Typing in contexts)**.**

1. *A sub-expression of a well-typed expression is also typable:*

$$E \vdash_{\mathbf{Q}\varsigma} \mathscr{E}[a] : A \textit{ implies } E \vdash_{\mathbf{Q}\varsigma} a : B \textit{ for some } B$$

2. *Replacement of a well-typed sub-expression in a well-typed expression yields a new well-typed expression:*

$$E \vdash_{\mathbf{Q}\varsigma} \mathscr{E}[b] : A \textit{ and } E \vdash_{\mathbf{Q}\varsigma} b : B \textit{ and } E \vdash_{\mathbf{Q}\varsigma} b' : B \textit{ implies } E \vdash_{\mathbf{Q}\varsigma} \mathscr{E}[b'] : A$$

*Proof.* Both properties follow easily by checking closure under the rules defining the typing relation. □

### 4.5.3 Progress

We have already shown that typing respects terms in contexts, and we now establish similar properties for term execution:

**Lemma 4.8** (Execution in contexts)**.** *A term may execute regardless of its context:*

$$(a, \sigma) \rightsquigarrow_{\mathbf{Q}\varsigma} (a', \sigma') \textit{ implies } (\mathscr{E}[a], \sigma) \rightsquigarrow_{\mathbf{Q}\varsigma} (\mathscr{E}[a'], \sigma')$$

*Proof.* By inspection of the rules defining the semantics, it suffices to show that the set of contexts is closed under context composition, that is, $\mathscr{E}[\mathscr{E}'] \equiv \mathscr{E}''$. This clearly holds by definition of context. □

The above lemma then allows us to show that the Progress Theorem holds.

**Theorem 4.9** (Progress)**.**

$$E \vdash_{\mathbf{Q}\varsigma} (a, \sigma) : A \textit{ implies } a \in \mathsf{value} \textit{ or } (a, \sigma) \rightsquigarrow_{\mathbf{Q}\varsigma} (a', \sigma')$$

*Proof.* From $E \vdash_{\mathbf{Q}\varsigma} (a, \sigma) : A$ we obtain, by (TYP CONFIG):

$$E \vdash_{\mathbf{Q}\varsigma} a : A$$
$$\mathsf{dom}(E) = \mathsf{dom}(\sigma)$$
$$E \vdash_{\mathbf{Q}\varsigma} \sigma$$

It then remains to show that, under the above assumptions, the following property is closed under the typing rules:

$$E \vdash_{\mathbf{Q}\varsigma} a : A \text{ and } (a \in \mathsf{value} \text{ or } \exists a', \sigma' : (a, \sigma) \rightsquigarrow_{\mathbf{Q}\varsigma} (a', \sigma'))$$

In general the proof proceeds as follows: For each of the typing rules, we may apply the inductive hypothesis to the antecedents of the rule, at least one of which always corresponds to a sub-term in hole position according to the definition of evaluation contexts.

Where every such antecedent is a value, then a RED rule is always applicable, and an execution step is immediately derived.

Otherwise, the induction hypothesis specifies that such a sub-term is executable and Lemma 4.8 (Execution in contexts) allows us to wrap the appropriate context around it to derive an execution step for the original term. □

### 4.5.4 Determinism of the semantics

We now show, for any well-typed configuration, that the execution step made available according to the Progress Theorem is unique. As a first step, we must show that the decomposition of a term into a context and sub-term of the form found in the conclusion of a reduction rule is unique, but this is routine and clear from inspection of the rules. The remaining source of non-determinism is the selection of a fresh location for object allocation in (RED OBJECT). Such non-determinism is immaterial to the execution of programs (in particular, we do not allow comparison of addresses) so we require only determinism up to renaming of locations:

**Lemma 4.10** (Determinism)**.** *The semantics is deterministic up to the renaming of locations:*

$$E \vdash_{\mathbf{Q}\varsigma} (a, \sigma) : A \textit{ and } (a, \sigma) \rightsquigarrow_{\mathbf{Q}\varsigma} (a', \sigma') \textit{ and } (a, \sigma) \rightsquigarrow_{\mathbf{Q}\varsigma} (a'', \sigma'') \textit{ implies}$$
$$\exists \iota, \iota' \notin \mathsf{dom}(\sigma) : (a', \sigma') \equiv (a'', \sigma'')\{\!\{\iota / \iota'\}\!\}$$

*Proof.* By inspection of the rules. In the case of (RED OBJECT), we need only pick the substitution that aligns the various possible choices of new location. □

### 4.5.5 Type preservation

As the converse to Lemma 4.8, we show that execution of sub-terms in contexts is, in fact, the only way that execution may come about:

**Lemma 4.11.** *If a well-typed term may make an execution step, so may a sub-term:*

$$E \vdash_{\mathbf{Q}\varsigma} (\mathscr{E}[a], \sigma) : A \text{ and } (\mathscr{E}[a], \sigma) \rightsquigarrow_{\mathbf{Q}\varsigma} (a', \sigma') \text{ and}$$
$$a \notin \mathsf{value} \text{ implies } a' \equiv \mathscr{E}[a''] \text{ and } (a, \sigma) \rightsquigarrow_{\mathbf{Q}\varsigma} (a'', \sigma')$$

*Proof.* By inspection of the rules. □

**Lemma 4.12** (Type Preservation)**.** *Types are preserved by execution:*

$$E \vdash_{\mathbf{Q}\varsigma} (a, \sigma) : A \text{ and } (a, \sigma) \rightsquigarrow_{\mathbf{Q}\varsigma} (a', \sigma') \text{ implies } \exists E' : E, E' \vdash_{\mathbf{Q}\varsigma} (a', \sigma') : A$$

*Proof.* We show the property is closed under the typing rules of Figure 4.9. □

Thus, every well-typed term may make a step according to the Progress Theorem, and that step will result in a new well-typed configuration. Therefore, well-typed **Qς** programs will not get stuck, and the guarantees provided by the type system are enforced.

## 4.6 Extensions

In this section, we propose some potential extensions to **Qς**'s basic form of query and give examples of their implementation, either by encoding or by extension of the language.

### 4.6.1 Richer query language

#### Subqueries

The queries permitted by **Qς** do not allow values of labels to be defined by property — for each label, either its value matches exactly that specified in the query, or it is unconstrained. Instead, we can imagine allowing a richer specification of the values that may be permitted. For example, the following query might express those buttons contained in a subwindow of *main*:

$$\langle \mathit{parent} : \mathsf{Window}\ \mathsf{in}\ \langle \mathit{parent} : \mathsf{Window} = \mathit{main} \rangle, \mathit{click} : \mathsf{Top} = ? \rangle$$

This could, of course, be written in **Qς** as:

$$\begin{aligned}&\mathsf{flatten}(\mathsf{map}(\langle \mathit{parent} : \mathsf{Window} = \mathit{main} \rangle,\\ &\qquad \lambda(x : [\mathit{parent} : \mathsf{Window}])\langle \mathit{parent} : \mathsf{Window} = x, \mathit{click} : \mathsf{Top} = ? \rangle))\end{aligned}$$

| Company | |
|---|---|
| **ID** | **Name** |
| 1 | Sprockets Ltd. |
| 2 | General Widgets Inc. |
| 3 | Smith & Sons |

| Invoice | |
|---|---|
| **CompanyID** | **Amount** |
| 1 | 100 |
| 3 | 200 |
| 3 | 300 |

**Company ⋈ Invoice**

| ID | Name | CompanyID | Amount |
|---|---|---|---|
| 1 | Sprockets Ltd. | 1 | 100 |
| 3 | Smith & Sons | 3 | 200 |
| 3 | Smith & Sons | 3 | 300 |

Figure 4.14: Example of a database join

Where flatten flattens members of list of list of *A* to list of *A* and map applies the function given as its second argument to successive elements of its first argument, returning a list of the results. Their definitions use an object to provide the recursion in their standard implementations, which are omitted here: see Figure 5.8 on page 130 for an example.

### Joins

Joins — where objects from two collections are paired according to some property — are common in database queries. For example, where we have a set of Company tuples and a set of Invoice tuples, where each Company has an ID and each Invoice has a CompanyID to indicate who was billed, we might join the tuples by matching the two identifiers, which we write using the ⋈ operator familiar from relational algebra [24][5]:

$$\mathsf{Company} \overset{\mathsf{ID=CompanyID}}{\bowtie} \mathsf{Invoice}$$

The result is a set of tuples containing the properties of both the Company and the Invoice, wherever a match was possible, as shown in Figure 4.14. Given the join condition, at least one of the ID/CompanyID columns is redundant, and would normally be projected out from the result.

We demonstrate how joins may be useful in a programming language by giving an example from a Graphical User Interface (GUI):

During execution, a GUI generates events. In this example, each occurrence of such an event gives rise to an object storing the event type, the target of the event and a flag to indicate whether the event has been handled by the system.

[5] Whilst Codd [24] used $*$ instead of the ⋈ operator, it is now standard

For example:

$$[target = \varsigma(x : \mathsf{Event})myButton, \ type = clickEvent, \ handled = no]$$

$$\mathsf{Event} \equiv [target : \mathsf{Window}, type : \mathsf{EventType}, handled : \mathsf{Bool}]$$

Suppose that a *behaviour* defines a set of actions associated with various system events such as a button click, or the minimization of a window. When such an event occurs, the behaviour's *handleEvent* label is invoked with the type of the event and the target as parameters. Thus, a behaviour takes the form:

$$[handleEvent = \varsigma(x : \mathsf{Behaviour})\lambda(target : \mathsf{Window})\lambda(type : \mathsf{EventType})\ldots]$$

It is possible that a button may have several behaviours, and buttons may share the same behaviour — they are in a many-to-many relationship. We associate a behaviour *bhv* with a button *btn* by relating the two in an object:

$$[button = \varsigma(x : [button : \mathsf{Button}, behaviour : \mathsf{Behaviour}])btn, \ behaviour = bhv]$$

For any given button, $b$, we can obtain its behaviours by posing the query:

$$\langle button : \mathsf{Button} = b, \ behaviour : \mathsf{Behaviour} = ?\rangle$$

and we can obtain its unhandled events with:

$$\langle target : \mathsf{Window} = b, \ type : \mathsf{EventType} = ?, \ handled : \mathsf{Bool} = no\rangle$$

With a join, however, we can obtain in one place the details of outstanding events as well as the behaviour responsible for handling them.

$$\langle behaviour : \mathsf{Behaviour} = ?\rangle \overset{button=target}{\bowtie} \langle target : \mathsf{Button} = ?, \ type : \mathsf{EventType} = ?, \ handled : \mathsf{Bool} = no\rangle$$

During execution of such a query, a database system would construct a set (or bag) of tuples to represent the result — in a database programming language, it is usual to manipulate tuples which do not exist in a table. In an object-oriented programming language, however, it is not usual to deal with data that does not exist in the program store. Broadly speaking, there are two possibilities for the result of such a query in **Qς**:

**List of objects** Just as for a query without join, we may return a list of objects as the result, where each object has type:

$$[behaviour : \mathsf{Behaviour}, \ target : \mathsf{Button}, \ type : \mathsf{EventType}, \ handled : \mathsf{Bool}]$$

However, unlike the forms of query previously presented, the result would

include objects that would not have existed before the query was executed. Nowhere have we defined objects that include labels returning behaviours alongside labels returning buttons. In this case, observing the heap modifies its state: executing the same query twice, even without any object updates in-between, may yield different results for each execution.

**List of tuples** **Q**ς could be extended with records, that could store tuples of type:

$$[\mathit{behaviour} : \mathsf{Behaviour}] \times [\mathit{target} : \mathsf{Button}, \mathit{type} : \mathsf{EventType}, \mathit{handled} : \mathsf{Bool}]$$

In common with databases, these are not objects and are not reflected in the program store.

Both options carry advantages, and both can — we believe — be constructed soundly.

### 4.6.2 Zoned query

When a query is executed, every object in the heap is considered. Instead suppose that we allow objects to be created in *zones*: effectively ring-fenced areas of the heap in which queries may operate. For example, buttons are constructed in the a zone of the heap dedicated to the graphical user interface:

$$[\mathit{click} = (\text{invoke click handler})] \text{ in } \mathit{GUIZone}$$

whilst elsewhere, an object may be created that has a *click* method, but which represents part of a sound system:

$$[\mathit{click} = \mathit{speaker}.\mathit{emit}(1\text{kHz for 2ms})] \text{ in } \mathit{SoundSystemZone}$$

Thus, the following queries would have disjoint results:

$$\mathit{GUIZone}.\langle \mathit{click} : \mathsf{Top} = ?\rangle \qquad \mathit{SoundZone}.\langle \mathit{click} : \mathsf{Top} = ?\rangle$$

and yet their union would represent the result of the $\langle \mathit{click} : \mathsf{Top} = ?\rangle$ query without any zone constraints.

Applications of such a mechanism are various, but in particular it can help us mitigate the effect of objects created by joins, as discussed above. Objects created in this way can be created in an ‘unqueryable’ zone, such that they do not affect other queries. Additionally, without the facility to destroy objects, which gives rise to the difficulties surrounding dangling pointers, we can move objects that no longer represent current relationships into similarly unqueryable zones.

We can implement zones in **Q**ς simply by adding an extra field to zoned objects and to zoned queries. For appropriate choices of a zone type, Zone[6] and

[6] This can reasonably be Top, as we do not expect to perform any operations on zones.

for a label to represent an object's zone, *zone*, we can encode the constructions above thus:

$$[l_i : A_i = m_i] \text{ in } Z \quad \text{translates to} \quad [l_i : A_i = m_i, zone : \mathsf{Zone} = Z]$$
$$Z.\langle l_i : A_i = v_i, l_j : A_j = ?\rangle \quad \text{translates to} \quad \langle l_i : A_i = m_i, zone : \mathsf{Zone} = Z\rangle$$

## 4.7 Conclusion

In this chapter we have introduced **Qς**: a formal treatment of an object calculus with the ability to query the heap. We have demonstrated a type system and semantics, and shown that they are sound. Furthermore, by investigating query in the setting of an object calculus, we have been able to explore typing issues which may not otherwise have become apparent with a nominal type-system.

In the presence of query, relationships once again become a matter of inter-object references. Instead of the complex structures of Section 1.2, however, heap query is able to recover relationships from a single reference.

In the following chapter, we proceed to translate RelJ, our model for the inclusion of relationships in a class-based language, into **Qς**.

CHAPTER 5

# Bridging the gap

In the preceding chapters we have given a class-based calculus and an object-based calculus, both of which have facilities for managing relationships between objects. In this chapter, we give a translation from RelJ to **Q**ς in order to demonstrate that **Q**ς's query mechanism allows the expression of RelJ-style relationships. Much of the work involved in such a translation relates to the provision of class-based features in an object-based language; once this is complete, the translation of the relationship-based portions of RelJ is reasonably straightforward.

## 5.1 Translation overview

Our translation is heavily based upon that given by Abadi and Cardelli, which translated a class-based (but not Java-like) language into the object-based calculus upon which **Q**ς is based [1]. However, their source language lacks a number of features present in RelJ, so the translation given here is somewhat more complicated in order to accommodate mutually-referencing classes, null pointers and errors. In particular, and unlike the Abadi-Cardelli translation, the translation given here is type-directed, following the derivation of the translated terms' RelJ type.

We do not give translations to all of RelJ's features: equality-testing and set manipulation are omitted as their translations are rather cumbersome and do not bear any relevance to our discussion.[1] We also omit local variables and covariant return types for methods: the former may be encoded using fields and the latter can be easily recovered using the technique discussed in Section 5.1.3 for retaining the covariance of RelJ's `from` and `to` fields during translation.

[1] Equality testing involves extending **Q**ς with access to an object's identifier, or the addition of serial numbers to objects which, in turn, requires the provision of naturals. Set manipulation requires equality testing so that, when iterating over a set to remove an object, we can determine whether the current object is that which should be removed. Neither of these is relevant to the distinction between class-based and object-based languages, nor to the provision of relationships.

### 5.1.1 Class factories

Classes are encoded as factories, just as in the Abadi-Cardelli translation. A factory for class $c$ has a *new* label which, when invoked, returns an object that represents an instance of $c$. Therefore, a factory for $c$ will have the following form:

$$[\mathit{new} = \varsigma(x : \mathit{FactoryType})[\ldots \text{representation of new } c\text{-instance} \ldots], \ldots]$$

where *FactoryType* is of the form below, in which $[\![c]\!]$ represents some appropriate translation of $c$ into a $\mathbf{Q}\varsigma$ type as discussed later in Section 5.3:

$$\mathit{FactoryType} \equiv [\mathit{new} : [\![c]\!], \ldots]$$

In the Abadi-Cardelli translation, each factory is let-bound in the program. For example:

$$\begin{array}{l} \mathsf{let}\, \mathit{factory}_c : A = [\mathit{new} = \varsigma(x : A)a, \ldots]\ \mathsf{in} \\ \quad \mathsf{let}\, \mathit{factory}_d : B = [\mathit{new} = \varsigma(x : B)b, \ldots]\ \mathsf{in} \\ \quad\quad \ldots \end{array}$$

In this encoding, however, $\mathit{factory}_c$ may not refer to $\mathit{factory}_d$ and is therefore unable to create instances of $d$ by invoking $\mathit{factory}_d.\mathit{new}$. RelJ *does* allow mutually-instantiating classes, however, so we collect all such factories in a single object where self-application provides the necessary recursion to permit mutual-instantiation:

$$\begin{array}{l} \mathsf{let}\ \mathit{allFactories} : \mathit{FactoryType} = \\ \quad [c = \varsigma(x : \mathit{FactoryType}) \ldots \text{factory for } c \ldots \\ \quad\ \ d = \varsigma(x : \mathit{FactoryType}) \ldots \text{factory for } d \ldots]\ \mathsf{in} \\ \ldots \end{array}$$

Therefore, invoking $\mathit{allFactories}.c$ yields a $c$-factory, so $\mathit{allFactories}.c.\mathit{new}$ will return a new $c$-instance. The $c$-factory obtained in this way can use the self-parameter $x$ to generate $d$-instances by invoking $x.d.\mathit{new}$.

### 5.1.2 Pre-methods

The objects returned by the $c$ factory's *new* method will have labels corresponding to each of $c$'s fields and methods. The fields will be rendered in the obvious way, as labels with values. For the translation of methods, we use the pre-method mechanism familiar from the Abadi-Cardelli translation.

A pre-method is a template for a $c$-instance's method, where the identity of the object — the value of `this` in RelJ — is a parameter. This parameter is instantiated at object construction time. For example, the following RelJ method

declaration in class $c$:

```
A m(B y) { return y.f }
```

will be translated into the pre-method $pre_{\mathtt{m}}$ in $c$'s factory, which is now of the following form:

$$[pre_{\mathtt{m}} = \varsigma(cFac : FactoryType)\lambda(x_{\mathtt{this}} : [\![c]\!])\lambda(y : [\![\mathtt{B}]\!])y.f \\ new = \varsigma(cFac : FactoryType)\ldots]$$

In subclasses of $c$, this method m is inherited, so factories for those subclasses may simply refer to the pre-method from $c$'s factory. Therefore $d$'s factory, where $d$ `extends` $c$ will have the form:

$$[pre_{\mathtt{m}} = \varsigma(cFac : FactoryType)allFactories.c.pre_{\mathtt{m}}, \\ new = \varsigma(cFac : FactoryType)\ldots]$$

When the time comes to construct a new $c$-instance and a factory's *new* label is invoked, it remains to resolve the chain of selections for inherited methods, to create an object with labels for each of $c$'s fields and methods, and to instantiate the new object's pre-methods to methods by applying them to the new object's self-parameter:

$$\begin{aligned} new = \varsigma(cFac : FactoryType)\text{let } & w_{\mathtt{m}} : PreMethType = cFac.pre_{\mathtt{m}} \\ \text{in } & [f = \varsigma(z : [\![c]\!])\text{initial value for } f, \\ & \ m = \varsigma(z : [\![c]\!])w_{\mathtt{m}}(z), \\ & \ \ldots] \end{aligned}$$

### 5.1.3 Relationships

So far we have only discussed the translation of classes, but the same strategy is used when translating relationships into **Q**$\varsigma$. This strategy need only be extended so that the `to` and `from` pseudo-fields are represented in the translation, as well as the concept of relationship instance activity — whether or not the relationship instance is 'current'.

First, the *new* method of a relationship factory becomes an abstraction with two parameters to represent the source and destination of the new relationship instance:

$$new = \varsigma(rFac : FactoryType)\lambda(x_1 : [\![n_1]\!])\lambda(x_2 : [\![n_2]\!])\ldots$$

Where $n_1$ and $n_2$ are $r$'s source and destination types, respectively.

The source and destination instances cannot be stored in single fields, as **Q**$\varsigma$ cannot express our desire to give up mutability of those fields in exchange for covariance. Instead, we assign them to a family of fields: a $to_{r'}$ field and a $from_{r'}$ field for every $r'$ that is a supertype of $r$ ($r$ included). Each $from_{r'}$ field is assigned

the source type from the declaration of $r'$ so that the type is as precise as possible; similarly each $to_{r'}$ field. For example, consider the following two relationships:

```
relationship R1 extends Relation (A, B) { ... }
relationship R2 extends R1 (C, D) { ... }
```

where $\vdash_{\mathbf{R}} \mathtt{C} \leq \mathtt{A}$ and $\vdash_{\mathbf{R}} \mathtt{D} \leq \mathtt{B}$. Instances of these two relationships have the following forms, with types:

$$
\begin{array}{rl}
\text{R1-instance:} & [\mathit{from}_{\mathrm{R1}} = \iota_1,\ \mathit{to}_{\mathrm{R1}} = \iota_2, \ldots] \\
& [\mathit{from}_{\mathrm{R1}} : [\![\mathtt{A}]\!], \mathtt{to}_{\mathrm{R1}} : [\![\mathtt{B}]\!]] \\
\text{R2-instance:} & [\mathit{from}_{\mathrm{R1}} = \iota_1,\ \mathit{to}_{\mathrm{R1}} = \iota_2, \mathit{from}_{\mathrm{R2}} = \iota_1,\ \mathit{to}_{\mathrm{R2}} = \iota_2] \\
& [\mathit{from}_{\mathrm{R1}} : [\![\mathtt{A}]\!], \mathtt{to}_{\mathrm{R1}} : [\![\mathtt{B}]\!], \mathit{from}_{\mathrm{R2}} : [\![\mathtt{C}]\!], \mathit{to}_{\mathrm{R2}} : [\![\mathtt{D}]\!]]
\end{array}
$$

Suppose now that C declares a method not available in A. If `from` were modelled with only one label, *from*, then its type would either have to be different between the two instances, in which case the translation of R1 would not be a supertype of R2, or it would have to be fixed at $[\![\mathtt{A}]\!]$, in which case C's method would not be available when accessing `from` on instances of R2. By using a label for each of the types at which the instance may be viewed — for R2-instances this includes R2, R1 and `Relation` — the declared participant types are preserved. Accesses to `to` or `from` are translated to the selection of the appropriate label according to the static type of the receiver, which is sufficient for ensuring that the expected fields and methods are available.

Immutability of the *from* and *to* labels is not assured by the $\mathbf{Q}\varsigma$ type system, but well-typed RelJ programs will not contain assignments to `to` or `from`, so we may expect the translation to be similarly free of such mutations.

Finally, we equip objects representing relationship instances with an *extent* field, which we use in a manner familiar from Section 4.6.2 on zoned queries. This *extent* label holds a reference to one of several global objects: $\mathit{extent}_r$ for each relationship $r$, and *none*. An object's *extent* label indicates both the dynamic RelJ type of the object, and whether or not the object is 'active':

- When an instance's *extent* field references $\mathit{extent}_r$, this indicates that it is an active instance of *precisely* type $r$. In RelJ terms, the instance would be in $\rho$, and its dynamic type would be $r$.

- When *extent* references *none*, then the instance is no longer active — in RelJ terms, it is no longer in $\rho$.

As we shall see, we only require these two states to be distinguishable by heap query, so only the identity of these *extent objects* is important. We shall discuss the use of the *extent* field when reviewing the implementation of RelJ's relationship manipulation and access operators in Section 5.3.

### 5.1.4 Null pointers and errors

**Q**ς has no concept of `null`, which must be simulated in any translation from RelJ programs to **Q**ς programs. The particular difficulty when representing `null` is that it occupies every reference type in RelJ, yet there is no such universal value in **Q**ς.

Instead, we create distinguished objects to represent a `null` value of each RelJ reference type. The `null` representative for type $c$ clearly must have same type as those objects representing instances of $c$. We determine whether an object represents a genuine instance of $c$, or whether it represents `null`, by extending **Q**ς with booleans and conditionals so that we may equip each instance with a flag, *isNull*, that we may test before subjecting the object to any field accesses or method invocations. Where the flag is set, indicating that the target instance represents `null`, we typically return a value representing a `null`-pointer error. For example, `x.f` may be translated to:

$$\mathsf{if}\ x.\mathit{isNull}\ \mathsf{then}\ \text{null-pointer error}\ \mathsf{else}\ x.f$$

In the RelJ semantics, a `null`-pointer error is immediately propagated to the top level. In the translation, we use a similar technique as that used to emulate `null`: we build objects that represent the error for all types (not just for reference types), and then use a continuation-passing style to check sub-terms' results for error before proceeding with execution of the larger term that uses those results. Whereas we only sought `null` when translating field lookup and method invocation, a `null`-pointer error may need to be propagated upwards through any RelJ term. Values are therefore checked for error wherever they are used: for example, in field accesses, in method invocations as receiver, parameter or return value, in relationship manipulations, and so on; even the translations of statements yield a return value, so that this may be checked for error. To accomplish this, all objects representing instances of RelJ reference types (or `null` for those types) are equipped with a *isNPE* flag, which functions in the same way as the *isNull* flag described previously. So that errors generated in statements may be propagated, a representative of `void` is also created, which contains only an *isNPE* label. An error from an expression returning a value of type `set`<*n*> is expressed as a list with *n*'s error representation at the head.

The details of checking intermediate values for error are similar to the checks conducted for `null`, so we leave the details for the full translation in the Section 5.3. However, it should be noted here that the type-direction of the translation results from the need to represent `null` and `null`-pointer errors: when translating an expression that may cause an error, we require the expression's RelJ type so that the correct error value may be selected.

### 5.1.5 Relationship manipulation

The translation of RelJ's class-based and imperative features is reasonably obvious, given the environment established above: field access and update are translated to label selection and update respectively; method invocation to selection and application; object allocation with new is translated to invocation of the appropriate factory's *new* method.

Of greater relevance is the implementation of RelJ's relationship manipulation and access operators. The relationship access operator $\iota\mathtt{:}r$, which yields the set of $r$-objects relating $\iota$ to some other object, is immediately translated to a heap query. The resulting query fixes the values of the $\mathit{from}_{r'}$ labels to $\iota$ for all supertypes $r'$ of $r$, and fixes the *extent* label to the $\mathit{extent}_r$ object. The *to* labels are not fixed. Thus, all active representatives of $r$ are selected which relate $\iota$ to some other object. So that the resulting objects have a type through which $r$'s field and methods may be accessed, the query also mentions field and method labels, but their values are not fixed.

The translation of $\iota\mathtt{.}r$ — the set of objects related to $\iota$ through $r$ — simply maps the selection of the $\mathit{to}_r$ label over the result of the translation of $\iota\mathtt{:}r$.

Relationship addition is translated to an appropriate call to the relationship factory's *new* method, whilst relationship removal uses a query similar to that used in relationship access in order to obtain all relevant instances of the relationship. An update of each instance's *extent* label to *none* is then mapped across the result so that the instances no longer appear in the result of the query used to access relationships.

## 5.2 Qς booleans and conditionals

We require conditional branches so that null values and error values may be detected. We therefore assume a boolean type which includes the two boolean values, true and false, and a conditional test. These may be included in **Qς** in two ways: by extending the definition of **Qς**, or by encoding.

An extension of **Qς**'s type system and semantics is shown in Figure 5.1. The extension is largely self-explanatory: a new type is introduced into the grammar and the valid-type relation; no adjustment is required to the subtyping relation, as it is reflexive by definition; the true and false values are always well-typed and an if takes one boolean term for the test and two terms of the same type for the continuation. The semantics of if is standard.

Alternatively, booleans may be encoded, as long there is a finite universe of types, $\mathscr{A}$, at which conditionals will be used; that is, the continuations of all uses of if have type $A \in \mathscr{A}$. The boolean values, true and false, are represented by objects that have a family of continuation-selecting methods, $\mathit{test}_A$, one for each type at which the test might be required. Each method accepts an object with a *true*- and a *false*-continuation, and selects one depending on the truth value represented by the enclosing object. The encoding of if simply packages the con-

$$A, B ::= \ldots \mid \mathsf{boolean} \qquad \text{extended type grammar}$$
$$a, b ::= \ldots \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ a\ \mathsf{then}\ b\ \mathsf{else}\ b' \qquad \text{extended term grammar}$$
$$\mathscr{E} ::= \ldots \mid \mathsf{if}\ \mathscr{E}\ \mathsf{then}\ a\ \mathsf{else}\ b \qquad \text{extended context grammar}$$

(IsTypBool) (Typ True) (Typ False)

$$\frac{\vdash_{\mathbf{Q}\varsigma} E\ \mathsf{ok}}{E \vdash_{\mathbf{Q}\varsigma} \mathsf{boolean}} \qquad \frac{}{E \vdash_{\mathbf{Q}\varsigma} \mathsf{true} : \mathsf{boolean}} \qquad \frac{}{E \vdash_{\mathbf{Q}\varsigma} \mathsf{false} : \mathsf{boolean}}$$

(Typ Cond)

$$\frac{E \vdash_{\mathbf{Q}\varsigma} a : \mathsf{boolean} \qquad E \vdash_{\mathbf{Q}\varsigma} b : B \qquad E \vdash_{\mathbf{Q}\varsigma} b' : B}{E \vdash_{\mathbf{Q}\varsigma} \mathsf{if}\ a\ \mathsf{then}\ b\ \mathsf{else}\ b' : B}$$

(Red Cond True)

$$\frac{}{(\mathscr{E}[\mathsf{if\ true\ then}\ a\ \mathsf{else}\ b], \sigma) \rightsquigarrow_{\mathbf{Q}\varsigma} (\mathscr{E}[a], \sigma)}$$

(Red Cond False)

$$\frac{}{(\mathscr{E}[\mathsf{if\ false\ then}\ a\ \mathsf{else}\ b], \sigma) \rightsquigarrow_{\mathbf{Q}\varsigma} (\mathscr{E}[b], \sigma)}$$

Figure 5.1: Extended grammar and rules for booleans and conditionals in **Q**ς

$$\begin{aligned}
\mathsf{boolean} &\mathrel{\widehat{=}} [\mathit{test}_A : \mathsf{Choice}_A \to A \quad {}^{A \in \mathscr{A}}] \\
\mathsf{true} &\mathrel{\widehat{=}} [\mathit{test}_A = \varsigma(x : \mathsf{boolean})\lambda(y : \mathsf{Choice}_A)y.\mathit{true} \quad {}^{A \in \mathscr{A}}] \\
\mathsf{false} &\mathrel{\widehat{=}} [\mathit{test}_A = \varsigma(x : \mathsf{boolean})\lambda(y : \mathsf{Choice}_A)y.\mathit{false} \quad {}^{A \in \mathscr{A}}] \\
\mathsf{if}_A\ a\ \mathsf{then}\ b\ \mathsf{else}\ b' &\mathrel{\widehat{=}} a.\mathit{test}_A([\mathit{true} = \varsigma(y : \mathsf{Choice}_B)b, \\
&\qquad\qquad \mathit{false} = \varsigma(y : \mathsf{Choice}_B)b'])
\end{aligned}$$

where $\mathsf{Choice}_B \equiv [\mathit{true} : B, \mathit{false} : B]$ and $y$ does not occur in $b, b'$

Figure 5.2: Encoding of booleans and conditionals in **Q**ς

tinuations in an object and passes it to the appropriate *test* method of the boolean value to be tested. We must annotate if, written $\text{if}_A$, with the type expected of the continuations so that the correct *test* method is used — in practice, this would be eliminated by type-directing the translation of **Qς** programs with booleans.

As we shall use booleans to flag and detect null and errors, the universe of types will be the set of translations of RelJ types from the program under consideration (see the following section):

$$\mathscr{A} \equiv \{ [\![ t ]\!] \mid t \in \mathscr{T} \}$$

We are therefore free choose either strategy for acquiring booleans in **Qς**: we assume their presence from here on.

## 5.3 Formalization of the translation

For the interested reader, we now give the translation of RelJ into **Qς** in full detail. As discussed in the previous section, it is convenient to regard `void` as a RelJ type for the purposes of the translation. For this chapter, therefore, we assume `void` $\in \mathscr{T}$, the set of types in a RelJ program.

### 5.3.1 Encoding RelJ types

In the previous section, we assumed a translation from RelJ types to **Qς** types, denoted $[\![-]\!]$. In fact, the situation is more complicated than previously suggested: the nominal type system in RelJ allows us to avoid dealing explicitly with recursion; for example, where classes hold references to themselves. Therefore, rather than encoding a class $c$ simply as an object type with labels for $c$'s fields and methods, we must encode it as a recursive object type, $\mu(X).[\ldots]$, where the object uses $X$ to type those fields and methods that refer to $c$ recursively.

We denote the translation $[\![-]\!]^{\phi}$, where $\phi$ is a mapping from class and relationship names to type variables. $\phi$ indicates which classes we have already 'seen' whilst translating the type, so that the type variable may be used in place of a (possibly infinite) repetition of the whole translated type. We write $[\![-]\!]$ when $\phi$ is empty.

Figure 5.3 defines the translation for RelJ types. For reference types $n$, if $n$ has been assigned a type variable in $\phi$, then that variable is used as the translation for the type. Otherwise, a fresh type variable is bound and wraps an object which contains labels for fields, methods, `null`- and error-checking and, in the case of relationships, labels for the source, destination and activity flag for the relationship instance. `set<−>` types are translated to **Qς** lists, and the `void` type is an object with labels for trapping error conditions.

The recursive types resulting from the translation of classes and relationships each give rise to a family of isomorphic types, accessed by type unfolding. Where we wish to make the fields and methods available for access or invocation, the

$$[\![n]\!]^{\phi} \mathrel{\hat{=}} \begin{cases} \phi(c) & \text{if } n \in \mathsf{dom}(\phi) \\ A & \text{if } n \in \mathsf{dom}(\mathscr{C}) \\ B & \text{if } n \in \mathsf{dom}(\mathscr{R}) \end{cases}$$

where, for fresh $X \notin \mathsf{rng}(\phi)$ and $\phi' = \phi \cup (n \mapsto X)$:

$$\begin{aligned} A \equiv \mu(X).[\, & f : [\![\mathscr{F}_n^+(f)]\!]^{\phi'}, \quad {}^{f \in \mathsf{dom}(\mathscr{F}_n^+)} \\ & m : [\![\mathscr{M}_n^+(m)_{\mathsf{arg}}]\!]^{\phi'} \to [\![\mathscr{M}_n^+(m)_{\mathsf{ret}}]\!]^{\phi'}, \quad {}^{m \in \mathsf{dom}(\mathscr{M}_n^+)} \\ & \mathit{isNull} : \mathsf{boolean}, \\ & \mathit{isNPE} : \mathsf{boolean}] \end{aligned}$$

$$\begin{aligned} B \equiv \mu(X).[\, & \mathit{extent} : \mathsf{Top}, \\ & \mathit{from}_r : [\![\mathscr{R}(r)_{\mathsf{from}}]\!]^{\phi'}, \quad {}^{r \text{ s.t. } \vdash_{\mathbf{R}} n \leq r} \\ & \mathit{to}_r : [\![\mathscr{R}(r)_{\mathsf{to}}]\!]^{\phi'}, \quad {}^{r \text{ s.t. } \vdash_{\mathbf{R}} n \leq r} \\ & f : [\![\mathscr{F}_n^+(f)]\!]^{\phi'}, \quad {}^{f \in \mathsf{dom}(\mathscr{F}_n^+)} \\ & m : [\![\mathscr{M}_n^+(m)_{\mathsf{arg}}]\!]^{\phi'} \to [\![\mathscr{M}_n^+(m)_{\mathsf{ret}}]\!]^{\phi'}, \quad {}^{m \in \mathsf{dom}(\mathscr{M}_n^+)} \\ & \mathit{isNull} : \mathsf{boolean}, \\ & \mathit{isNPE} : \mathsf{boolean}] \end{aligned}$$

and, for non-reference types:

$$\begin{aligned} [\![\texttt{set<}n\texttt{>}]\!]^{\phi} &\mathrel{\hat{=}} \mathsf{list\ of}\ [\![n]\!]^{\phi} \\ [\![\texttt{void}]\!]^{\phi} &\mathrel{\hat{=}} [\mathit{isNPE} : \mathsf{boolean}] \end{aligned}$$

Figure 5.3: Translation of RelJ types to **Q**ς types

type must be unfolded. We write $[\![t]\!]^{\circlearrowleft}$ for the unfolded translation of a type $t$:

$$A^{\circlearrowleft} \mathrel{\hat{=}} \begin{cases} A\{\!\!\{\mu(X).A/X\}\!\!\} & \text{where } A \equiv \mu(X).A \\ A & \text{otherwise} \end{cases}$$

The unfolding of $\texttt{set}<->$ and $\texttt{void}$ types has no effect.

We follow a simple convention for determining the type at which values and terms are viewed: values are always passed and held in fields and variables at their folded type, and are only unfolded for field access or method invocation. This convention is usually enforced easily, although as queries in **Q**ς are typed $\mathsf{list\ of}\ [-]$ rather than $\mathsf{list\ of}\ \mu(X).[-]$, some work is required when translating relationship access.

Finally, we translate typing environments by mapping the translation of types

$$
\begin{aligned}
\mathsf{null}_c \mathrel{\widehat{=}} \mathsf{fold}(\llbracket c \rrbracket, [&l = \varsigma(z : \llbracket c \rrbracket^{\circlearrowleft})z.l \;{}^{l \in \mathsf{dom}(\mathscr{F}_c^+) \cup \mathsf{dom}(\mathscr{M}_c^+)}, \\
&\mathit{isNull} = \mathsf{true} \\
&\mathit{isNPE} = \mathsf{false}]) \\
\mathsf{null}_r \mathrel{\widehat{=}} \mathsf{fold}(\llbracket r \rrbracket, [&l = \varsigma(z : \llbracket r \rrbracket^{\circlearrowleft})z.l \;{}^{l \in \mathsf{dom}(\mathscr{F}_r^+) \cup \mathsf{dom}(\mathscr{M}_r^+)}, \\
&l' = \varsigma(z : \llbracket r \rrbracket^{\circlearrowleft})z.l' \;{}^{l' \in \{\mathit{from}_{r'}, \mathit{to}_{r'} \mid \vdash_{\mathbf{R}} r \leq r'\}}, \\
&\mathit{extent} = \mathit{none}, \\
&\mathit{isNull} = \mathsf{true} \\
&\mathit{isNPE} = \mathsf{false}])
\end{aligned}
$$

Figure 5.4: **Q**ς terms which act as factories of `null` values for RelJ reference types

across the environment's assigned types — variables are not modified:

$$\llbracket \Gamma \rrbracket(x) = \llbracket t \rrbracket \text{ where } \Gamma(x) = t$$

### 5.3.2 Translation of terms

We now proceed to set up a translation relation from well-typed RelJ expressions and statements to **Q**ς terms. When expression $e$ (or statement $s$), which is well typed in the presence of environment $\Gamma$, may be translated to **Q**ς term $a$ we write:

$$\Gamma \vdash_{\mathbf{R}} e : t \rhd a \qquad\qquad \Gamma \vdash_{\mathbf{R}} s \rhd a$$

Before we can define this translation, however, we must establish encodings for `null` and for errors, and a strategy for detecting and propagating such errors.

#### `null` values

Figure 5.4 gives the definitions for `null`-encodings for each RelJ reference type, $n$. So that the `null` encoding for $n$ inhabits the same type as the encoding of non-`null` $n$-instances, these encodings have labels to represent $n$'s fields and methods and, in the case of relationships, labels for source and destination pointers as discussed earlier. These labels all take method bodies of the form:

$$\varsigma(x : \llbracket n \rrbracket^{\circlearrowleft})x.l$$

Such labels will infinitely recurse if selected, but our translation will raise an error before such a selection is made on an object representing `null`. Notice that the methods' self parameters are annotated with the unfolded form of $n$'s type, as labels may not be selected from a folded term. Finally, these objects have their *isNull* flags set to true. RelJ permits the passage of `null` through the program, as long as no field access or method invocations are performed upon it: the `null` representatives' error flags are therefore set to false.

$$\begin{aligned}
\mathsf{npe}_c \mathrel{\widehat{=}} \mathsf{fold}(\llbracket c \rrbracket, [& l = \varsigma(z : \llbracket c \rrbracket^{\circlearrowleft})z.l^{\ l \in \mathsf{dom}(\mathscr{F}_c^+) \cup \mathsf{dom}(\mathscr{M}_c^+)}, \\
& \mathit{isNull} = \mathsf{false} \\
& \mathit{isNPE} = \mathsf{true}]) \\
\mathsf{npe}_r \mathrel{\widehat{=}} \mathsf{fold}(\llbracket r \rrbracket, [& l = \varsigma(z : \llbracket r \rrbracket^{\circlearrowleft})z.l^{\ l \in \mathsf{dom}(\mathscr{F}_r^+) \cup \mathsf{dom}(\mathscr{M}_r^+)}, \\
& l' = \varsigma(z : \llbracket r \rrbracket^{\circlearrowleft})z.l'^{\ l' \in \{from_{r'}, to_{r'} \mid \vdash_{\mathbf{R}} r \leq r'\}}, \\
& \mathit{extent} = \mathit{none}, \\
& \mathit{isNull} = \mathsf{false} \\
& \mathit{isNPE} = \mathsf{true}])
\end{aligned}$$

$$\begin{aligned}
\mathsf{npe}_{\mathtt{set<}n\mathtt{>}} &\mathrel{\widehat{=}} \mathsf{cons}(\mathsf{npe}_n, \mathsf{nil}) \\
\mathsf{npe}_{\mathtt{void}} &\mathrel{\widehat{=}} [\mathit{isNPE}_t = \mathsf{true}]
\end{aligned}$$

Figure 5.5: **Q**ς terms to construct error values for RelJ types $t \in \mathscr{T}$

The following proposition specifies that $n$'s `null` representative is well-typed in the **Q**ς type system:

**Proposition 5.1.** *For a well-formed program, $(\mathscr{C}, \mathscr{R}, s)$, and for every reference type $n \in dom(\mathscr{C}) \cup dom(\mathscr{R})$:*

$$\emptyset \vdash_{\mathbf{Q}\varsigma} null_n : \llbracket n \rrbracket$$

Thus, we have a `null` representative for every reference type in the translated RelJ program, just as `null` can be typed with any such type in the RelJ type system.

### void value

For the encoding of statements, we define a term that generates a `void` value. This is simply an object containing a label that allows checking for error conditions. The `void` value's error flag is set to false, indicating that the statement from which it originates did not attempt to dereference `null`:

$$\mathsf{void} \mathrel{\widehat{=}} [\mathit{isNPE} = \mathsf{false}]$$

Naturally, we require that this representation of a `void` value inhabits the **Q**ς equivalent of the `void` type:

**Proposition 5.2.**

$$\emptyset \vdash_{\mathbf{Q}\varsigma} void : \llbracket \mathtt{void} \rrbracket$$

### Errors

Just as for `null`, we have encodings for `null`-pointer errors, given in Figure 5.5. However, we have a representative for every RelJ type, including `void` so that we

$$\text{ckNPE } x : n = a \text{ at } t' \text{ in } b \mathrel{\widehat{=}} \text{let } x : [\![n]\!] = a \text{ in if unfold}(x).\mathit{isNPE} \text{ then npe}_{t'} \text{ else } b$$

$$\text{ckNPE } x : \texttt{set<}n\texttt{>} = a \text{ at } t' \text{ in } b \mathrel{\widehat{=}} \text{let } x : [\![\texttt{set<}n\texttt{>}]\!] = a \text{ in match } x : [\![\texttt{set<}n\texttt{>}]\!] \text{ with } \text{cons}(y, ys) \Rightarrow \text{if unfold}(y).\mathit{isNPE} \text{ then npe}_{t'} \text{ else } b \mid \text{nil} \Rightarrow b$$

$$\text{ckNPE } x : \texttt{void} = a \text{ at } t' \text{ in } b \mathrel{\widehat{=}} \text{if } a.\mathit{isNPE} \text{ then npe}_{t'} \text{ else } b$$

$$\text{ckNull } x : n = a \text{ at } t' \text{ in } b \mathrel{\widehat{=}} \text{ckNPE } x : n = a \text{ at } t' \text{ in if unfold}(x).\mathit{isNull} \text{ then npe}_{t'} \text{ else } b$$

Figure 5.6: Constructs to check for `null` values and error conditions, where $y$ and $ys$ are fresh in the continuation $b$

may check for error conditions resulting from statements. For reference types, the principle is the same as for `null` encodings. Errors arising from expressions returning a value of type `set<n>` are encoded simply by wrapping $n$'s error representative in a singleton list.

**Proposition 5.3.** *For a well-formed program, $(\mathscr{C}, \mathscr{R}, s)$, and for every type $t \in \mathscr{T}$:*

$$\emptyset \vdash_{\mathbf{Q}\varsigma} npe_t : [\![t]\!]$$

**Detecting and propagating errors**

When executing a statement or expression, execution must halt when a sub-expression evaluates to an error, just as errors are propagated to the top level in the RelJ semantics. To do this, we use the ckNPE construct, which we write as if it were a $\mathbf{Q}\varsigma$ term, but which has a translation to 'pure' $\mathbf{Q}\varsigma$ as given in Figure 5.6:

$$\text{ckNPE } x : t = a \text{ at } t' \text{ in } b$$

The above checks the sub-expression $a$, which is a translation of a RelJ term of type $t$, for error. If there is no error, then the continuation, $b$, is executed, wherein it may make use of $a$'s result in the variable $x$. The continuation $b$ represents the translation of the surrounding statement or expression, which had RelJ type $t'$. Where $a$ does evaluate to an error — that is one which has a true *isNPE* flag — then the continuation in $b$ is discarded, and the error value appropriate to $t'$ is returned.

The details of the implementations in Figure 5.6 are routine: the result of the sub-expression is assigned to a variable for later use in the continuation; the variable's value is checked for error (possibly after being unfolded, or unwrapped, depending on type); and the continuation is offered as the non-error continuation to the conditional.

Error values are only generated when a field access or method invocation is attempted on `null` value. We use a similar strategy to trap `null`s as that used for detecting error, using the ckNull construct:

$$\mathsf{ckNull}\ x : n = a\ \mathsf{at}\ t\ \mathsf{in}\ b$$

If $a$ is `null`, then an error value appropriate to type $t$ is returned. If not, continuation $b$ is executed, which will usually be the translation of a RelJ field access or method invocation. For brevity, ckNull includes a check which ensures that $a$ does not evaluate to an error — progress is impossible whether $a$ is an error or `null`.

Where we have more than one sub-expression to check for `null` or for error, we use the following shortened form:

$$\mathsf{ckNPE}\ x : n = a, x' : n' = a'\ \mathsf{at}\ t\ \mathsf{in}\ b$$

which is equivalent to:

$$\mathsf{ckNPE}\ x : n = a\ \mathsf{at}\ t\ \mathsf{in}\ \mathsf{ckNPE}\ x' : n' = a'\ \mathsf{at}\ t\ \mathsf{in}\ b$$

Similarly, where one sub-expression needs to be checked for `null`, and the other only for error (recall that the check for `null` contains an implicit error-check), we write:

$$\mathsf{ckNull}\ x : n = a\ \mathsf{ckNPE}\ x' : n' = a'\ \mathsf{at}\ t\ \mathsf{in}\ b$$

as shorthand for:

$$\mathsf{ckNull}\ x : n = a\ \mathsf{at}\ t\ \mathsf{in}\ \mathsf{ckNPE}\ x' : n' = a'\ \mathsf{at}\ t\ \mathsf{in}\ b$$

We need only specify the surrounding type once in the shortened form, as the type of the overall expression is identical regardless of which sub-expression is being examined.

Finally, we assume that the variables used by the above terms are fresh wherever they are introduced.

**Translation environment**

When translating RelJ terms, we assume that variables to represent the various relationship extents are present in the environment, and that there is a variable through which the instance factories may be accessed. This environment is there-

(TRANS NULL)

$$\frac{\vdash_{\mathbf{R}} n}{\Gamma \vdash_{\mathbf{R}} \texttt{null} : n \rhd \mathsf{null}_n}$$

(TRANS VAR)

$$\frac{\Gamma(x) = t}{\Gamma \vdash_{\mathbf{R}} x : t \rhd x}$$

(TRANS NEW)

$$\frac{\vdash_{\mathbf{R}} c}{\Gamma \vdash_{\mathbf{R}} \mathsf{new}\ \ c\texttt{()} : c \rhd \mathit{instFacs}.c.\mathit{new}}$$

(TRANS FLD)

$$\frac{\Gamma \vdash_{\mathbf{R}} e : n \rhd a \qquad \mathscr{F}_n^+(f) = t}{\Gamma \vdash_{\mathbf{R}} e\texttt{.}f\ : t \rhd \mathsf{ckNull}\ x : n = a\ \mathsf{at}\ t\ \mathsf{in}\ \mathsf{unfold}(x).f}$$

(TRANS FLDASS)

$$\frac{\Gamma \vdash_{\mathbf{R}} e : n \rhd a \qquad \Gamma \vdash_{\mathbf{R}} e' : t' \rhd b \qquad \mathscr{F}_n^+(f) = t'' \qquad \vdash_{\mathbf{R}} t' \leq t''}{\Gamma \vdash_{\mathbf{R}} e\texttt{.}f\ \texttt{=}\ e' : t' \rhd \mathsf{ckNull}\ x : n = a\ \mathsf{ckNPE}\ y : t' = b\ \mathsf{at}\ t'\ \mathsf{in}\ \mathsf{fold}([\![n]\!], \mathsf{unfold}(x).f \Leftarrow \varsigma(x : [\![n]\!]^{\circlearrowleft})y)}$$

(TRANS CALL)

$$\frac{\Gamma \vdash_{\mathbf{R}} e : n \rhd a \qquad \Gamma \vdash_{\mathbf{R}} e' : t \rhd b \qquad \mathscr{M}_n^+(m) = (x, t', t'', mb) \qquad \vdash_{\mathbf{R}} t \leq t'}{\Gamma \vdash_{\mathbf{R}} e\texttt{.}m(e') : t'' \rhd \mathsf{ckNull}\ y : n = a\ \mathsf{ckNPE}\ z : t = b\ \mathsf{at}\ t''\ \mathsf{in}\ \mathsf{unfold}(y).m(z)}$$

Figure 5.7: Translation of standard class-based features of RelJ into $\mathbf{Q}\varsigma$

fore of the form:

$$\mathit{none} : \mathsf{Top}, \mathit{extent}_r : \mathsf{Top}^{\ r \in \mathsf{dom}(\mathscr{R})},\ \mathit{instFacs} : \mathsf{Factories}$$

where Factories is the type of the object with factories for each reference type. These variables are established in the preamble of the translated program, which is given at the end of this chapter. For now, we assume their presence and, in the case of *instFacs*, that it contains a label for each reference type $n$ from which a factory for $n$ may be obtained.

**Translation of class-based features**

The translation of a RelJ expression $e$ to $\mathbf{Q}\varsigma$ term $a$ is written:

$$\Gamma \vdash_{\mathbf{R}} e : t \rhd a$$

where we assume that $e$ has type $t$ in the presence of environment $\Gamma$. We define the translation for the class-based portions of RelJ in Figure 5.7, where we do not repeat the constraints imposed by the RelJ type system unless they add clarity. We

assume throughout this translation that a RelJ term translated at some type can indeed be assigned that type:

$$\Gamma \vdash_{\mathbf{R}} e : t \rhd a \text{ implies } \Gamma \vdash_{\mathbf{R}} e : t$$

(TRANS NULL) translates `null` to its appropriately-typed representative from the $\mathsf{null}_n$ family.

Program variables are left untranslated by (TRANS VAR), as variable names are preserved by the translation of methods to pre-methods, given later.

(TRANS NEW) translates the allocation of a new class instance to the invocation of the appropriate factory's *new* method, via the *instFacs* variable that will be established in the translation preamble.

Field lookup is translated in the obvious way in (TRANS FLD): ckNull is used to ensure that the receiver is not `null` (or error) and, where this is not the case, the receiver is unfolded to expose its labels, and the field's label is selected.

Field update is translated by (TRANS FLDASS) in a similar way, with an additional check to ensure that the new value does not represent error (it is allowed to be `null`). The result of the update must also be folded, so that it conforms to the convention for reference type encodings.

Finally, method call is translated by (TRANS CALL). We check the receiver is not `null` and that the argument is not an error before selecting the method's label from the unfolded receiver, and applying the resulting function to the argument.

**Translation of relationship-based features**

The translation of relationship access is based on find, defined in Figure 5.8. The find construct takes the name of a relationship, $r$, and two terms, $a$ and $b$, which represent the source and destination of the $r$-instances to be returned. We allow $b$ to be left blank, written $\mathsf{find}(r, a, ?)$, where we only want to fix the source address.

The evaluation of $\mathsf{find}(r, a, b)$ uses query to emulate the function of $\rho$ from RelJ's operational semantics. This query fixes the values of the *from* and *to* labels to $a$ and $b$ respectively, and fixes the result objects' *extent* labels to $extent_r$.[2] The query does not fix values for the labels which represent fields, methods, *isNull* or *isNPE*, but we require their presence in the query so that the type makes all the objects' features available to the continuing computation.

The result of the query will have type list of $[\![r]\!]^{\circlearrowleft}$, which does not conform to the typing convention for translated reference types — we require the result to be passed as $[\![\texttt{set<}r\texttt{>}]\!]$, which is equivalent to list of $[\![r]\!]$. Therefore, we fold each member of the resulting list to the $[\![r]\!]$ type using foldall, whose definition is also given in Figure 5.8. The definition of foldall is based on map, which shall also be used in later portions of the translation. The map construct takes a list-

[2]Recall that we assume $extent_r$ exists as a variable in scope, which will be established when we give translations to entire programs.

$$\begin{aligned}
\mathsf{find}(r,a,b) \mathrel{\widehat{=}} \mathsf{foldall}(r,\ & \\
& \langle \mathit{extent} : \mathsf{Top} = \mathit{extent}_r, \\
& \mathit{from}_{r'} : [\![\mathcal{R}(r')_{\mathsf{from}}]\!] = a \quad {}^{r' \text{ s.t. } \vdash_{\mathbf{R}} r \leq r'}, \\
& \mathit{to}_{r'} : [\![\mathcal{R}(r')_{\mathsf{to}}]\!] = b \quad {}^{r' \text{ s.t. } \vdash_{\mathbf{R}} r \leq r'}, \\
& f : [\![\mathcal{F}_r^{+}(f)]\!] = ? \quad {}^{f \in \mathsf{dom}(\mathcal{F}_r^{+})}, \\
& m : [\![\mathcal{M}_r^{+}(m)_{\mathsf{arg}}]\!] \rightarrow [\![\mathcal{M}_r^{+}(m)_{\mathsf{ret}}]\!] = ? \quad {}^{m \in \mathsf{dom}(\mathcal{M}_r^{+})}, \\
& \mathit{isNull} : \mathsf{boolean}, \\
& \mathit{isNPE} : \mathsf{boolean} \rangle)
\end{aligned}$$

$$\mathsf{foldall}(r,a) \mathrel{\widehat{=}} \mathsf{map}(a, \lambda(x : [\![r]\!]^{\circlearrowleft})\mathsf{fold}([\![r]\!], x), [\![r]\!])$$

$$\begin{aligned}
\mathsf{map}(a, \lambda(x:A)b, B) \mathrel{\widehat{=}} \ & [\mathit{next} = \varsigma(s : [\mathit{next} : \mathsf{list\ of\ } A \rightarrow \mathsf{list\ of\ } B])\lambda(z : \mathsf{list\ of\ } A) \\
& \mathsf{match}\ z : \mathsf{list\ of\ } A \mathsf{\ with} \\
& \quad \mathsf{cons}(y, ys) \Rightarrow \mathsf{cons}((\lambda(x:A)b)(y), s.\mathit{next}(ys)) \mid \\
& \quad \mathsf{nil} \Rightarrow \mathsf{nil}].\mathit{next}(a)
\end{aligned}$$

Figure 5.8: Query-based implementation of RelJ's relationship store, with auxiliary functions foldall and map

generating term, a function to be applied to the list's members, and a return type for that function. Its implementation is standard; an object with a single method is used to iterate over successive members of the list. In the case of foldall, the function folds each list member so that they have type $[\![r]\!]$ as required by our typing convention.

Figure 5.9 gives translations for RelJ's relationship-based features, most of which are based on find. The simplest use of the query occurs in (TRANS RELINST), which checks the receiver for error and `null`, before using find to obtain all $r$-instances which relate the receiver to some other object.

A similar strategy is used in (TRANS RELOBJ), but to obtain the objects related to the receiver through $r$, rather than the $r$-instances themselves, we map the selection of each object's $\mathit{to}_r$ label across the result.

The translations of `from` and `to` in (TRANS TO) and (TRANS FROM) are equivalent to the translation of field access, except that the appropriate label is selected based on the receiver's static type in order to obtain the most precise type, as discussed in Section 5.1.3.

(TRANS RELADD) implements the RelJ's add operator. After the operands are checked for error and `null`, we use find to search for an instance of $r$ between them. If such an instance exists, it is returned; otherwise, the *new* method of $r$'s factory is invoked to create a fresh instance.

(TRANS RELREM) is structured similarly. Where no $r$-instance exists between the operands, then the appropriate `null` representative is returned. If an $r$-instance does exist, then its *extent* label is modified to *none* so that it no longer

(TRANS RELINST)

$$\frac{\Gamma \vdash_{\mathbf{R}} e : n \rhd a}{\Gamma \vdash_{\mathbf{R}} e{:}r : \texttt{set<}r\texttt{>} \rhd \text{ckNull } x : n = a \text{ at } \texttt{set<}r\texttt{>} \text{ in find}(r, x, ?)}$$

(TRANS RELOBJ)

$$\frac{\Gamma \vdash_{\mathbf{R}} e : n_1 \rhd a}{\begin{array}{l}\Gamma \vdash_{\mathbf{R}} e\texttt{.}r : \texttt{set<}\mathscr{R}(r)_{\text{to}}\texttt{>} \rhd \text{ckNull } x : n_1 = a \text{ at } \texttt{set<}\mathscr{R}(r)_{\text{to}}\texttt{>} \text{ in} \\ \qquad\qquad\qquad\qquad \text{let } ys : [\![\texttt{set<}r\texttt{>}]\!] = \text{find}(r, x, ?) \text{ in} \\ \qquad\qquad\qquad\qquad\quad \text{map}(ys, \lambda(z : [\![r]\!])\text{unfold}(z).to_r, [\![\mathscr{R}(r)_{\text{to}}]\!])\end{array}}$$

(TRANS FROM)

$$\frac{\Gamma \vdash_{\mathbf{R}} e : r \rhd a}{\Gamma \vdash_{\mathbf{R}} e\texttt{.from} : \mathscr{R}(r)_{\text{from}} \rhd \text{ckNull } x : r = a \text{ at } \mathscr{R}(r)_{\text{from}} \text{ in unfold}(x).from_r}$$

(TRANS TO)

$$\frac{\Gamma \vdash_{\mathbf{R}} e : r \rhd a}{\Gamma \vdash_{\mathbf{R}} e\texttt{.to} : \mathscr{R}(r)_{\text{to}} \rhd \text{ckNull } x : r = a \text{ at } \mathscr{R}(r)_{\text{to}} \text{ in unfold}(x).to_r}$$

(TRANS RELADD)

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathbf{R}} e_1 : n_3 \rhd a \qquad \Gamma \vdash_{\mathbf{R}} e_2 : n_4 \rhd b \\ \mathscr{R}(r) = (_, n_1, n_2, _, _) \qquad \vdash_{\mathbf{R}} n_3 \leq n_1 \qquad \vdash_{\mathbf{R}} n_4 \leq n_2\end{array}}{\begin{array}{l}\Gamma \vdash_{\mathbf{R}} r\texttt{.add(}e_1\texttt{,}e_2\texttt{)} : r \rhd \text{ckNull } x : n_3 = a, y : n_4 = b \text{ at } r \text{ in} \\ \qquad\qquad\qquad\qquad \text{let } z : [\![\texttt{set<}r\texttt{>}]\!] = \text{find}(r, x, y) \text{ in} \\ \qquad\qquad\qquad\qquad\quad \text{match } z : [\![\texttt{set<}r\texttt{>}]\!] \text{ with} \\ \qquad\qquad\qquad\qquad\qquad \text{cons}(z', zs') \Rightarrow z' \mid \\ \qquad\qquad\qquad\qquad\qquad \text{nil} \Rightarrow instFacs.r.new(x)(y)\end{array}}$$

(TRANS RELREM)

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathbf{R}} e_1 : n_3 \rhd a \qquad \Gamma \vdash_{\mathbf{R}} e_2 : n_4 \rhd b \\ \mathscr{R}(r) = (_, n_1, n_2, _, _) \qquad \vdash_{\mathbf{R}} n_3 \leq n_1 \qquad \vdash_{\mathbf{R}} n_4 \leq n_2\end{array}}{\begin{array}{l}\Gamma \vdash_{\mathbf{R}} r\texttt{.rem(}e_1\texttt{,}e_2\texttt{)} : r \rhd \\ \quad \text{ckNull } x : n_3 = a, y : n_4 = b \text{ at } r \text{ in} \\ \quad\quad \text{let } z : [\![\texttt{set<}r\texttt{>}]\!] = \text{find}(r, x, y) \text{ in} \\ \quad\quad\quad \text{match } z : [\![\texttt{set<}r\texttt{>}]\!] \text{ with} \\ \quad\quad\quad\quad \text{cons}(z', zs') \Rightarrow \text{fold}([\![r]\!], \text{unfold}(z').extent \Leftarrow \varsigma(w : [\![r]\!]^{\circlearrowleft})none) \mid \\ \quad\quad\quad\quad \text{nil} \Rightarrow \text{null}_r\end{array}}$$

Figure 5.9: Translations RelJ's relationship-based features

(TRANS EMPTY)

$$\frac{}{\Gamma \vdash_{\mathbf{R}} \epsilon \rhd \mathsf{void}}$$

(TRANS EXP)

$$\frac{\Gamma \vdash_{\mathbf{R}} e : t \rhd a \qquad \Gamma \vdash_{\mathbf{R}} s \rhd b}{\Gamma \vdash_{\mathbf{R}} e\,;\, s \rhd \mathsf{ckNPE}\ x : t = a\ \mathsf{at\ void\ in}\ b}$$

(TRANS FOR)

$$\frac{\Gamma \vdash_{\mathbf{R}} e : \mathtt{set}\text{<}n_1\text{>} \rhd a \qquad \Gamma, x \mapsto n_2 \vdash_{\mathbf{R}} s_1 \rhd b_1 \qquad \Gamma \vdash_{\mathbf{R}} s_2 \rhd b_2 \qquad \vdash_{\mathbf{R}} n_1 \le n_2}{\begin{array}{l} \Gamma \vdash_{\mathbf{R}} \mathtt{for}\ (n_2\ x\ :\ e)\ \{s_1\}\,;\, s_2 \rhd \\ \qquad\qquad \mathsf{ckNPE}\ y : \mathtt{set}\text{<}n_2\text{>} = a\ \mathsf{at\ void\ in} \\ \qquad\qquad\quad \mathsf{ckNPE}\ z : \mathtt{void} = \mathsf{for}(y, \lambda(x : [\![n_2]\!])b_1)\ \mathsf{at\ void\ in}\ b_2 \end{array}}$$

$$\begin{array}{l} \mathsf{for}(a, \lambda(w : [\![n]\!])b) \widehat{=} [\mathit{next} = \varsigma(s : [\mathit{next} : \mathsf{list\ of}\ [\![n]\!] \to [\![\mathtt{void}]\!]]) \\ \qquad\qquad \lambda(x : \mathsf{list\ of}\ [\![n]\!]) \\ \qquad\qquad\quad \mathsf{match}\ x : \mathsf{list\ of}\ [\![n]\!]\ \mathsf{with} \\ \qquad\qquad\qquad \mathsf{cons}(y, ys) \Rightarrow \\ \qquad\qquad\qquad\quad \mathsf{ckNPE}\ z : \mathtt{void} = (\lambda(w : [\![n]\!])b)(y)\ \mathsf{at\ void\ in} \\ \qquad\qquad\qquad\qquad s.\mathit{next}(ys) \mid \\ \qquad\qquad\qquad \mathsf{nil} \Rightarrow \mathsf{void}].\mathit{next}(a) \end{array}$$

Figure 5.10: Translation of statement sequences

appears in the find query — the instance is rendered inactive.

**Proposition 5.4.** *For a well-formed* RelJ *program, the translation of a well-typed expression:*

$$\Gamma \vdash_{R} e : t \rhd a$$

*will be well-typed in the presence of the assumed environment containing instFacs and extent objects, combined with the translation of e's typing environment,* $\Gamma$*:*

$$\mathit{none} : \mathit{Top},\ \mathit{extent}_r : \mathit{Top}^{\,r \in dom(\mathcal{R})},\ \mathit{instFacs} : \mathit{Factories},\ [\![\Gamma]\!] \vdash_{\mathbf{Q}\varsigma} a : [\![t]\!]$$

### Statements

RelJ statements are translated by the rules in Figure 5.10. Statements are not explicitly assigned types in the RelJ type system, so the translation relation is written:

$$\Gamma \vdash_{\mathbf{R}} s \rhd a$$

where we assume $s$ is well-typed in the presence of environment $\Gamma$.

In general, individual statements are translated to a $\mathbf{Q}\varsigma$ term of type $[\![\mathtt{void}]\!]$. Once that void value is checked for error, it is discarded and the remaining statements are left as the continuation. The resulting chain of let-bindings is terminated by the translation of an empty statement to void by (TRANS EMPTY).

Expressions in statement position are translated as usual by (TRANS EXP), but their results are discarded after being checked for error.

The `for` iterator is given a translation by (TRANS FOR). The target set is checked for error, after which the set is processed by for, based on map from Figure 5.8. Unlike map, for checks the result of each step for error before discarding it and making the recursive step: if an error has occurred, an error value of `void` type is immediately returned and no recursion takes place. Notice also that the translation of the `for` iterator's body is expected to have a free variable, $x$, which is then $\lambda$-bound when the body is passed to for.

**Proposition 5.5.** *For a well-formed RelJ program, the translation of a well-typed statement:*

$$\Gamma \vdash_{\boldsymbol{R}} s \rhd a$$

*will be well-typed in the presence of the assumed environment and the translation of the typing environment, $\Gamma$:*

$$none : Top,\ extent_r : Top^{\ r \in \mathrm{dom}(\mathscr{R})},\ instFacs : \mathsf{Factories},\ [\![\Gamma]\!] \vdash_{\mathbf{Q}\varsigma} a : [\![\mathtt{void}]\!]$$

### 5.3.3 Instance factories

As discussed in the previous section, we enclose all instance factories in a single object, so that classes and relationships may instantiate one another. This collection of factories will therefore have the following type, where $\mathsf{Factory}_n$ is an $n$-indexed family of types, one for each $n$-factory.

$$\mathsf{Factories} \mathrel{\widehat{=}} [c : \mathsf{Factory}_c{}^{\ c \in \mathrm{dom}(\mathscr{C})}, r : \mathsf{Factory}_r{}^{\ r \in \mathrm{dom}(\mathscr{R})}]$$

The factories themselves have the following types, which reflect the *new* methods that yield fresh instances, and the pre-methods from which those instances are constructed:

$$\begin{aligned} \mathsf{Factory}_c \mathrel{\widehat{=}} [&new : [\![c]\!], \\ &pre_m : [\![c]\!] \to [\![\mathscr{M}^+_c(m)_{\mathsf{arg}}]\!] \to [\![\mathscr{M}^+_c(m)_{\mathsf{ret}}]\!]^{\ m \in \mathrm{dom}(\mathscr{M}^+_c)}] \\ \mathsf{Factory}_r \mathrel{\widehat{=}} [&new : [\![\mathscr{R}(r)_{\mathsf{from}}]\!] \to [\![\mathscr{R}(r)_{\mathsf{to}}]\!] \to [\![r]\!], \\ &pre_m : [\![r]\!] \to [\![\mathscr{M}^+_r(m)_{\mathsf{arg}}]\!] \to [\![\mathscr{M}^+_r(m)_{\mathsf{ret}}]\!]^{\ m \in \mathrm{dom}(\mathscr{M}^+_r)}] \end{aligned}$$

Recall that there is a pre-method entry for every method contained in the class or relationship, but that the pre-method defers to the supertype's pre-method when that method is inherited.

We now give $\mathbf{Q}\varsigma$ implementations of factories for RelJ classes and relationships. These are based on the generation of pre-methods, which is specified by (TRANS METH) in Figure 5.11. (TRANS METH) uses the translation relation of the previous section to construct pre-methods by translating the method body's statement, $s$, and return expression, $e$. The statement body's value is checked for

(TRANS METH)

$$\frac{\begin{array}{l}\mathscr{M}_n(m) = (x, t, t', \{\ s\ \texttt{return}\ e;\ \})\\ \{x \mapsto t, \texttt{this} \mapsto n\} \cup \mathscr{L} \vdash_{\mathbf{R}} s \rhd a\\ \{x \mapsto t, \texttt{this} \mapsto n\} \cup \mathscr{L} \vdash_{\mathbf{R}} e : t' \rhd b\end{array}}{n \vdash_{\mathbf{R}} m \rhd \lambda(\texttt{this} : [\![n]\!])\lambda(x : [\![t]\!])\mathsf{ckNPE}\ y : \texttt{void} = a\ \mathsf{at}\ t'\ \mathsf{in}\ b}$$

Figure 5.11: Translation of RelJ methods, $m$, to $\mathbf{Q}\varsigma$ pre-methods, $a$.

error before the return expression is evaluated. Notice that the free variables representing the method's formal parameter and `this` are $\lambda$-bound in the resulting pre-method.

The definitions of factories for classes and relationships are given in Figure 5.12. Factories for classes are defined by (TRANS CLASS). A factory for a class, $c$, contains a constructor, *new*, and a collection of pre-methods, $pre_m$ for each (possibly inherited) method $m$. Where $c$ declares $m$, the translation of (TRANS METH) is used to obtain a pre-method; where $m$ is inherited, the pre-method is imported from the superclass's factory. When constructing an object, the methods are let-bound in order to resolve the resulting indirections, before being installed in the new object alongside labels for fields, `null`- and error-checking. When a method label is later selected on this newly-generated instance, the pre-method is applied to the self-parameter to produce a method which, when given an argument, will perform the method's action. The self-parameter must be folded, as per our typing convention. Fields are given suitable initial values by initSel.

Factories for relationships are derived from (TRANS REL), and have much the same form as those for classes, with the addition of the *from* and *to* labels, and the *extent* label which is used by the find query to isolate those $r$-instances which are current, and to exclude instances of $r$'s sub-relationships from the query result.

**Proposition 5.6.** *For a well-formed* $(\mathscr{C}, \mathscr{R}, s)$*, and for all reference types n, the n-factory:*

$$\vdash_{\boldsymbol{R}} n \rhd a_n$$

*will be well-typed in the presence of the assumed environment containing instFacs and extent objects:*

$$none : \mathsf{Top},\ extent_r : \mathsf{Top}^{\ r \in dom(\mathscr{R})},\ instFacs : \mathsf{Factories} \vdash_{\mathbf{Q}\varsigma} a_n : \mathsf{Factory}_n$$

### Programs

We now arrive at the top-level of our translation, which brings together the factories into a single, translated program, and which is given by (TRANS PROGRAM) in Figure 5.13. The class and relationship factories are installed in a single object in the presence of a preamble which sets up the extent objects *none* and $extent_r$ for

$$\mathsf{initSel}(t) \mathrel{\widehat{=}} \begin{cases} \mathsf{null}_t & \text{if } t = n \\ \mathsf{nil} & \text{otherwise} \end{cases}$$

(TRANS CLASS)

$$\frac{\mathscr{C}(c) = (c', \mathscr{F}, \mathscr{M}) \qquad \forall m \in \mathsf{dom}(\mathscr{M}) : c \vdash_{\mathbf{R}} m \rhd a_m}{\begin{array}{l} \vdash_{\mathbf{R}} c \rhd [new = \varsigma(cFac : \mathsf{Factory}_c) \\ \qquad \left(\begin{array}{l}\text{let } w_m : [\![c]\!] \to [\![\mathscr{M}_c^+(m)_{\mathsf{arg}}]\!] \to [\![\mathscr{M}_c^+(m)_{\mathsf{ret}}]\!] \\ \qquad\qquad = cFac.pre_m \text{ in}\end{array}\right)^{m \in \mathsf{dom}(\mathscr{M}_c^+)} \\ \qquad\quad [f = \varsigma(z : [\![c]\!]^{\circlearrowleft})\mathsf{initSel}(t)^{\,f \in \mathsf{dom}(\mathscr{F}_c^+)}, \\ \qquad\quad\ m = \varsigma(z : [\![c]\!]^{\circlearrowleft}) w_m(\mathsf{fold}([\![c]\!], z))^{\,m \in \mathsf{dom}(\mathscr{M}_c^+)}, \\ \qquad\quad\ isNull = \mathsf{false} \\ \qquad\quad\ isNPE = \mathsf{false}], \\ \quad pre_m = a_m{}^{m \in \mathsf{dom}(\mathscr{M})}, \\ \quad pre_{m'} = instFacs.c'.pre_{m'}{}^{\,m' \in \mathsf{dom}(\mathscr{M}_c^+) \setminus \mathsf{dom}(\mathscr{M})}] \end{array}}$$

(TRANS REL)

$$\frac{\mathscr{R}(r) = (r', n_1, n_2, \mathscr{F}, \mathscr{M}) \qquad \forall m \in \mathsf{dom}(\mathscr{M}) : r \vdash_{\mathbf{R}} m \rhd a_m}{\begin{array}{l} \vdash_{\mathbf{R}} r \rhd [new = \varsigma(rFac : \mathsf{Factory}_r)\lambda(x_1 : [\![n_1]\!])\lambda(x_2 : [\![n_2]\!]) \\ \qquad \left(\begin{array}{l}\text{let } w_m : [\![r]\!] \to [\![\mathscr{M}_r^+(m)_{\mathsf{arg}}]\!] \to [\![\mathscr{M}_r^+(m)_{\mathsf{ret}}]\!] \\ \qquad\qquad = rFac.pre_m \text{ in}\end{array}\right)^{m \in \mathsf{dom}(\mathscr{M}_r^+)} \\ \qquad\quad [extent = extent_r, \\ \qquad\quad\ from_{r'} = x_1{}^{\,r' \text{ s.t. } \vdash_{\mathbf{R}} r \leq r'}, \\ \qquad\quad\ to_{r'} = x_2{}^{\,r' \text{ s.t. } \vdash_{\mathbf{R}} r \leq r'}, \\ \qquad\quad\ f = \varsigma(z : [\![r]\!]^{\circlearrowleft})\mathsf{initSel}(t)^{\,f \in \mathsf{dom}(\mathscr{F}_r^+)}, \\ \qquad\quad\ m = \varsigma(z : [\![r]\!]^{\circlearrowleft}) w_m(\mathsf{fold}([\![r]\!], z))^{\,m \in \mathsf{dom}(\mathscr{M}_r^+)}, \\ \qquad\quad\ isNull = \mathsf{false}, \\ \qquad\quad\ isNPE = \mathsf{false}], \\ \quad pre_m = a_m{}^{m \in \mathsf{dom}(\mathscr{M})}, \\ \quad pre_{m'} = instFacs.r'.pre_{m'}{}^{\,m' \in \mathsf{dom}(\mathscr{M}_r^+) \setminus \mathsf{dom}(\mathscr{M})}] \end{array}}$$

Figure 5.12: Translation of RelJ classes and relationships into $\mathbf{Q}\varsigma$ factories

(TRANS PROGRAM)

$$\frac{\emptyset \vdash_{\mathbf{R}} s \rhd a \qquad \forall c \in \mathsf{dom}(\mathscr{C}) : \vdash_{\mathbf{R}} c \rhd a_c \qquad \forall r \in \mathsf{dom}(\mathscr{R}) : \vdash_{\mathbf{R}} r \rhd a_r}{\begin{array}{l}\vdash_{\mathbf{R}} (\mathscr{C},\mathscr{R},s) \rhd \mathsf{let}\ none : \mathsf{Top} = [l = \varsigma(x : [l : \mathsf{Top}])x.l]\ \mathsf{in} \\ \quad \mathsf{let}\ extent_r : \mathsf{Top} = \mathsf{clone}(none)\ \mathsf{in}\ ^{r \in \mathsf{dom}(\mathscr{R})} \\ \quad \mathsf{let}\ instFacs : \mathsf{Factories} = \\ \qquad [c = \varsigma(x : \mathsf{Factories})x.c\ ^{c \in \mathsf{dom}(\mathscr{C})}, \\ \qquad\ r = \varsigma(x : \mathsf{Factories})x.r\ ^{r \in \mathsf{dom}(\mathscr{R})}]\ \mathsf{in} \\ \quad \left(\begin{array}{l}\mathsf{let}\ instFac_n : \mathsf{Factory}_n = a_n\ \mathsf{in} \\ \mathsf{let}\ _ : \mathsf{Factories} = \\ \quad instFacs.n \Leftarrow \varsigma(x : \mathsf{Factories})instFac_n\ \mathsf{in}\end{array}\right)^{n \in \mathsf{dom}(\mathscr{C}) \cup \mathsf{dom}(\mathscr{R})} \\ \quad \vdots \\ \quad \mathsf{in}\ a\end{array}}$$

Figure 5.13: Translation of RelJ programs into $\mathbf{Q}\varsigma$

each relationship $r$. Notice that we install the factories after creation of the *instFacs* object, so that the factory objects are instantiated only once — if the *instFacs* object were to have methods of the form $\varsigma(x : \mathsf{Factories})a_c$, a new factory would be created for every factory access. Finally, the translation of the program's 'main' statement forms the body of the resulting nest of `let`-bindings to yield a closed $\mathbf{Q}\varsigma$ term that represents the original RelJ program.

**Proposition 5.7.** *For well-formed program* $(\mathscr{C},\mathscr{R},s)$*:*

$$\vdash_{\mathbf{R}} (\mathscr{C},\mathscr{R},s) \rhd a \text{ implies } \emptyset \vdash_{\mathbf{Q}\varsigma} a : [\![\mathtt{void}]\!]$$

## 5.4 Conclusion

This chapter demonstrated how RelJ programs may be translated to $\mathbf{Q}\varsigma$. It is important to note that whilst RelJ's model of class-based programming is very different to $\mathbf{Q}\varsigma$'s object-based model — indeed, significant effort was required to provide an environment in which RelJ's model could be supported — the implementation of RelJ relationships was straightforward, using heap query. Thus, we have shown that the relationship store of RelJ is merely an abstraction of information already available in the heap, which may be exposed by query.

CHAPTER 6

# Conclusion

In this thesis, we have seen how relationships are well-represented in models of object-oriented systems, and therefore in programmer intuition, but not in object-oriented languages themselves. We conducted a formal exploration of relationships in object-oriented languages with RelJ and **Q**$\varsigma$.

RelJ introduced relationships in a class-based language with the same status as classes: relationships in RelJ have fields and methods, they can extend other relationships, their instances can be referenced and relationships can themselves take part in other relationships. Relationships in RelJ are implemented using the relationship store, which is an abstract representation of the heap and encapsulates much of the complexity involved in maintaining relationship structures. This complexity was particularly evident in the demonstration of a semantics of relationship inheritance that differs from traditional class inheritance.

Based on the observation that the relationship store is derivable from the heap, we described **Q**$\varsigma$, a strongly-typed object calculus with heap query. Relationships are once again reunited with inter-object references, but query eliminates the complex machinery required to encode them. We demonstrated this by giving a translation from RelJ to **Q**$\varsigma$. The translation of RelJ's class-based features into **Q**$\varsigma$ was rather cumbersome owing to RelJ's comparatively liberal, nominal type system and the need to emulate `null`. However, the relationship-based features were easily translated into heap queries.

For both RelJ and **Q**$\varsigma$, we gave a formal type system and semantics, and showed that no well-typed programs in either language can get into an unsafe state.

## 6.1 Future work

We conclude by reviewing some opportunities for further investigation:

**Query and encapsulation** Without query, an object must be able to follow a chain of references to another object in order to access it. Heap query eliminates

this restriction so that all parts of the heap may be accessed regardless of references. As a result, encapsulation of aggregate objects is threatened: techniques to encapsulate aggregate objects by restricting aliasing [4, 6, 17, 22, 44] lose much of their power in the presence of query. The issue of encapsulation certainly deserves more work: in particular, the ability for objects to tag themselves as 'unqueryable', or the division of the heap into disjoint queryable sections as in Section 4.6.2 may help to preserve encapsulation. How objects are allocated to such zones — that is, what precisely constitutes an object that ought to be encapsulated — remains an open problem, though various concepts of ownership seem promising [4, 23].

**Garbage collection** Garbage collectors normally dispose of unreferenced objects, but these objects may now appear in the results of heap queries. Whilst better encapsulation will help inform decisions about which objects ought to be garbage collected, it is also important that the collector does not dispose of objects that may appear in query results that are still being examined: cooperation between the query engine and the garbage collector will be required.

**Query for class-based languages** Willis et al.'s work on run-time heap query for Java [76] gives an indication of how **Q**ς's query operator might be mapped into a more mainstream language. Nevertheless, the formal work of **Q**ς must be carried over to the class-based world, especially when considering the application of orthogonal techniques to help restore encapsulation boundaries. It is possible that hiding queries inside abstractions like RelJ's relationships might prove less dangerous to encapsulation in general.

**Improving the query language** **Q**ς's query language is relatively simple, but more complex object-based query languages exist, notably Object Query Language (OQL) from the ODMG [20] and native query expressions in LINQ [54]. Whilst filtering mechanisms and joins can be encoded with **Q**ς's simple queries combined with iterator objects, a more expressive query language would be desirable in practice.

**Accessing databases** Language extensions have been proposed that permit database queries to be expressed natively in object-oriented programming languages, such as the LINQ and C$\omega$ extensions to C# [11, 54]. Alongside native queries, these provide a more flexible model for supporting the representation of XML and relational data from databases. It would be interesting to investigate the rôles of relationships and heap query in strategies for handling data from object-oriented and object-relational databases.

**Development experience** Whilst researchers may freely generate new extensions to programming languages and software engineering techniques, it is ul-

timately the experience of programmers that will decide the usefulness and the form of relationship abstractions in the field. How developers use relationships in practice may only be determined by observation, but a flexible formal model is vital for determining whether features demanded by programmers can be provided safely.

**Implementation** Implementations of languages and libraries supporting relationships exist [2, 59, 62], as do implementations heap query [76]. Whilst RelJ and $\mathbf{Q}\varsigma$ could be adopted as formal models for such implementations, it would be both interesting and useful to implement RelJ and $\mathbf{Q}\varsigma$ directly. By equipping the C# run-time environment or the Java virtual machine with relationships/query, they can be represented natively and securely in bytecode, though established techniques would undoubtedly be used in the implementation [59, 76]. Furthermore, it is essential that any such implementation be flexible enough that models can be adjusted as programmer experience is evaluated — such flexibility may even be desirable at the language level so that programmers may apply their intuition to select an appropriate implementation of relationships.

# Guide to notation

This guide summarizes the notation used in this thesis, divided into three groups: notation for RelJ, notation for $\mathbf{Q}\varsigma$ and notation for the translation, which inherits notation from both the language formalisms.

## RelJ (Chapters 2, 3 and 5)

**Symbols**

| Symbol | Description |
|---|---|
| $\diamond$ | see **Judgements** |
| $\bullet$ | Context hole (Fig. 2.10, p. 53) |
| $\uparrow_\iota$ | see **Compound forms** |
| $\vdash_{\mathbf{R}}$ | see **Judgements** |
| $\rightsquigarrow_{\mathbf{R}}$ | Execution relation (Fig. 2.13, p. 58) |
| $[-]$ | see **Compound forms** |
| $\langle - \rangle$ | see **Compound forms** |
| $\langle\!\langle - \rangle\!\rangle$ | see **Compound forms** |
| $\sqcup$ | Least upper bound of one or more types (p. 49) |
| | |
| $\Gamma$ | Typing environment (p. 45) |
| | |
| $\gamma$ | Local variable store (p. 55) |
| $\epsilon$ | Empty statement (Fig. 2.3, p. 41) |
| $\iota$ | Object identifier/address (p. 54) |
| $\rho$ | Relationship store (p. 54) |
| $\sigma$ | Heap (p. 54) |
| | |
| $\mathscr{C}$ | Class lookup table (p. 42) |
| $\mathscr{E}$ | Evaluation context (Fig. 2.10, p. 53) |
| $\mathscr{E}_{\mathrm{e}}$ | for expressions |
| $\mathscr{E}_{\mathrm{s}}$ | for statements |
| $\mathscr{F}$ | Field lookup table (p. 42) |
| $\mathscr{F}_n$ | for reference type $n$ |

**Symbols (continued)**

| Symbol | Description |
|---|---|
| $\mathscr{F}_n^+$ | including inherited fields |
| $\mathscr{L}$ | Local variable lookup table (p. 43) |
| $\mathscr{M}$ | Method lookup table (p. 42) |
| $\mathscr{M}_n$ | for reference type $n$ |
| $\mathscr{M}_n^+$ | including inherited methods |
| $\mathscr{P}$ | Abstract program (p. 42) |
| $\mathscr{R}$ | Relationship lookup table (p. 43) |
| $\mathscr{T}$ | Set of valid program types (p. 42) |
| | |
| $R$ | Term (Fig. 2.3, p. 41) |
| | |
| $c$ | Class name (Fig. 2.3, p. 41) |
| $e$ | Expression (Fig. 2.3, p. 41) |
| $f$ | Field name (Fig. 2.3, p. 41) |
| $l$ | Assignment target, l-value (Fig. 2.3, p. 41) |
| $m$ | Method name (Fig. 2.2, p. 40) |
| $mb$ | Method body (Fig. 2.3, p. 41) |
| $mc$ | Multiplicity constraint (Fig. 3.6, p. 77) |
| $n$ | Reference type (Fig. 2.3, p. 41) |
| $o$ | Object (p. 54) |
| $p$ | Program (Fig. 2.3, p. 41) |
| $r$ | Relationship name (Fig. 2.2, p. 40) |
| $s$ | Statement (Fig. 2.3, p. 41) |
| $t$ | Type (Fig. 2.3, p. 41) |
| $v$ | Value (Fig. 2.3, p. 41) |
| $w$ | Error (p. 53) |
| $x$ | Variable name (Fig. 2.2, p. 40) |

**Compound forms**

| Form | Description |
|---|---|
| $- \vdash_{\mathbf{R}} -$ | see **Judgements** |
| $R[x'/x]$ | Substitution of $x$ with $x'$ in term $R$ (p. 53) |
| $\phi[a \mapsto b]$ | Update map $\phi$ such that $a$ maps to $b$ (p. 45) |
| $\sigma \uparrow_\iota f$ | Find $f$'s implementation for $\iota$ (Fig. 3.2, p. 72) |
| $\mathscr{E}[e]$ | Context instantiation (p. 52) |
| $\langle \sigma, \rho, \gamma, R \rangle$ | Configuration (p. 55) |
| $\langle \sigma, \rho, \gamma, w \rangle$ | Error configuration |
| $\langle \sigma, \rho, \gamma, R \rangle \rightsquigarrow_{\mathbf{R}} \langle \sigma, \rho, \gamma, R \rangle$ | Execution step (Fig. 2.13, p. 58) |
| $\langle\langle c \Vert f_1 : v_1, \ldots \rangle\rangle$ | Class instance/Object (p. 54) |
| $\langle\langle r, \iota_1, \iota_2 \Vert f_1 : v_1, \ldots \rangle\rangle$ | Relationship instance (p. 54) |
| $\langle\langle r, \iota_1, \iota_2, \iota \Vert f_1 : v_1, \ldots \rangle\rangle$ | with shared inheritance (p. 71) |

**Judgements**

| Judgement | Description |
|---|---|
| $\vdash_{\mathbf{R}} \diamond$ | The program is well-formed (Fig. 2.9, p. 52) |
| $\vdash_{\mathbf{R}} t$ | $t$ is a well-formed type (Fig. 2.9, p. 52) |
| $\vdash_{\mathbf{R}} t_1 \leq t_2$ | $t_1$ is a subtype of $t_2$ (Fig. 2.5, p. 45) |
| $\Gamma \vdash_{\mathbf{R}} \gamma$ | Locals store $\gamma$ agrees with $\Gamma$ (Fig. 2.12, p. 56) |
| $\Gamma \vdash_{\mathbf{R}} \sigma$ | Heap $\sigma$ is well-formed in $\Gamma$ (Fig. 2.12, p. 56) |
| $\Gamma \vdash_{\mathbf{R}} \langle \sigma, \rho, \gamma, e \rangle : t$ | Expression configuration is well-typed (Fig. 2.12, p. 56) |
| $\Gamma \vdash_{\mathbf{R}} \langle \sigma, \rho, \gamma, s \rangle$ | Statement configuration is well-typed (Fig. 2.12, p. 56) |
| $\Gamma \vdash_{\mathbf{R}} e : t$ | $e$ has type $t$ in environment $\Gamma$ (Fig. 2.6, p. 46) |
| $\Gamma \vdash_{\mathbf{R}} o$ | Object $o$ is well-formed in $\Gamma$ (Fig. 2.12, p. 56) |
| $\Gamma \vdash_{\mathbf{R}} s$ | $s$ is well-typed in environment $\Gamma$ (Fig. 2.6, p. 46) |
| $n \vdash_{\mathbf{R}} f$ | $f$ is a well-declared field of $n$ (Fig. 2.8, p. 50) |
| $n \vdash_{\mathbf{R}} m$ | $m$ is a well-declared method of $n$ (Fig. 2.8, p. 50) |

## Qς (Chapters 4 and 5)

**Symbols**

| Symbol | Description |
|---|---|
| ? | Query wildcard (Fig. 4.4, p. 93) |
| $\vdash_{\mathbf{Q\varsigma}}$ | see **Judgements** |
| $\#$ | see **Compound forms** |
| $\rightarrow$ | see **Compound forms** |
| $[-]$ | see **Compound forms** |
| $\langle\!\langle - \rangle\!\rangle$ | see **Compound forms** |
| | |
| $\iota$ | Object address (Fig. 4.1, p. 88) |
| $\lambda$ | Abstraction binder (Fig. 4.1, p. 88) |
| $\mu$ | Recursive type binder (Fig. 4.2, p. 89) |
| $\sigma$ | Store (p. 101) |
| $\varsigma$ | Self-parameter binder (Fig. 4.1, p. 88) |
| | |
| $\mathscr{E}$ | Evaluation context (Fig. 4.11, p. 102) |
| | |
| $A, B$ | Type (Fig. 4.2, p. 89) |
| $E$ | Environment (p. 96) |
| $\mathsf{FV}(a)$ | Free variables in $a$ (p. 91) |
| $O$ | Object type (Fig. 4.2, p. 89) |
| $Q$ | Queryable type (Fig. 4.4, p. 93) |
| $X, Y$ | Type variable (Fig. 4.1, p. 88) |
| | |
| $a, b$ | Term (Fig. 4.1, p. 88) |
| $c$ | Configuration (p. 101) |

**Symbols (continued)**

| Symbol | Description |
|---|---|
| $l$ | Label (Fig. 4.1, p. 88) |
| $lv$ | List value (Fig. 4.4, p. 93) |
| $m$ | Method definition (Fig. 4.1, p. 88) |
| $o$ | Object (Fig. 4.1, p. 88) |
| $qv$ | Queryable value (Fig. 4.4, p. 93) |
| $v$ | Value (Fig. 4.1, p. 88) |
| $w, x, y, z$ | Variable (Fig. 4.1, p. 95) |

**Compound forms**

| Form | Description |
|---|---|
| $(a, \sigma)$ | Configuration (p. 101) |
| $x \# a$ | Variable $x$ is fresh for term $a$ (p. 91) |
| $A <: B$ | $A$ is a subtype of $B$ (Fig. 4.8, p. 98) |
| $A \to B$ | Function type (Fig. 4.2, p. 89) |
| $[f_1 : m_1, \ldots]$ | Object definition (Fig. 4.1, p. 88) |
| $[f_1 : A_1, \ldots]$ | Object type (Fig. 4.2, p. 89) |
| $\phi[\iota \mapsto o]$ | Store update (p. 101) |
| $\langle l_1 = m_1, \ldots \rangle$ | Heap query (Fig. 4.4, p. 93) |
| $\langle\langle z \mid \psi(z) \rangle\rangle$ | Deterministic list comprehension (p. 103) |
| $a\{\!\{x'/x\}\!\}$ | Substitution of $x'$ for $x$ in term $a$ (p. 91) |
| $A\{\!\{X'/X\}\!\}$ | Substitution of $X'$ for $X$ in type $A$ (p. 91) |

**Judgements**

| Judgement | Description |
|---|---|
| $\vdash_{\mathbf{Q}\varsigma} E \ \mathsf{ok}$ | $E$ is a well-formed environment (Fig. 4.6, p. 96) |
| $E \vdash_{\mathbf{Q}\varsigma} A$ | $A$ is a well-formed type in $E$ (Fig. 4.7, p. 97) |
| $E \vdash_{\mathbf{Q}\varsigma} A <: B$ | $A$ is a subtype of $B$ in $E$ (Fig. 4.8, p. 98) |
| $E \vdash_{\mathbf{Q}\varsigma} a : A$ | $a$ has type $A$ in the presence of $E$ (Fig. 4.9, p. 99) |
| $E \vdash_{\mathbf{Q}\varsigma} \sigma$ | Store $\sigma$ is well-formed (Fig. 4.10, p. 102) |
| $E \vdash_{\mathbf{Q}\varsigma} (a, \sigma) : A$ | Configuration is well-typed (Fig. 4.10, p. 102) |

## Translation (Chapter 5)

Most notation for the translation is inherited from the formalizations of RelJ and **Q**ς, with the following additions:

**Notation**

| | Description |
|---|---|
| $\mathscr{A}$ | Finite collection of types (p. 120) |
| $[\![ t ]\!]$ | Translation of RelJ type $t$ (Fig. 5.3, p. 123) |

**Notation (continued)**

| | Description |
|---|---|
| $[\![t]\!]^{\circ}$ | unfolded (p. 122) |
| $[\![\Gamma]\!]$ | Translation of RelJ environment (p. 123) |
| $\vdash_{\mathbf{R}} (\mathscr{C}, \mathscr{R}, s) \rhd a$ | Program translation (Fig. 5.13, p. 136) |
| $\vdash_{\mathbf{R}} n \rhd a$ | Class/relationship translation (Fig. 5.12, p. 135) |
| $n \vdash_{\mathbf{R}} m \rhd a$ | Method translation (Fig. 5.12, p. 135) |
| $\Gamma \vdash_{\mathbf{R}} e : t \rhd a$ | Expression translation (Figs. 5.7–5.9, p. 128) |
| $\Gamma \vdash_{\mathbf{R}} s \rhd a$ | Statement translation (Fig. 5.10, p. 132) |

# Bibliography


[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer Verlag, 1996. (Cited on pages 14, 85, 86, 88, 95, 97, 100, and 115.)

[2] A. Albano, G. Ghelli, and R. Orsini. A relationship mechanism for a strongly typed object-oriented database programming language. In *Proceedings of VLDB*, pages 565–575. Morgan Kaufmann, 1991. (Cited on pages 26, 27, and 139.)

[3] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An object data model with roles. In *Proceedings of VLDB*, pages 39–51. Morgan Kaufmann, 1993. (Cited on page 71.)

[4] J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In *Proceedings of ECOOP*, volume 3086 of *Lecture Notes in Computer Science*, pages 1–25. Springer Verlag, 2004. (Cited on pages 9, 15, 22, and 138.)

[5] C. A. Alexander. *A pattern language*. Oxford University Press, 1977. (Cited on page 15.)

[6] P. S. Almeida. Balloon types: Controlling sharing of state in data types. In *Proceedings of ECOOP*, volume 1241 of *Lecture Notes in Computer Science*, pages 32–59. Springer Verlag, 1997. (Cited on pages 9, 15, 22, and 138.)

[7] C. Anderson. *Type inference for Javascript*. PhD thesis, Department of Computing, Imperial College, March 2006. (Cited on page 14.)

[8] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithm language ALGOL 60. *Communications of the ACM*, 6(1):1–17, 1963. (Cited on page 10.)

[9] S. Balzer, P. Eugester, and T. R. Gross. Relations: Abstracting object collaborations. Technical Report 539, Department of Computer Science, ETH Zurich, 2006. (Cited on pages 33 and 82.)

[10] S. Balzer, T. R. Gross, and P. Eugster. A relational model of object collaborations and its use in reasoning about relationships. In *Proceedings of ECOOP, to appear*. Springer Verlag, 2007. (Cited on page 26.)

[11] G. Bierman, E. Meijer, and W. Schulte. The essence of C$\omega$. In *Proceedings of ECOOP*, volume 3586 of *Lecture Notes in Computer Science*, pages 287–311. Springer Verlag, 2005. (Cited on pages 83 and 138.)

[12] G. M. Bierman and A. Wren. First-class relationships in an object-oriented language. In *Proceedings of ECOOP*, volume 3586 of *Lecture Notes in Computer Science*, pages 262–286. Springer Verlag, 2005. (Cited on page 29.)

[13] G. M. Bierman and A. Wren. First-class relationships in an object-oriented language. Technical Report 642, Computer Laboratory, University of Cambridge, August 2005. (Cited on page 75.)

[14] G. M. Bierman, M. J. Parkinson, and A. M. Pitts. MJ: A core imperative calculus for Java and Java with effects. Technical Report 563, University of Cambridge Computer Laboratory, 2003. (Cited on pages 29 and 44.)

[15] G. Booch. *Object-oriented analysis and design with applications*. Benjamin-Cummings Publishing Co., second edition, 1994. (Cited on pages 10 and 16.)

[16] J. Bosch. Design patterns as language constructs. *Journal of Object-Oriented Programming*, 11(2):18–32, 1998. (Cited on pages 16, 20, and 25.)

[17] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Proceedings of POPL*, volume 38(1) of *SIGPLAN Notices*, pages 213–223. ACM Press, 2003. (Cited on pages 15, 22, and 138.)

[18] British Standards Institute and B. Stroustrup. *The C++ Standard: Incorporating Technical Corrigendum No. 1*. John Wiley & Sons, 2003. (Cited on pages 10, 29, and 71.)

[19] M. Broy and E. Denert, editors. *Software pioneers: contributions to software engineering*. Springer Verlag, 2002. (Cited on page 10.)

[20] R. G. G. Cattell et al. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann, 2000. (Cited on pages 27, 31, 38, and 138.)

[21] P. P.-S. Chen. The entity-relationship model – toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976. (Cited on pages 9, 10, 16, and 18.)

[22] D. G. Clarke and T. Wrigstad. External uniqueness is unique enough. In *Proceedings of ECOOP*, volume 2743 of *Lecture Notes in Computer Science*, pages 176–200. Springer Verlag, 2003. (Cited on pages 22 and 138.)

[23] D. G. Clarke, J. Noble, and J. M. Potter. Simple ownership types for object containment. In *Proceedings of ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, pages 53–76. Springer Verlag, 2001. (Cited on pages 9, 15, 22, 85, and 138.)

[24] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970. (Cited on page 111.)

[25] W. R. Cook. A proposal for making Eiffel type-safe. *Computer Journal*, 32 (4):305–311, 1989. (Cited on page 23.)

[26] O.-J. Dahl and K. Nygaard. Class and subclass declarations. In *Proceedings of Simulation Programming Languages*, pages 158–174, 1967. (Cited on page 10.)

[27] D. Dean, E. W. Felten, and D. S. Wallach. Java security: From HotJava to Netscape and beyond. In *IEEE Symposium on Security and Privacy*, pages 190–200. IEEE Computer Society Press, 1996. (Cited on page 23.)

[28] E. W. Dijkstra. Letters to the editor: Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968. (Cited on page 10.)

[29] S. Drossopoulou. An abstract model of Java dynamic linking and loading. In *Proceedings of TIC*, volume 2071 of *Lecture Notes in Computer Science*, pages 53–84. Springer Verlag, 2000. (Cited on page 42.)

[30] S. Drossopoulou and S. Eisenbach. Java is type safe — probably. In *Proceedings of ECOOP*, volume 1241 of *Lecture Notes in Computer Science*, pages 389–418. Springer Verlag, 1997. (Cited on pages 23, 29, and 59.)

[31] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Fickle: Dynamic object re-classification. In *Proceedings of ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, pages 130–149. Springer Verlag, 2001. (Cited on page 71.)

[32] S. Ducasse, M. Blay-Fornarino, and A. M. Pinna-Dery. A reflective model for first class dependencies. In *Proceedings of OOPSLA*, volume 30(10) of *SIGPLAN Notices*, pages 265–280. ACM Press, 1995. (Cited on page 22.)

[33] Ecma International. *Standard ECMA-262: The ECMAScript language specification*. Iuniverse, third edition, December 1999. (Cited on pages 14 and 85.)

[34] B. Emir, A. Kennedy, C. Russo, and D. Yu. Variance and generalized constraints for C# generics. In *Proceedings of ECOOP*, volume 4067 of *Lecture Notes in Computer Science*, pages 279–303. Springer Verlag, 2006. (Cited on pages 14 and 95.)

[35] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proceedings of POPL*, pages 171–183. ACM Press, 1998. (Cited on pages 14 and 29.)

[36] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *Proceedings of ECOOP*, volume 707 of *Lecture Notes in Computer Science*, pages 406–431. Springer Verlag, 1993. (Cited on page 15.)

[37] A. D. Gordon, P. D. Hankin, and S. B. Lassen. Compilation and equivalence of imperative objects. In *Proceedings of FSTTCS*, volume 1346 of *Lecture Notes in Computer Science*, pages 74–87. Springer Verlag, 1997. (Cited on pages 85 and 86.)

[38] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., 2000. (Cited on pages 10, 29, 44, and 95.)

[39] Y.-G. Guéhéneuc and H. Albin-Amiot. Recovering binary class relationships: putting icing on the UML cake. In *Proceedings of OOPSLA*, volume 39(10) of *SIGPLAN Notices*, pages 301–314. ACM Press, 2004. (Cited on page 20.)

[40] J. Guttag. Abstract data types and the development of data structures. *Communications of the ACM*, 20(6):396–404, 1977. (Cited on page 10.)

[41] W. Harrison, C. Barton, and M. Raghavachari. Mapping UML designs to Java. In *Proceedings of OOPSLA*, volume 35(10) of *SIGPLAN Notices*, pages 178–187. ACM Press, 2000. (Cited on page 19.)

[42] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *Proceedings of ECOOP/OOPSLA*, volume 25(10) of *SIGPLAN Notices*, pages 169–180. ACM Press, 1990. (Cited on page 22.)

[43] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. (Cited on page 10.)

[44] J. Hogg. Islands: Aliasing protection in object-oriented languages. In *Proceedings of OOPSLA*, volume 26(10) of *SIGPLAN Notices*, pages 271–285. ACM Press, 1991. (Cited on pages 9, 15, 22, and 138.)

[45] S. Hwang and S. Lee. Modelling semantic relationships and constraints in object-oriented databases. In *Proceedings of SIGBDP TDES*, pages 396–416. ACM Press, 1990. (Cited on page 27.)

[46] I. Jacobson. *Object-oriented software engineering*. ACM Press, 1992. (Cited on page 16.)

[47] I. Jacobson, G. Booch, and J. E. Rumbaugh. *The unified software development process*. Addison-Wesley, 1999. (Cited on pages 9, 10, and 16.)

[48] A. C. Kay. The early history of Smalltalk. In *Proceedings of HOPL*, volume 28(3) of *SIGPLAN Notices*, pages 69–95. ACM Press, 1993. (Cited on page 10.)

[49] B. B. Kristensen. Complex associations: Abstractions in object-oriented modeling. In *Proceedings of OOPSLA*, volume 29(10) of *SIGPLAN Notices*, pages 272–286. ACM Press, 1994. (Cited on page 25.)

[50] R. Lencevicius, U. Hölzle, and A. K. Singh. Query-based debugging of object-oriented programs. In *Proceedings of OOPSLA*, volume 32(10) of *SIGPLAN Notices*, pages 304–317. ACM Press, 1997. (Cited on page 26.)

[51] B. Liskov and S. Zilles. Programming with abstract data types. In *Proceedings of SIGPLAN symposium on Very High Level Languages*, volume 9(4) of *SIGPLAN Notices*, pages 50–59. ACM Press, 1974. (Cited on page 10.)

[52] Y. D. Liu and S. F. Smith. Interaction-based programming with classages. In *Proceedings of OOPSLA*, volume 40(10) of *SIGPLAN Notices*, pages 191–209. ACM Press, 2005. (Cited on pages 25 and 82.)

[53] M. Lutz. *Programming Python*. O'Reilly, third edition, August 2006. (Cited on page 29.)

[54] E. Meijer, B. Beckman, and G. M. Bierman. LINQ: Reconciling object, relations and XML in the .NET framework. In *SIGMOD International Conference on Management of Data*, pages 706–706. ACM Press, 2006. (Cited on page 138.)

[55] Microsoft Corporation. *Microsoft C# Language Specifications*. Microsoft Press, 2001. (Cited on pages 10 and 29.)

[56] J. Noble. Basic relationship patterns. *Pattern Languages of Program Design*, 4, 1999. (Cited on pages 9 and 19.)

[57] J. Noble and J. Grundy. Explicit relationships in object-oriented development. In *Proceedings of TOOLS*, pages 2–13. IEEE Computer Society, 1998. (Cited on pages 9 and 22.)

[58] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972. (Cited on page 10.)

[59] D. J. Pearce and J. Noble. Relationship aspects. In *Aspect-Oriented Software Development (AOSD)*, pages 75–86. ACM Press, 2006. (Cited on pages 26 and 139.)

[60] B. C. Pierce. *Types and programming languages*. MIT Press, 2002. (Cited on pages 88, 94, 97, and 100.)

[61] J. E. Rumbaugh. Controlling propagation of operations using attributes on relations. In *Proceedings of OOPSLA*, volume 23(11) of *SIGPLAN Notices*, pages 285–296. ACM Press, 1988. (Cited on page 25.)

[62] J. E. Rumbaugh. Relations as semantic constructs in an object-oriented language. In *Proceedings of OOPSLA*, volume 22(12) of *SIGPLAN Notices*, pages 466–481. ACM Press, 1987. (Cited on pages 9, 22, 24, and 139.)

[63] J. E. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented modeling and design*. Prentice-Hall, 1991. (Cited on pages 10 and 16.)

[64] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behaviour. In *Proceedings of ECOOP*, volume 2743 of *Lecture Notes in Computer Science*, pages 248–274. Springer Verlag, 2003. (Cited on page 14.)

[65] A. V. Shah, J. H. Hamel, R. A. Borsari, and J. E. Rumbaugh. DSM: An object-relationship modeling language. In *Proceedings of OOPSLA*, volume 24(10) of *SIGPLAN Notices*, pages 191–202. ACM Press, 1989. (Cited on page 24.)

[66] J. M. Smith and D. C. P. Smith. Database abstractions: Aggregation and generalizations. *ACM Transactions on Database Systems*, 2(2):105–133, 1977. (Cited on pages 18 and 35.)

[67] J. Soukup. *Implementing patterns*, pages 395–412. ACM Press/Addison-Wesley, 1995. (Cited on page 20.)

[68] P. Stevens and R. Pooley. *Using UML: Software engineering with objects and components*. Addison-Wesley, 1999. (Cited on pages 9 and 32.)

[69] B. Stroustrup. What is object-oriented programming? *IEEE Software*, 5(3): 10–20, 1988. (Cited on page 11.)

[70] Sun Microsystems Inc. JHAT: Java Heap Analysis Tool. `http://java.sun.com/javase/6/docs/technotes/tools/share/jhat.html`, 2006. (Cited on page 26.)

[71] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Proceedings of OOPSLA*, volume 22(12) of *SIGPLAN Notices*, pages 227–242. ACM Press, 1987. (Cited on pages 10 and 14.)

[72] W3C. XML path language (XPath). `http://www.w3.org/TR/xpath`, November 1999. (Cited on page 83.)

[73] R. L. Wexelblat, editor. *History of programming languages*. ACM Press, 1981. (Cited on page 10.)

[74] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of OOPSLA*, volume 34(10) of *SIGPLAN Notices*, pages 187–206. ACM Press, 1999. (Cited on page 22.)

[75] M. V. Wilkes, D. J. Wheeler, and S. Gill. *The Preparation of Programs for an Electronic Digital Computer (Charles Babbage Institute Reprint)*. MIT Press, 1984. (Cited on page 10.)

[76] D. Willis, D. J. Pearce, and J. Noble. Efficient object querying for Java. In *Proceedings of ECOOP*, volume 4067 of *Lecture Notes in Computer Science*, pages 28–49. Springer Verlag, 2006. (Cited on pages 26, 138, and 139.)

[77] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994. (Cited on pages 52 and 101.)