Ownership and Immutability in Coq

by

Julian Mackay

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Master of Science
in Computer Science.

Victoria University of Wellington
2013

Abstract

A significant issue in modern programming languages is unsafe aliasing. Modern type systems have attempted to address this in two prominent ways; immutability and ownership, and often a combination of the two [4][17].

The goal of this thesis is to formalise Immutability and Ownership using the Coq Proof Assistant, a formal proof management system [13]. We encode three type systems using Coq; Featherweight Immutable Java, Featherweight Generic Java and Featherweight Ownership Generic Java, and prove them sound. We describe the challenges presented in encoding immutability, ownership and type systems in general in Coq.

Acknowledgments

Thanks go to Alex Potanin and Lindsay Groves for their guidance and encouragement. Hannes Mehnert gave much needed insight into the intricacies Coq.

Contents

1 Introduction **1**

1.1 Type Systems 1
1.2 Immutability 2
1.3 Ownership 4
1.4 Soundness 5
1.5 Type Systems in Coq 6
1.6 Overview 6

2 Background **9**

2.1 Featherweight Java 9
2.1.1 FJ Syntax 9
2.1.2 FJ Functions 11
2.1.3 FJ Subtyping 12
2.1.4 FJ Type Rules 13
2.1.5 FJ Reduction Rules 14
2.1.6 FJ Soundness 15
2.2 Coq 15
2.2.1 A Coq Primer 15

3 Featherweight Immutable Java **25**

3.1 Featherweight Immutable Java 25
3.2 FIJ Type System 29
3.2.1 FIJ Syntax 29

3.2.2 FIJ Substitution 33
3.2.3 FIJ Functions 34
3.2.4 FIJ Subtyping and Well-Formedness 38
3.2.5 FIJ Expression Typing 40
3.2.6 FIJ Reduction 49
3.3 FIJ Soundness 52

4 Featherweight Generic Java **59**
4.1 FGJ 59
4.2 FGJ Type System 61
4.2.1 FGJ Syntax 61
4.2.2 FGJ Substitution 64
4.2.3 FGJ Functions 66
4.2.4 FGJ Subtyping and Well-Formedness 73
4.2.5 FGJ Expression Typing 75
4.2.6 FGJ Reduction 81
4.3 FGJ Soundness 85
4.3.1 Substitution 85
4.3.2 Preservation 89
4.3.3 Progress 91

5 Featherweight Ownership Generic Java **93**
5.1 FOGJ 93
5.2 FOGJ Type System 95
5.2.1 FOGJ Syntax 95
5.2.2 FOGJ Substitution 99
5.2.3 FOGJ Functions 100
5.2.4 FOGJ Subtyping and Well-Formedness 104
5.2.5 FOGJ Expression Typing 108
5.2.6 FOGJ Reduction 116
5.3 FOGJ Soundness 117
5.3.1 Type Substitution 118

5.3.2 FOGJ Soundness 118

6 Conclusion **121**

6.1 Related Work 121

6.2 Encoding a Type System in Coq 122

6.3 Future Work 126

6.3.1 Readonly References 126

6.3.2 Method Guards 127

6.3.3 Featherweight Ownership Immutability Generic Java 128

6.4 Conclusion 129

Chapter 1

Introduction

1.1 Type Systems

When writing software, it is necessary to be sure that the software behaves as expected. Moreover, we wish to be sure that software is safe and predictable, while still being fairly flexible. In order to prevent undesirable behaviour, we need some way of restricting the behaviour of programs so that certain errors are excluded. Programs can be checked for errors statically before execution where unsafe programs are discarded, or dynamically where unsafe programs throw an exception. Dynamic checking is necessary in most languages since there are often errors that cannot be caught statically such as out of bound array accesses [2].

A type system is a formal system used for reasoning about and ensuring program correctness statically [12]. Pierce [12] provides the following definition:

> "A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute."

That is, a type system is a formal set of constraints that seeks to remove unexpected errors from programs that conform to those constraints. This is

generally done by differentiating language expressions by types, and specifying allowable behaviour for each type. Type restrictions can be used to restrict an operation to arguments that support that operation. There are a limited number of errors that can be detected statically due to the level of reasoning available. The example below is a common example of such a limitation.

```
1 if(1 > 0){print "safe";}
2 else{
3  <some type error>
4 }
```

It is obvious that `1 > 0` will always hold, and so the above code fragment is safe, however a type system may not be able to infer this and so the code would fail type checking due to the potential type error. Due to these restrictions, type systems generally tend to be conservative, and many programs may be rejected by a type system that are technically safe.

While at the most basic level type systems can be used to prevent critical errors from occurring, they can also be used to control the way variables and objects are accessed or modified, providing more functionality and control to a programmer. Using types, we can further restrict behaviour that while not resulting in a critical error would be undesirable.

1.2 Immutability

A common problem in writing safe programs is aliasing [9]. Aliasing refers to objects having multiple references. Unsafe aliasing can lead to objects being modified without the knowledge of objects that rely on that object. This can lead to unpredictable program behaviour and even security violations. There are existing mechanisms to protect data but these are often flawed. In Java the **`private`** keyword is the primary method for protecting internal data, however a careless programmer is still able to

expose or change a **private** field by providing a getter or setter method. An example is given below.

```
1 class foo extends Object{
2   private Object f;
3   foo (Object f){this.f=f;}
4   Object exposef (){
5     return this.f;
6   }
7 }
```

Field `f` has been exposed by the method `exposef`. The programmer may then wrongly believe that `f` is protected from aliasing, and would not prepare for the possibility that an external reference may be used to change the value of `f`.

Immutability allows for safe aliasing of objects. Some objects may be declared *immutable* on initialization. An *immutable* object may not be modified. Changing the previous example to use immutability rather than **private** gives us the following:

```
1 class foo extends Object{
2   Object<immutable> f;
3   foo (Object f){this.f=f;}
4   Object<immutable> exposef (){return this.f;}
5 }
```

As before, `f` has been exposed, but now it cannot be modified. This protects the field `f` from unsafe aliases. Immutability comes in several varieties that all restrict the modification of objects in some way. This is usually implemented through type information, i.e. an immutable object is of an immutable type. The three most common forms of immutability are given below.

- *Object Immutability*: An instance of class may be annotated as `immutable`.

Immutable instances may not be modified, while non-immutable instances behave as normal [16][10].

- *Class Immutability*: A class may be annotated as `immutable`. All instances of an `immutable` class are immutable [16][10]. An object may not be modified via a readonly reference, but might be modified via a non-readonly reference [16][15][10].

Each of these kinds of immutability provide some level of assurance that aliased objects will not be incorrectly modified with varying levels of flexibility. All three of these are implemented through type information. Unlike **`private`** fields an immutable object will always have an immutable type.

1.3 Ownership

While immutability attempts to allow for aliasing, but mitigate the effects of unintended or bad aliasing, ownership is an attempt to enforce encapsulation of objects, and restrict aliasing by introducing the concept of an owner. In Object Oriented languages, data is abstracted by the concept of objects. Objects have various characteristics and structure as well as relationships with other objects. For example a `Student` object uses a `Pen` object to write. It would seem natural to be able to say that a certain `Student` "owns" a certain `Pen`. That is, that `Pen` may only be used by that `Student`. Ownership allows objects to be owned by other objects.

If an object is owned by another object then references or mutations to the first object are controlled in some manner by the second. How strict the restrictions on references to or mutations of an owned object are depends on the form of ownership used, and how it is enforced. Below are a few examples of the more common forms of ownership.

- *Owners as Dominators*: References to objects are strictly limited to an object's owner. This is the strictest form of ownership [3].

- *Owners as Modifiers*: Objects can only be modified by their owners, but can be referenced by any object [4].

Enforcing encapsulation allows us to restrict access to sensitive parts of an object's structure. For example, a list can be created that owns its structure, but does not own its elements. Restrictions like this can prevent careless programming errors that can expose a list's state, and potentially break code, while also allowing access to the elements.

Extending the ownership metaphor to types allows us to enforce ownership restrictions through a type system. For a `Pen p` owned by a `Student s`, we can say `p` is of type `Pen<s>`. Now we can specify that `s` doesn't just need a `Pen` to write, but `s` needs his pen to write, or a pen of type `Pen<s>`.

1.4 Soundness

If the purpose of a type system is to prevent certain behaviour from occurring, we would like to be able to prove that for any given program that conforms to the type system, evaluation will not result in any unexpected errors and will not get stuck. This property is called *Soundness*. A type system can be formally expressed as a calculus by defining the syntax of the language and the type constraints (or rules) mathematically. If we can express the type system mathematically, then we can construct theorems to prove the type system sound or not.

Type soundness is important when extending a type system with functionality such as immutability or ownership, because we want to be sure that added type restrictions do not result in unpredictable behaviour, and our type system does what it says on the box.

In the case of immutability for example, we need to be sure that the type constraints of such a type system result in programs that do not allow modifications to immutable objects, this would be an immutability guarantee. In the case of ownership, we need to be sure that type safe

modifications to an object are only done by that object's owner, an ownership guarantee.

1.5 Type Systems in Coq

Coq is a proof assistant based upon the calculus of inductive constructions. Coq can be used to define various mathematical concepts and subsequently prove properties of those concepts in theorems. Coq therefore has direct application to the world of type systems. Since a type system is a formal system, the type constraints of that type system, the elements of the language and the evaluation of those elements can be represented mathematically.

Once a type system has been expressed in Coq, we can reason about the type system by formulating a series of proofs. Specifically, we are able to prove the type system sound or not. Since soundness is such an important property of a type system it is essential that the proof of soundness be correct. Coq provides a level of rigor in proofs that is not available with pen and paper by providing a framework for proofs, and checking that each step in a proof is correct.

1.6 Overview

This thesis presents the following three type systems and their Coq encodings.

1. Featherweight Immutable Java (FIJ): A type system featuring a variant of immutability.
2. Featherweight Generic Java (FGJ): A type system featuring generic types.
3. Featherweight Ownership Generic Java (FOGJ): A type system that builds upon FGJ with a variant of ownership.

We prove these type systems sound with Coq, providing added insurance to programs written in these languages. We also provide a basis for encodings of type systems featuring different forms of immutability and ownership.

Chapter 2 provides a background to the type systems and Coq in later chapters. Chapter 3 presents Featherweight Immutable Java [8] (FIJ), an extension of Featherweight Java [6] with Immutability and its Coq encoding. Chapter 4 presents a modified Featherweight Generic Java [6] (FGJ) and its Coq encoding. Chapter 5 presents Featherweight Ownership Generic Java (FOGJ), an extension of FGJ with Ownership, and its Coq encoding. Chapter 6 provides a discussion of these type systems and their respective Coq encodings.

Chapter 2

Background

2.1 Featherweight Java

The basis of all formal systems described in this thesis is that of Featherweight Java (FJ)[6], a lightweight, minimal core calculus of the Java language and type system. The FJ syntax is limited to new expressions, field accesses, method invocations and casts. This tiny subset of Java provides an easy platform from which to reason about and extrapolate to the larger and far more complex Java type system. It is useful to start with a minimal calculus when trying to prove a type system sound since larger type systems usually contain mechanisms extraneous to what is being investigated. As casts are extraneous to the language properties being investigated in this thesis, they are left out of all type systems discussed in later Chapters. This section gives a brief overview of the FJ type system without casts and how it is structured. FJ without casts will be referred to as FJ in this and later Chapters for ease, although that name is not strictly correct.

2.1.1 FJ Syntax

Since a type system is applicable to a language, a type system needs a syntax to represent the various components of that language. The syntax

```
e ::= x | e.f | e.m(e) | new C(e)
v ::= new C(e)
M ::= C m (C x){return e;}
K ::= C(C x){super(x); this.f=x;}
L ::= class C extends C{C f; K M}
Pr ::= L; e
```

Figure 2.1: FJ Syntax

for FJ is given in Figure 2.1. The FJ syntax is made up of expressions (e), values (v), method declarations (M), constructors (K), class declarations (L) and programs (Pr). C, D and E refer to class names, and the convention of an over bar is used to denote a list of something, e.g. $\overline{C}$ is a list of classes. Types in FJ are just classes, and so C and D also represent FJ types. Expressions are the most basic components of a program. An FJ expression may be a variable (x), a field access (e.f), a method invocation (e.m($\overline{e}$)) or a new expression (new C($\overline{e}$)). These expressions are often made up of sub-components such as sub-expressions, method or field names and class names. Method names are given by m and field names as f. Values are those expressions that are irreducible. Values in FJ are new expressions. All methods are defined using method declarations. An FJ method declaration C m ($\overline{C}$ $\overline{x}$){return e;}, gives the return type of the method call C, the name of the method m, the list of parameters and their types $\overline{C}$ $\overline{x}$ and the body of the method e. An FJ constructor method K of a class C takes a list of parameters corresponding to the fields of C, and assigns them to those fields. The body of the constructor makes use of the super type's constructor by calling the super() method. Classes in FJ are defined using a class declaration L. A class declaration consists of a class, its super type, a list of field names and their type, a constructor and a list of method declarations. Finally, an FJ program Pr consists of a list of class declarations (a class table), and an expression.

$$fields(\texttt{Object}) = \emptyset \quad \text{(F-OBJECT)}$$

$$\frac{\texttt{class C extends D}\{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\}}{fields(\texttt{C}) = \overline{\texttt{C}}\ \overline{\texttt{f}} \cup fields(\texttt{D})}\ \text{(F-CLASS)} \qquad \frac{\texttt{D f} \in fields(\texttt{C})}{ftype(\texttt{C}, \texttt{f}) = \texttt{D}}$$

$$\frac{\texttt{class C extends D}\{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \qquad \texttt{C}_0\ \texttt{m}\ (\overline{\texttt{D}}\ \overline{\texttt{x}})\{\texttt{return e}_0;\} \in \overline{\texttt{M}}}{mbody(\texttt{C}, \texttt{m}) = (\overline{\texttt{x}}, \texttt{e}_0)}\ \text{(MB-CLASS)}$$

$$\frac{\texttt{class C extends D}\{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \qquad \texttt{m} \notin \overline{\texttt{M}}}{mbody(\texttt{C}, \texttt{m}) = mbody(\texttt{D}, \texttt{m})}\ \text{(MB-SUPER)}$$

$$\frac{\texttt{class C extends D}\{\overline{\texttt{T}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \qquad \texttt{C}_0\ \texttt{m}\ (\overline{\texttt{e}})\{\texttt{return e}_0;\} \in \overline{\texttt{M}}}{mtype(\texttt{C}, \texttt{m}) = \overline{\texttt{C}} \to \texttt{C}_0}\ \text{(MT-CLASS)}$$

$$\frac{\texttt{class C extends D}\{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \qquad \texttt{m} \notin \overline{\texttt{M}}}{mtype(\texttt{C}, \texttt{m}) = mtype(\texttt{D}, \texttt{m})}\ \text{(MT-SUPER)}$$

Figure 2.2: FJ Field and Method Retrieval Functions

2.1.2 FJ Functions

In order to make judgments about types and expressions, information about them needs to be retrieved. In FJ this would be information about fields and method from the class table, but in another type system other functions may be defined that retrieve relevant information about types and expressions. The functions used in FJ are found in Figure 2.1.2. The first function is the $fields$ function. $fields(\texttt{C})$ returns the field names and types of type `C`. The base case, `Object`, has no types (F-OBJECT). For all other classes `C`, $fields(\texttt{C})$ returns the fields defined in the class declaration of `C`, as well as those of the super class of `C` (F-CLASS). $ftype$ makes use of the $fields$ function, and returns the type of a specific field `f` for a type `C`. Method body retrieval is handled by $mbody$, which returns the body and

$$\texttt{C <: C} \quad \text{(S-REFL)} \qquad \frac{\texttt{C <: D} \qquad \texttt{D <: E}}{\texttt{C <: E}} \quad \text{(S-TRANS)}$$

$$\frac{\texttt{C extends D}}{\texttt{C <: D}} \quad \text{(S-EXTEND)}$$

Figure 2.3: FJ Subtyping Rules

parameter variables of a method for a given type. $mbody(\texttt{C}, \texttt{m})$ will return $(\overline{\texttt{x}}, \texttt{e})$ in one of two cases; if method $\texttt{m}$ with parameters $\overline{\texttt{x}}$ and body $\texttt{e}$ is defined in the class declaration of $\texttt{C}$ (MB-CLASS), or if $mbody(\texttt{D}, \texttt{m}) = (\overline{\texttt{x}}, \texttt{e})$, where $\texttt{D}$ is the super type of $\texttt{C}$ (MB-SUPER). The $mtype$ function retrieves the type of a method for a type. $mtype$ acts in a similar way to $mbody$, i.e. $mtype(\texttt{C}, \texttt{m}) = \overline{\texttt{C}} \rightarrow \texttt{C}_0$ if method $\texttt{m}$ is defined in the class declaration of $\texttt{C}$ with parameter type $\overline{\texttt{C}}$, and return type $\texttt{C}$ (MT-CLASS), or if $mtype(\texttt{D}, \texttt{m}) = \overline{\texttt{C}} \rightarrow \texttt{C}_0$, where $\texttt{D}$ is the super type of $\texttt{C}$ (MT-SUPER).

2.1.3 FJ Subtyping

In typed programming languages, *subtyping* refers to the type substitutability property between types. If one type $\texttt{S}$ *subtypes* another type $\texttt{T}$, then we can treat $\texttt{S}$ in the same manner as we would treat $\texttt{T}$. This relation is captured in type systems by the introduction of a subtype judgment. In FJ, types are synonymous with classes. When referring to classes, subclasses are used to extend the properties of one class to other classes by inheritance. That is, if one class subclasses another, it inherits the properties of the first class. Therefore, in FJ, subtyping is done by subclassing. Figure 2.1.3 gives the subtyping relation for FJ. The judgment $\texttt{C <: D}$ is read as $\texttt{C}$ is a subtype of $\texttt{D}$. Subtyping is reflexive, i.e. a type $\texttt{C}$ is a subtype of itself (S-REFL). Subtyping is also transitive, i.e. if $\texttt{C}$ subtypes $\texttt{D}$, and $\texttt{D}$ subtypes $\texttt{E}$, then $\texttt{C}$ also subtypes $\texttt{E}$ (S-TRANS). The basic form of subtyping is one

$$\frac{\Gamma(\mathtt{x}) = \mathtt{C}}{\Gamma \vdash \mathtt{x} : \mathtt{C}} \text{ (T-VAR)} \qquad \frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0 \qquad \mathit{ftype}(\mathtt{C}_0, \mathtt{f}) = \mathtt{C}}{\Gamma \vdash \mathtt{e}_0.\mathtt{f} : \mathtt{C}} \text{ (T-FIELD)}$$

$$\frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{C}_0 \qquad \mathit{mtype}(\mathtt{C}_0, \mathtt{m}) = \overline{\mathtt{C}} \to \mathtt{C} \qquad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{D}} \qquad \overline{\mathtt{D}} <: \overline{\mathtt{C}}}{\Gamma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) : \mathtt{C}} \text{ (T-INVK)}$$

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}} \qquad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{D}} \qquad \overline{\mathtt{D}} <: \overline{\mathtt{C}}}{\Gamma \vdash \mathtt{new}\ \ \mathtt{C}(\overline{\mathtt{e}}) : \mathtt{C}} \text{ (T-NEW)}$$

Figure 2.4: FJ Typing Rules

class extending another, i.e. if C is declared to extend D, then C subtypes D (S-EXTEND).

2.1.4 FJ Type Rules

The central aspect of the FJ type system is the typing of expressions. Since a program is just a series of expressions, how expressions are typed, and whether or not that typing is able to ensure any level of safety is critical to any type system. The typing judgment for FJ is given in Figure 2.1.4. An expression e is said to be well-typed with respect to a type C in the context of an environment Γ if $\Gamma \vdash \mathtt{e} : \mathtt{C}$ holds. An environment provides a mapping from expression variables to types. As with previous rules, typing is presented in a case by case basis for each expression. A variable x has type C if Γ maps x to C (T-VAR). A field access $\mathtt{e}_0.\mathtt{f}$ has type C if $\mathtt{e}_0$ has type $\mathtt{C}_0$, and f has type C in $\mathtt{C}_0$ (T-FIELD). A method invocation $\mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}})$ has type C if $\mathtt{e}_0$ has type $\mathtt{C}_0$, m has a return type of C in $\mathtt{C}_0$, and the parameters $\overline{\mathtt{e}}$ have types $\overline{\mathtt{D}}$ that subtype the parameters of m, $\overline{\mathtt{C}}$ (T-INVK). A new expression $\mathtt{new}\ \ \mathtt{C}(\overline{\mathtt{e}})$ has type C if $\overline{\mathtt{e}}$ have types $\overline{\mathtt{D}}$ which subtype the field types of C (T-NEW).

$$\frac{fields(\texttt{C}) = \overline{\texttt{C f}}}{(\texttt{new C}(\overline{\texttt{v}})).\texttt{f}_i \longrightarrow \texttt{v}_i} \quad \text{(R-FIELD)}$$

$$\frac{mbody(\texttt{m}, \texttt{C}) = (\overline{\texttt{x}}, \texttt{e}_0)}{(\texttt{new C}(\overline{\texttt{v}}_1)).\texttt{m}(\overline{\texttt{v}}_2) \longrightarrow [\overline{\texttt{v}}_2/\overline{\texttt{x}}, (\texttt{new C}(\overline{\texttt{v}}_1))/\texttt{this}]\texttt{e}_0} \quad \text{(R-INVK)}$$

Figure 2.5: FJ Reduction Rules

$$\frac{\texttt{e}_0 \longrightarrow \texttt{e}_0'}{\texttt{e}_0.\texttt{f}_i \longrightarrow \texttt{e}_0'.\texttt{f}_i} \quad \text{(RC-FIELD)} \qquad \frac{\texttt{e}_0 \longrightarrow \texttt{e}_0'}{\texttt{e}_0.\texttt{m}(\overline{\texttt{e}}) \longrightarrow \texttt{e}_0'.\texttt{m}(\overline{\texttt{e}})} \quad \text{(RC-INVK-RECV)}$$

$$\frac{\texttt{e}_i \longrightarrow \texttt{e}_i'}{\texttt{e}_0.\texttt{m}(..., \texttt{e}_i, ...) \longrightarrow \texttt{e}_0.\texttt{m}(..., \texttt{e}_i', ...)} \quad \text{(RC-INVK-ARG)}$$

$$\frac{\texttt{e}_i \longrightarrow \texttt{e}_i'}{\texttt{new C}(..., \texttt{e}_i, ...) \longrightarrow \texttt{new C}(..., \texttt{e}_i', ...)} \quad \text{(RC-NEW-ARG)}$$

Figure 2.6: FJ Context Reduction Rules

2.1.5 FJ Reduction Rules

The reduction of expressions is represented by the reduction rules in Figure 2.1.5. Expression $\texttt{e}_1$ reduces to expression $\texttt{e}_2$ if $\texttt{e}_1 \longrightarrow \texttt{e}_2$ holds. A field access $\texttt{new C}(\overline{\texttt{v}}).\texttt{f}_i$ reduces to $\texttt{v}_i$ (R-FIELD). A method invocation $\texttt{new C}(\overline{\texttt{v}}_1).\texttt{m}(\overline{\texttt{v}}_2)$, where $\texttt{m}$ has body $(\overline{\texttt{x}}, \texttt{e}_0)$ in class $\texttt{C}$, reduces to $\texttt{e}_0$ with the method parameters $\overline{\texttt{v}}_2$ substituted for the parameter variables $\overline{\texttt{x}}$, and $\texttt{new C}(\overline{\texttt{v}}_1)$ substituted for the $\texttt{this}$ variable. Reduction of expressions may also occur by by the reduction of subexpressions. These are captured in Figure 2.1.5 as the context reduction rules.

2.1.6 FJ Soundness

Soundness in a type system is can be summed up as: If $\Gamma \vdash \mathtt{e} : \mathtt{C}$ holds for environment Γ, expression $\mathtt{e}$ and type $\mathtt{C}$, then the evaluation of $\mathtt{e}$ will never result in an unexpected error. To express this property, we break it up in to two components; *Preservation* and *Progress*. The statement of *Preservation* is given below in Theorem 2.1.1.

Theorem 2.1.1 (Preservation)**.** *If* $\Gamma \vdash e : C$ *and* $e \longrightarrow e'$*, then* $\Gamma \vdash e' : C'$ *and* $C' <: C$

Proof. By induction on the derivation of $\mathtt{e} \longrightarrow \mathtt{e}'$. □

Preservation requires that for a given well-typed expression, any reduction will result in another well-typed expression that subtypes the type of the original expression. *Progress* completes soundness and is presented below in Theorem 2.1.2.

Theorem 2.1.2 (Progress)**.** *If* $\Gamma \vdash e : C$ *then either*

(i) e *is a value, or*

(ii) $\exists e'$ *s.t.* $e \longrightarrow e'$

Proof. By induction on the derivation of $\Gamma \vdash \mathtt{e} : \mathtt{C}$. □

Progress requires that for a well-typed expression, either that expression is an irreducible value, or it can "make progress", and can be reduced. In other words, no well-typed expression will get stuck.

2.2 Coq

2.2.1 A Coq Primer

This section gives an overview of Coq basics. The definitions and tactics described here are only intended to help understand the type systems and

their respective proofs in later chapters. For a more complete description of these and other Coq definitions and tactics the book Coq'Art [1] is a good resource.

Coq Definitions

All Coq definitions fall into one of two categories, `Prop` or `Type` [13]. `Prop` refers to propositions while definitions that fall into the `Type` category are data types. The type systems in the following chapters are all constructed using a combination of the following Coq definitions. The first and most common definition type used is the `Inductive` definition. An example of the `Inductive` definition is given below.

```
Inductive nat : Type :=
  | O : nat
  | S : nat -> nat.
```

`nat` is the Coq representation of natural numbers. An inductive definition is made up of two parts, the header and the body. The header gives the definition's name and what type of Coq object it is. In this case the name is `nat` and the type is `Type`. The body provides the ways in which an object can be constructed. There are two ways to construct a Coq object of type `nat`, `O` and `S`. `O` corresponds to 0 in the natural number system, and `S` to the successor function, i.e. a `nat` may either be `O` or `S n` where `n` is some `nat`. Using this scheme, `1=S O`, `2=S 1=S (S O)`, etc.

Another inductively defined Coq datatype is `list`. Lists in Coq are represented in the same manner as natural numbers, and is given below.

```
Variable A : Type.

Inductive list : Type :=
  | nil : list
  | cons : A -> list -> list.
```

```
Infix "::" := cons (at level 60, right associativity).
```

We start by declaring some variable type `A` using **`Variable A : Type`**. `A` will be used in the definition of `list` as a generic data type so we may create lists of different types. We can construct a list in one of two ways; `nil` and `cons`. `nil`, an empty list, is the base case, and is analogous to `O` in `nat`. `cons a l` (where `a` is of type `A`, and `l` is a list of elements of type `A`) is the Inductive case, and is analogous to `S` in `nat`. `cons a l` appends `a` to the front of `l`. We can construct the list `[a1, a2, a3]` as `(cons a1 (cons a2 (cons a3 nil)))`. We also define `::` as short hand for `cons`. Therefore, `[a1, a2, a3] = (a1::(a2::(a3::nil)))`.

An `Inductive` definition may also be of type `Prop`. Below is an example of such a definition, `lt`, a predicate that determines if one natural is less that of another.

```
Inductive lt: nat -> nat -> Prop :=
  | lt_O : forall n,
           lt O (S n)
  | lt_S : forall n m,
           lt n m ->
           lt (S n) (S m).
```

`lt` is of type `nat -> nat -> Prop`, i.e. given two objects of type `nat`, `n` and `m`, `lt n m` has type `Prop`. The body provides two cases where `lt n m` holds. `lt_O` and `lt_S`. `lt_O` says that `O` is less than `S n` for all `n`. `lt_S` says that given two naturals `n` and `m`, `lt n m` implies the same for their successors, i.e. `lt (S n) (S m)`. Using `lt` we can say that `2` is less than `3`. `2 = S (S O)` and `3 = S (S (S O))`. Therefore `lt (S (S O)) (S (S (S O)))` holds if `lt (S O) (S (S O))` holds by `lt_S`. Similarly `lt (S O) (S (S O))` holds if `lt O (S O)` holds, also by `lt_S`. Finally, `lt O (S O)` holds by `lt_O`.

`Inductive` definitions can also be defined mutually if they both depend on each other. The predicate `even` defined below holds if a supplied

natural is even. `even` is defined along with `odd`. `odd` is a predicate that holds if a supplied natural is odd, similarly `odd` is defined using `even`.

```
Inductive even : nat -> Prop :=
  | even_O : even O
  | even_S : forall n,
              odd n ->
              even (S n)
with odd : nat -> Prop :=
  | odd_S : forall n,
             even n ->
             odd (S n).
```

A natural number `n` is even if either `n = O` (`even_O`), or the predecessor of `n` is odd (`even_S`). A natural `n` is odd if its predecessor is even (`odd_S`).

Coq objects can also be defined using previously defined predicates with the `Definition`.

```
Definition leq (n m : nat) := (lt n m) \/ (n = m).
```

The predicate `leq` determines if one natural is less than or equal to another by making use of the previously defined `lt`. `gt n m` holds if `n` is less than `m` (`lt n m`) or if `n` equals `m` (n = m). These two propositions are joined by a disjunction (`\/`).

Recursive functions can also be defined in Coq, using the a `Fixpoint` function definition.

```
Fixpoint sum (n m : nat) : nat :=
  match n with
  | O => m
  | S n' => S (sum n' m)
end.
```

The function `sum` is defined above, and takes two naturals, `n` and `m`, as inputs and returns their sum. The first input `n` is matched with one of two

cases, `O` and `S n'` for some `n'`. If `n = O`, then `sum n m` returns `m`. If `n = S n'`, then `sum n m` recursively calls `sum`, and returns `S (sum n' m)`. In the second case, eventually recursion will end when `n = O`, and at that point the result will be `m + 1 + 1 + 1 + ...` (`n` times) = `m + n`. The end of the function is marked by `end`.

Coq Proofs

Theorems and their proofs in Coq generally follow a simple structure that is outlined here. A Coq theorem is composed of three parts, a name that is used to reference it, a statement of the theorem, i.e. what it is we wish to prove, and finally the proof. The basic structure is given below.

```
Theorem <name> :
  <statement of theorem>
Proof.
  <proof>
Qed.
```

An example is given below: a simple proof showing that $\forall n \in \mathbb{N}, n < (Sn)$.

```
Theorem lt_n_S : forall (n : nat),
  lt n (S n).
Proof.
  intro n.
  induction n as [|n'].
  Case "n = O".
    apply lt_O.
  Case "n = S n'".
    apply lt_S.
    assumption.
Qed.
```

The theorem begins with the name of the theorem, `lt_n_S`, and follows with the statement of the theorem **`forall`** `n, lt n (S n)`. In other words the theorem `lt_n_S` states that for all `n` of type `nat`, `n` is less that its successor `S n`. The proof of a theorem makes use of a series of tactics that act upon the goals or the premises of a proof in order to resolve the goals. Each goal represents a separate case to be resolved. In this primer and throughout the encodings presented in this thesis, we make use of a Coq tactic from the Software Foundations course [11]. Before any tactics are applied entering the theorem above gives the following set of premises and goals.

```
1 subgoals
_____________________________________(1/1)
forall n : nat, lt n (S n)
```

To begin with we need to introduce some `n`. To do this we apply the tactic `intro n`.

```
1 subgoals
n : nat
_____________________________________(1/1)
lt n (S n)
```

Now a `n` of type `nat` has been introduced as a premise. Since `n` is an inductive type, we can proceed by induction on `n`. So we say in order to prove `lt_n_S` holds for some general `n` we need to prove that it first holds for `O` and then show that if it holds some `n'`, it follows that it holds for `S n'`. In order to do this we use the tactic `induction n as [|n']`.

```
2 subgoal
_____________________________________(1/2)
lt 0 1
_____________________________________(2/2)
lt (S n') (S (S n'))
```

The original goal has now been replaced by two subgoals; one for `n = 0`, and one for `n = S n'`. The qualifier `[|n']` allows us to specify names for variables used in each case. In the second case we want the variable `n'` to be used. The first goal can be easily derived from the definition of `lt` from the previous section. `lt` has two cases `lt_O` and `lt_S`. `lt_O` states that for all `n`, `lt O (S n)` holds. Since `1 = S 0`, `lt 0 1` holds by `lt_O`. We can use this in our proof by using the tactic `apply lt_O`.

```
1 subgoals
n' : nat
IHn' : lt n' (S n')
_____________________________________(1/1)
lt (S n') (S (S n'))
```

The first case has now been resolved, and we can move on to the next case. We are now trying to show that for some `n'`, it follows that if `lt_n_S` holds for `n'`, it also holds for `S n'`. Since this is the inductive case, we start with the inductive hypothesis `IHn' : lt n' (S n')`. We are now trying to show that `lt (S n') (S (S n'))` holds. As with the previous goal, we can make use if the definition of `lt`. This time we can use `lt_S` instead of `lt_O`. `lt_S` states that for all `n` and `m`, `lt n m` implies that `lt (S n) (S m)` holds. We can apply this to our current goal with the tactic `apply lt_S`.

```
1 subgoals
n' : nat
IHn' : lt n' (S n')
_____________________________________(1/1)
lt n' (S n')
```

Applying `lt_S` means we still have to prove that `lt n' (S n')` holds. However we already know this is true by the induction hypothesis `IHn'` mentioned before. Since this is already assumed, we can apply the tactic `assumption` to resolve the final goal. To complete and save the proof for,

later reference we can use the keyword `Qed`.

Induction is a very commonly used proof tactic in Coq due to the use of `Inductive` definitions. Usually using induction is a simple case of applying induction to a premise as we did in the previous example, however when dealing with mutually defined definitions like `even` and `odd`, we need to define the induction scheme for them mutually. Below is a mutually defined induction scheme for `even` and `odd`. First we define `even_induct` and `odd_induct` as inductive definitions for `even` and `odd` respectively. Second, we instruct Coq to use them together by defining them as `even_odd_mutind`.

```
Scheme even_induct := Minimality for even Sort Prop
with odd_induct := Minimality for odd Sort Prop.

Combined Scheme even_odd_mutind from even_induct,
odd_induct.
```

Now, when we have a theorem that requires induction on both `even` and `odd`, we can apply `even_odd_mutind` in order to resolve it. Below is an example that makes use of the above mutually inductive scheme. `sum_of_odd_even_mutind` states that the sum of two even numbers is even, and the sum of an even and an odd number is odd. These two separate statements would be impossible to prove separately given the way we previously defined `even` and `odd`. If they had not been defined mutually, we might be able to proceed by regular induction with both separately, but since both definitions make use of each other we need to prove each these statements in order to prove the other.

```
Theorem sum_of_odd_even_mutind :
  (forall n, even n -> (forall m, even m ->
    even (sum n m))) /\
  (forall n, odd n -> (forall m, even m ->
    odd (sum n m))).
```

```
Proof.
  apply even_odd_mutind; intros.
  Case "even_O".
    simpl. assumption.
  Case "even_S".
    simpl. apply even_S.
    apply H0. assumption.
  Case "odd_S".
    simpl. apply odd_S.
    apply H0. assumption.
Qed.
```

Applying `even_odd_mutind` to the initial goal results in three cases, one for each inductive case of `even` and `odd`. These cases are resolved fairly easily in the same way we resolved the goals from `lt_n_S`. Another tactic introduced in this theorem is `simpl`. `simpl` just simplifies a goal or premise, for example a goal of `sum (S n) m` would be simplified to `S (sum n m)` by the straight forward application of the `sum` function.

Chapter 3

Featherweight Immutable Java

3.1 Featherweight Immutable Java

Featherweight Immutable Java (FIJ) extends FJ with immutability. As discussed in Chapter 1, immutability as we wish to define it, requires field assignment. This means we need to extend FJ by introducing a new assignment expression of the form $\texttt{e.f} = \texttt{e}'$, where expression $\texttt{e}'$ is being assigned to field `f` of receiver `e`. With assignment we need to introduce a store. A store represents the memory in our program, and contains all the objects that have been initialized. To refer to those objects we use locations. A location is an expression that points to an object within the store, and is used in place of that object. A store is contextual, only having meaning in relation to an expression, and is maintained and updated as that expression is reduced. Reduction in FIJ is extended from FJ to include a store. $\texttt{e}|\mathcal{H} \longrightarrow \texttt{e}'|\mathcal{H}'$ is the reduction of expression `e` and store $\mathcal{H}$ to expression $\texttt{e}'$ and store $\mathcal{H}'$.

Along with locations, we also introduce two more expressions; null expressions and errors. A null expression is an empty pointer, i.e. a pointer that does not point to an object. This allows us to initialize object with null fields, and only assign them later. A null expression is well typed with respect to all well-formed types. This means they are assignable to

all fields. While a null expression may be used anywhere, that does not change the fact that it is not an object, and so does not have fields or methods. Field and method calls on null expressions result in an error. An error expression is used when performing reductions on expressions with a null receiver, and in our type system will not be well-typed with respect to any type.

Immutability in FIJ is enforced by adding mutability information to types. All FIJ types are parameterized by a mutability parameter. An object of a type parameterized as mutable acts in the same way as a normal Java object, while an object of a type parameterized as immutable disallows field assignment.

All FIJ classes are parameterized with a mutability parameter. A class mutability parameter limits the mutability of the objects of that class. Mutability parameters fall in to one of three categories; mutable (`mut`), immutable (`imm`) or a mutability variable `X`. For a class `C` with mutability `I`, the mutability of all objects of class `C` must conform to `I`. If a class `C` is parameterized as `mut / imm`, then all objects of class `C` must be mutable / immutable. If `C` is parameterized by `X`, objects of class `C` may be initialized as either `mut` or `imm`. An example of some FIJ classes are provided below.

```
1 class C<imm> extends Object<imm>{}
2 class D<X> extends Object<X>{}
3
4 new C<mut>(); // results in a type error
5 new C<imm>(); // type checks
6 new D<imm>(); // type checks
7 new D<mut>(); // type checks
```

Parameterizing a class by a mutability variable means that mutability is defined at object initialization, this implies FIJ conforms to the *Object Immutability* discipline. Parameterizing classes with `mut` or `imm` on the other hand, provides for *Class Immutability* since all instances of a class

parameterized with `imm` are immutable.

In FIJ, immutable objects are restricted from field assignment, however the fields of an immutable object are not required to be immutable, and so may be mutated themselves. This means that the immutability of FIJ is on the surface *shallow*. This is useful if we want to make an objects that have immutable structure, but we still want to retain some flexibility in mutating fields. An example of this would be an immutable list which always contains the same elements, but those elements are not guaranteed to be immutable.

While on the surface the immutability of FIJ is *shallow*, using a mutability variable we can enforce *deep* immutability. If a class is parameterized with a mutability variable `X`, then `X` may be used throughout the class. That is, fields and methods may treat `X` as a defined mutability parameter. An example of such a class is given below.

```
1 class Cx<X> extends Object<X>{
2   Object<X> fx;
3   Cx<X> copy pure(){return new Cx<X>(this.fx);}
4   Cx<X> setfx mutating(Object<X> fx){return this.fx=fx;}
5 }
```

The field `fx` will have the same mutability as the instance of `Cx` it is declared in. The method `copy` can also use the mutability variable `X`. the return type of `copy` will depend on the mutability of the receiver.

The `copy` method in the above listing makes use of the **`pure`** and **`mutating`** keywords. In FIJ, a method may be annotated as **`pure`** if it does not mutate the object in anyway, otherwise it must be annotated as **`mutating`**. Only **`pure`** methods may be called on an immutable receiver. Type errors can arise from methods that mutate the receiver being called on an immutable receiver. Disallowing **`mutating`** method calls on immutable receivers prevents such type errors.

Notice in the previous code segment that there is no explicit construc-

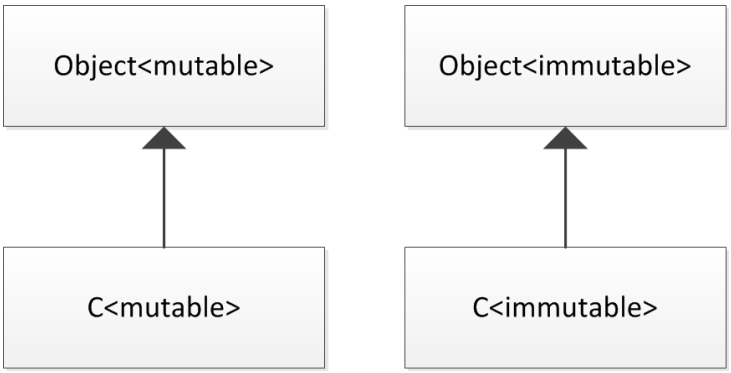

Figure 3.1: FIJ Type Hierarchy

tor for `Cx`. Constructors in FIJ are not explicitly declared, and the **`new`** expression takes a series of expression parameters as inputs and assigns them to the objects fields. This presents an issue during object initialization of immutable objects. If the fields of immutable objects can not be assigned to, then how are the fields initialized? During object initialization, the mutability of the object is not taken into account. This is the only time immutability restrictions are lifted. In all other cases assignment must respect the mutability of the receiver.

Subtyping in FIJ is complicated by the introduction of mutability parameters. It would seem natural that `Cx<mut>` should subtype `Cx <imm>` since in all cases an object of type `Cx<mut>` is substitutable for an object of type `Cx<imm>`. The problem arises when we want to be sure that an immutable object really is immutable. If we can assign a mutable object to an immutable field then we cannot rely on the immutable nature of any field we declare. IGJ [16] dealt with this by using a `readonly` mutability type that both `imm` and `mut` subtyped, but `mut` did not subtype `imm`. To maintain a level of simplicity, FIJ does not use a `readonly` mutability type, rather it uses a split type hierarchy. `Cx<mut>` subtypes `Object<mut>` and `Cx<imm>` subtypes `Object<imm>`, but `Cx<mut>` does not subtype `Cx<imm>`. This creates the type hierarchy seen in Figure 3.1. As mentioned in Chapter 1, alternative immutability variants [15] [16] [17]

$$
\begin{aligned}
\texttt{e} &::= \texttt{null} \mid \iota \mid \texttt{x} \mid \texttt{err} \mid \texttt{e.f} \mid \texttt{e.f} = \texttt{e} \mid \texttt{e.m}(\overline{\texttt{e}}) \mid \texttt{new T}(\overline{\texttt{e}}) \mid \texttt{e; e} \\
\texttt{v} &::= \texttt{null} \mid \iota \\
\texttt{I} &::= \texttt{mut} \mid \texttt{imm} \mid \texttt{X} \\
\texttt{P} &::= \texttt{mutating} \mid \texttt{pure} \\
\texttt{T} &::= \texttt{C<I>} \\
\texttt{M} &::= \texttt{T m P } (\overline{\texttt{T}}\ \overline{\texttt{x}})\{\texttt{return e;}\} \\
\texttt{L} &::= \texttt{class T extends T}\{\overline{\texttt{T}}\ \overline{\texttt{f}};\ \overline{\texttt{M}}\} \\
\texttt{Pr} &::= \overline{\texttt{L}}\texttt{; e}
\end{aligned}
$$

Figure 3.2: FIJ Syntax

include `readonly` as a mutability parameter. The inclusion of `readonly` completes the hierarchy, allowing references to be created that prohibit assignment, while not ensuring such restrictions from other references. Such a type system is clearly more descriptive, and while FIJ's type system is not ideal, it does allow FIJ to focus on the encoding of assignment to immutable types without complicating the type hierarchy.

The rest of this chapter is devoted to describing the FIJ type system and its Coq encoding in detail. Section 3.2 presents the FIJ type system and its Coq encoding. Section 3.3 describes the soundness proofs of the FIJ type system, and the Coq of those soundness proofs.

3.2 FIJ Type System

3.2.1 FIJ Syntax

Figure 3.2 gives the syntax of FIJ. An FIJ expression (`e`) may be a null expression (`null`), a location (ι), a variable (`x`), an error (`err`), a field access (`e.f`), a field assignment (`e.f = e`), a method invocation (`e.m`$(\overline{\texttt{e}})$), a new expression (`new T`$(\overline{\texttt{e}})$) or a sequence (`e; e`). A value (`v`) is either the null expression, or a location. A mutability parameter (`I`) may be either `mut`, `imm` or `X`, a mutability variable. All methods are annotated by a method

mutability (P). A method mutability annotation may either be mutating or pure. An FIJ type (T) is represented by a class name (C) parameterized by a mutability parameter. A method declaration (M) is given by a return type, a method name (m), a method mutability annotation, a list of method parameters ($\overline{\texttt{T}}\,\overline{\texttt{x}}$) and a method body (e). A class declaration (L) is given by a type (T) that extends a super type (T), a list of field names and their types ($\overline{\texttt{T}}\,\overline{\texttt{f}}$) and a list of declared methods ($\overline{\texttt{M}}$). Lastly, an FIJ program (Pr) is a list of class declarations ($\overline{\texttt{L}}$) and an expression (e). The list of class declarations in a program is referred to as a *class table*.

Now the Coq encoding of the FIJ syntax will be presented.The Coq definition of classes is given below (class). Classes are defined inductively with one of two constructors; Object and Extend. Object is the base class, and is analogous to the natural number 0. Extend is the inductive case, and defined with a unique identifier (ClassName) and a super class. Extend C D is read as "the class with ClassName C extends class D". Extend is analogous to the successor function S for natural numbers. The notation C extends D is also defined below as the notation for Extend C D.

```
Inductive ClassName : Type :=|Class : nat -> ClassName.
Inductive class : Type :=
  | Object : class
  | Extend : ClassName -> class -> class.
Notation "C 'extends' D" := (Extend C D) (at level 0).
```

Initially when encoding the FIJ type system, classes were defined in the same manner as ClassName, i.e. by simply using a natural number (nat) to distinguish between them. Classes were only later encoded inductively as they are above with the intention of making it easier to reason about the class hierarchy. If classes are defined as either Object, or the extension of some other class, it becomes quite easy to show that all classes extend Object. It also becomes easier to determine if one class extends another simply by the information contained in the class rather than hav-

ing to check with the artificial construct of the class table. One side effect was that within a class declaration the super type was split up in to class information (contained in the subclass) and a mutability parameter. This has no effect on FIJ since the structure of the type hierarchy (Figure 3.1) means that the mutability parameter of the super type will be the same as that of the subtype. In later encodings this was changed back to the simpler encoding (as in `ClassName`). With the introduction of multiple type parameters in generics and ownership, it became inconvenient to split up the parameters and the class information of super types.

Mutability parameters are defined below as `mutability`. Possible mutability parameters are `mutable`, `immutable` and `variable`. `mutable` and `immutable` correspond to `mut` and `imm`. `variable` is the Coq encoding of mutability variables. There can only ever be one mutability variable in scope at any one time, and therefore there will never be any ambiguity about mutability variables. Thus, for simplicity, all mutability variables are encoded as `variable`.

```
Inductive mutability : Type :=
  | mutable   : mutability
  | immutable : mutability
  | variable  : mutability.
```

Types are defined below as `ty`. A type is constructed using a class and a mutability parameter (`Ty C I`). The notation `C <<I>>` is used throughout the FIJ encoding instead of `Ty C I`.

```
Inductive ty : Type :=
  | Ty : class -> mutability -> ty.
Notation "C '<<' I '>>'" := (Ty C I) (at level 0).
```

Method mutability annotations are defined below as `meth_mut` in a similar fashion to type mutability parameters. They may be either `pure` or `mutating`.

```
Inductive meth_mut : Type :=
  | pure     : meth_mut
```

```
  | mutating : meth_mut.
```

Expressions are defined below as `exp`. `e_null` is the Coq encoding of a null expression. `e_loc` is the encoding of locations. `e_loc i` a location with position `i` in the store. `e_var` is the Coq encoding of variables, and is constructed using a unique variable identifier (`var`). `e_err` encodes an error expression. `e_field` encodes field accesses. `e_field e f` is a field access `f` on receiver `e`. `e_assign` encodes field assignments. `e_assign e f e'` assigns expression `e'` to field `f` in receiver `e`. `e_new` encodes new expressions. `e_new C I es` corresponds to `new C<I>(es)`, where `C` is a class, `I` is a mutability parameter, and `es` is a list of constructor parameters. `e_meth` is the encoding of method invocations. `e_meth e m es` is a method invocation `m` with parameters `es` on receiver `e`. `e_seq` is the encoding for sequences. `e_seq e1 e2` corresponds to `e1; e2`.

```
Inductive exp : Type :=
  | e_null    : exp
  | e_loc     : nat -> exp
  | e_var     : var -> exp
  | e_err     : exp
  | e_field   : exp -> field -> exp
  | e_assign  : exp -> field -> exp -> exp
  | e_meth    : exp -> meth -> list exp -> exp
  | e_new     : class -> mutability -> list exp -> exp
  | e_seq     : exp -> exp -> exp.
```

Method (`MethDecl`) and class (`ClassDecl`) declarations are defined below. A method declaration `mDecl m T0 P xs e0` declares a method with name `m`, return type `T0`, method mutability `P`, parameters `xs` and body `e0`. A class declaration `cDecl C I fs ms` is declares a class `C` with mutability parameter `I`, fields `fs` and methods `ms`. A `ClassTable` is defined as a list of class declarations (`ClassDecl`). Throughout the encoding a generic class table, `CT` is used. `CT` is defined below. This is done for simplicity so we do not require a class table parameter in every predicate. A common premise used in many predicates throughout all en-

codings in this thesis is `In (cDecl C I fs ms) CT`. This checks that a class declaration with class `C`, mutability `I`, fields `fs` and method declarations `ms` is in the generic class table `CT`. If `C = C0 extends D` (for some class name `C0` and class `D`), then this can be considered a Coq encoding of `class C<I> extends D{fs; ms}`

```
Inductive MethDecl : Type :=
  | mDecl : meth -> ty -> meth_mut ->
              list (var * ty) -> exp -> MethDecl.
Inductive ClassDecl : Type :=
  | cDecl : class -> mutability ->
              flds -> mths -> ClassDecl.
Notation ClassTable := (list ClassDecl).
Parameter CT : ClassTable.
```

A store is required for locations to be used. A store is a collection of locations, each containing an object. An object is represented as a type-expression list pair. The type is the type of the object, and the expression list is the list of expressions corresponding to the fields of the type. A store is thus a list of type-expression list pairs. A location `e_loc i`, described in `exp`, points to the `i`th position in the store.

```
Definition store := list (ty * (list exp)).
```

3.2.2 FIJ Substitution

When pen and paper proofs are done that involve substitution not much attention is paid to what happens during the substitution, and much is assumed about the functioning of the substitution. When these proofs are translated into a Coq script, Coq does not allow us to ignore these aspects of the proof. This section describes the encoding of expression substitution in FIJ.

When method calls are evaluated in FJ and FIJ, expression parameters are substituted for their corresponding parameter variables into the

method body. In Coq we need to encode a function that represents an expression substitution such as $[\overline{e}/\overline{x}]e$. First we need a way to represent the substitution relation $[\overline{e}/\overline{x}]$. In our encoding, `SubstRel` is defined as a list of variable-expression (`var-exp`) pairs. For a `R` of type `SubstRel`, each `(x,e)` (`v` of type `var` and `e` of type `exp`) in `R` corresponds to some $[e/x]$ in $[\overline{e}/\overline{x}]$ i.e. `x` is replaced by `e`. The **`Fixpoint`** function `subst` below takes a substitution mapping (`SubstRel`) `E` and an expression (`exp`) `e` as inputs, and returns `e` with `E` substituted into it. To do this, `subst` recursively substitutes `E` into all the sub-expressions of `e`, except for variables. For `e = e_var x`, `subst` searches `E` for `(x,e0)` (where `e0` is some expression), and returns `e0`. If `x` is not in `E`, then the original variable `e_var x` is returned. To search `E`, the function `get` is used below. The encoding of `get` is not given, but `get x E` either returns `None` if `x` is not in `E`, or `Some e0` if `(x,e0)` is in `E`.

```
Fixpoint subst (E : SubstRel)
               (e : exp) : exp := match e with
  | e_var x => match get x E with
               | None => e_var x
               | Some e0 => e0
               end
  | e_field e0 f => e_field (subst E e0) f
```

The encoding for `e_field` is also given. For a field expression `e_field e0 f`, `subst` is applied recursively, and `e_field (subst E e0) f` is returned. The rest of the cases for `subst` are not given since each one simply recursively applies `subst` in the same way as `e_field`.

3.2.3 FIJ Functions

In this section, the method and field lookup functions are encoded. In the paper presentation of the functions, they are functions that take inputs and return an output. In the encoding however the functions are encoded as relations. For example the *fields* function (Figure 3.3) takes a class as input

$$fields(\texttt{Object}) = \emptyset \quad \text{(F-OBJECT)}$$

$$\frac{\texttt{class C<I> extends D<I>}\{\overline{\texttt{T}}\ \overline{\texttt{f}};\ \overline{\texttt{M}}\}}{fields(\texttt{C}) = \overline{\texttt{T}}\ \overline{\texttt{f}} \cup fields(\texttt{D})} \quad \text{(F-CLASS)} \qquad \frac{\texttt{T f} \in fields(\texttt{C})}{fType(\texttt{f}, \texttt{C}) = \texttt{T}}$$

Figure 3.3: FIJ Field Lookup Function

and returns a list of the fields of that class. The Coq encoding of *fields*, `fields` is encoded as a relation that takes a class and a list of fields as inputs, and holds if the list of fields is the correct list of fields of the class. In other words, for a class `C` with fields `fs`, $fields(\texttt{C}) = \texttt{fs}$ translates to `fields C fs`. This is the same for the method lookup function.

The FIJ field lookup functions are shown in Figure 3.3. *fields*(`T`) retrieves the field names and types for type `T`. `Object<I>` has no fields for all `I` (F-OBJECT). For all other types, the fields for that type are the fields declared in the class declaration of that type and the fields of the super type (F-CLASS). Field overriding is not permitted in FIJ. This is enforced during class typing later on in Section 3.2.5. *fType*(`f`,$\texttt{T}_0$) returns the type of field `f` in type $\texttt{T}_0$. Field `f` has type `T` in type $\texttt{T}_0$ if `T` f is in the set of fields of $\texttt{T}_0$. The `fields` predicate below, encodes *fields*. `fields C fs` holds if the list of field-type pairs `fs` is the list of fields associated with class `C`. F-OBJECT is encoded as `fields_obj`, the `Object` class has no fields (i.e. `nil`, see Chapter 2). F-CLASS is encoded as `fields_extends`. A class `C` that extends a class `D` (`C = C0 extends D`), where `D` has fields `Df` and `Cf` are the fields declared in the class declaration for `C`, has fields `concat Cf Df` (`Cf` concatenated with `Df`).

```
Inductive fields : class -> flds -> Prop :=
  | fields_obj : fields Object nil
  | fields_extends : forall C C0 Cf D Df  mutX ms,
                In (cDecl C mutX Cf ms) CT ->
                C = C0 extends D ->
                fields D Df ->
```

$$\frac{\texttt{class C<I> extends T}\{\overline{\texttt{S}}\ \overline{\texttt{f}};\ \overline{\texttt{M}}\} \quad \texttt{T}_0\ \texttt{m}\ \texttt{P}\ (\overline{\texttt{U}}\ \overline{\texttt{x}})\{\texttt{return e;}\} \in \overline{\texttt{M}}}{mType(\texttt{m}, \texttt{C}) = \texttt{P}\ \overline{\texttt{U}} \rightarrow \texttt{T}_0} \quad \text{(MT-CLASS)}$$

$$\frac{\texttt{class C<I> extends D<I>}\{\overline{\texttt{S}}\ \overline{\texttt{f}};\ \overline{\texttt{M}}\} \quad \forall\, \texttt{T}_0, \texttt{P}\ \overline{\texttt{U}}, \overline{\texttt{x}}, \texttt{e} : \texttt{T}_0\ \texttt{m}\ \texttt{P}\ (\overline{\texttt{U}}\ \overline{\texttt{x}})\{\texttt{return e;}\} \notin \overline{\texttt{M}}}{mType(\texttt{m}, \texttt{C}) = mType(\texttt{m}, \texttt{D})} \quad \text{(MT-SUPER)}$$

$$\frac{\texttt{class C<I> extends T}\{\overline{\texttt{S}}\ \overline{\texttt{f}};\ \overline{\texttt{M}}\} \quad \texttt{T}_0\ \texttt{m}\ \texttt{P}\ (\overline{\texttt{U}}\ \overline{\texttt{x}})\{\texttt{return e;}\} \in \overline{\texttt{M}}}{mBody(\texttt{m}, \texttt{C}) = (\overline{\texttt{x}}; \texttt{e})} \quad \text{(MB-CLASS)}$$

$$\frac{\texttt{class C<I> extends D<I>}\{\overline{\texttt{S}}\ \overline{\texttt{f}};\ \overline{\texttt{M}}\} \quad \forall\, \texttt{T}_0, \texttt{P}\ \overline{\texttt{U}}, \overline{\texttt{x}}, \texttt{e} : \texttt{T}_0\ \texttt{m}\ \texttt{P}\ (\overline{\texttt{U}}\ \overline{\texttt{x}})\{\texttt{return e;}\} \notin \overline{\texttt{M}}}{mBody(\texttt{m}, \texttt{C}) = mBody(\texttt{m}, \texttt{D})} \quad \text{(MB-SUPER)}$$

Figure 3.4: FIJ Method Lookup Function

```
fields C (concat Cf Df).
```

The predicate `validField` is defined below. `validField C fi Ti` holds if the field `fi` with type `Ti` is declared for class `C`. This encodes the *fType* function from Figure 3.3. The encoding is straightforward, and retrieves the fields of `C` (`fields C fs`), and checks that the field-type pair `(fi,Ti)` is in `fs` (`In (fi,Ti) fs`).

```
Definition validField (C : class)
                      (fi : field)(Ti : ty) : Prop :=
  exists fs, fields C fs /\ In (fi,Ti) fs.
```

The method type and body lookup functions are found in Figure 3.4. $mType(\texttt{m}, \texttt{C})$ retrieves the type of method `m` in class `C`. A method `m` has type $\texttt{P}\ \overline{\texttt{U}} \rightarrow \texttt{T}_0$ in class `C` if `m` is has that signature in the list of methods of

the class declaration of C (MT-CLASS). A method m has the same type in C as in the super class D providing that there is no method m declared in class C (MT-SUPER). $mBody(\mathtt{m}, \mathtt{C})$ retrieves the method body of m in class C. As with *mType*, a method m has body e for a class C if it is declared in the class declaration of C (MB-CLASS), or if it is inherited from the super class D (MB-SUPER). *mBody* returns not only the body of the method, but also the parameter variables ($\overline{\mathtt{X}}$) of the method that are needed for parameter substitution.

Below is method, the predicate encoding both *mType* and *mBody*. They are encoded together for two reasons. They both operate in the same way, only returning a different part of the method signature, so combining the encoding allows for reuse of the same predicate. Encoding them together allows for ease during soundness when attempting to prove that the body and the type of the same method in a class are derived from the same declaration.

method C decl holds if the method declaration decl is defined for class C. m_this encodes both MT-CLASS and MB-CLASS. For a class with method list ms, decl is a valid method declaration for C if decl is in ms. m_inherit encodes MT-SUPER and MB-SUPER. For a class C that extends class D (C = Cn extends D), a method declaration decl is defined for C if decl is defined for D (method D decl), and C has no method by the same name as decl in its method list.

```
Inductive method : class ->  MethDecl -> Prop :=
  | m_this : forall decl T0 m As e0 C fs ms mutX mutM,
             decl = mDecl m T0 mutM As e0 ->
             In (cDecl C mutX fs ms) CT ->
             In decl ms ->
             method C decl
  | m_inherit : forall C D mutX fs ms Cn decl m T0 mutM As e0,
                decl = mDecl m T0 mutM As e0 ->
                In (cDecl C mutX fs ms) CT ->
                (forall T0' mutM' As' e0',
```

$$\texttt{T <: T}\ \text{(S-REFL)} \qquad \frac{\texttt{S <: T} \quad \texttt{T <: E}}{\texttt{S <: E}}\ \text{(S-TRANS)}$$

$$\frac{\texttt{C<I> extends D<I>}}{\texttt{C<I> <: D<I>}}\ \text{(S-EXTEND)}$$

Figure 3.5: FIJ Subtyping Rules

```
  ~ In (mDecl m T0' mutM' As' e0') ms) ->
C = Cn extends D ->
method D decl ->
method C decl.
```

3.2.4 FIJ Subtyping and Well-Formedness

Figure 3.5 gives the FIJ subtype rules. The rules provide reflexivity (S-REFL), transitivity (S-TRANS) and class extension (S-EXTEND). They are the same rules as those in FJ [6], except for the introduction of the mutability parameter in S-EXTEND. S-EXTEND requires that both the super type and the subtype must have the same mutability parameter. This creates the split in the type hierarchy that is illustrated by Figure 3.1 and discussed in Section 3.1.

Below is the encoding for the `subclass` predicate. `subtype C D` holds if class `C` is a subclass of class `D`. `S_Refl`, `S_Trans` and `S_Extends` capture the reflexivity, transitivity and class extension relationships.

```
Inductive subclass : class -> class -> Prop :=
  | S_Refl    : forall C, subclass C C
  | S_Trans   : forall C D E, subclass C D ->
                subclass D E -> subclass C E
  | S_Extends : forall C D C0 mutC fs ms,
                In (cDecl C mutC fs ms) CT ->
```

$$\texttt{Object<I>} : \texttt{ok} \quad \text{(WF-OBJECT)}$$

$$\frac{\texttt{class C<}\mathtt{I_C}\texttt{> extends D<}\mathtt{I_C}\texttt{>}\{\overline{\texttt{T}}\ \overline{\texttt{f}};\ \overline{\texttt{M}}\} \quad (\mathtt{I_C} = \texttt{mutable} \vee \texttt{immutable}) \Rightarrow \texttt{I} = \mathtt{I_C} \qquad \texttt{D<I>} : \texttt{ok}}{\texttt{C<I>} : \texttt{ok}} \quad \text{(WF-CLASS)}$$

Figure 3.6: FIJ Type Well-Formedness

```
C = C0 extends D ->
subclass C D.
```

The `subtype` predicate below encodes the subtype judgment of Figure 3.5. `subtype S T` holds if type `S` is a subtype of type `T`. For `S` to be a subtype of `T`, the class of `S` must subclass the class of `T`, and they must have the same mutability parameter. The last requirement enforces the type hierarchy of Figure 3.1.

```
Definition subtype (T1 T2 : ty) : Prop :=
  exists mut0, exists2 C, exists2 D,
    subclass C D & (T2 = D <<mut0>>) & (T1 = C <<mut0>>).
```

FIJ type well-formedness is given in Figure 3.6. `Object<I>` is well-formed for all `I` (WF-OBJECT). A type `C<I>` is said to be well-formed if `I` conforms to the mutability as declared in class `C` (WF-CLASS). That is, if `C` is defined as having mutability parameter $\mathtt{I_C}$, then if $\mathtt{I_C}$ is a non-variable mutability parameter, $\mathtt{I} = \mathtt{I_C}$. The super class `D` is also required to be

The encoding for type well-fomedness (`ok_type`) is given below. `ok_type` is an inductive predicate that takes a type and a class table as inputs, and holds if the type is well-formed with respect to the class table. While this predicate is defined with an explicit class table parameter, it could just as easily be defined using the generic class table `CT`. `ok_Obj` encodes WF-OBJECT. `ok_class` encodes WF-CLASS.

```
Inductive ok_type : ty -> ClassTable-> Prop :=
```

```
  | ok_Obj   : forall mutObj CTbl,
               ok_type Object <<mutObj>> CTbl
  | ok_class : forall C Cn D mutC mutX fs ms CTbl,
               C = Cn extends D ->
               ok_type D <<mutC>> CTbl ->
               In (cDecl C mutX fs ms) CTbl ->
               (mutability_defined mutX -> mutC = mutX) ->
               ok_type C <<mutC>> CTbl.
```

3.2.5 FIJ Expression Typing

Before expression typing can be expressed, two environments need to be mentioned. First, Chapter 2 introduced an environment mapping variables to types. This environment is still used, and is referred to as simply an environment in the type system. Below is the Coq encoding of environments, `env`.

```
Notation env := (list  (var * ty)).
```

An `env` is a list of variable-type pairs. Each pair represents a mapping in the environment. The second environment used in the typing of FIJ expressions is a store typing environment, mapping locations in a store to types. Store typing environments are encoded below as `store_typing`.

```
Definition store_typing := list ty.
```

A `store_typing` is a list of types (`ty`). Since store typing environments all correspond to some store, there must be a mapping from a location in `store H` to a type in `store_typing Delta`. For a location `e_loc i`, that points to the `i`th position in `H`, it also maps to the `i`th position in `Delta`.

Expression type rules for FIJ are given in Figure 3.7. An expression `e` is said to have type `T` with respect to environment Γ and store typing Δ if $\Gamma; \Delta \vdash$ `e : T` holds. This is encoded by the Coq predicate `typing` that takes an environment (`env`), a store typing environment (`store_typing`),

$$\frac{}{\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x})}\ \text{(T-VAR)} \qquad \frac{\mathtt{T} : \mathtt{ok}}{\Gamma \vdash \mathtt{null} : \mathtt{T}}\ \text{(T-NULL)}$$

$$\frac{}{\Gamma \vdash \iota : \Delta(\iota)}\ \text{(T-LOC)} \qquad \frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{C_0<I>} \qquad fType(\mathtt{f}, \mathtt{C}_0) = \mathtt{T}}{\Gamma \vdash \mathtt{e}_0.\mathtt{f} : \mathtt{T}}\ \text{(T-FIELD)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{e}_0 : \mathtt{C<mutable>} \qquad \Gamma \vdash \mathtt{e} : \mathtt{T} \\ fType(\mathtt{f}_i, \mathtt{C}_0) = \mathtt{T}_i \qquad \mathtt{T} \mathtt{<:} \mathtt{T}_i\end{array}}{\Gamma \vdash \mathtt{e}_0.\mathtt{f}_i = \mathtt{e} : \mathtt{T}}\ \text{(T-ASSIGN)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{e}_0 : \mathtt{C<I>} \qquad mType(\mathtt{m}, \mathtt{C}) = \mathtt{P}\ \overline{\mathtt{U}} \to \mathtt{T} \\ \mathtt{I} \neq \mathtt{mutable} \Rightarrow \mathtt{P} = \mathtt{pure} \qquad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}} \qquad \overline{\mathtt{S}} \mathtt{<:} \overline{\mathtt{U}}\end{array}}{\Gamma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) : \mathtt{T}}\ \text{(T-INVK)}$$

$$\frac{\mathtt{C<I>} : \mathtt{ok} \qquad fields(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}} \qquad \overline{\mathtt{S}} \mathtt{<:} \overline{\mathtt{T}}}{\Gamma \vdash \mathtt{new\ C<I>}(\overline{\mathtt{e}}) : \mathtt{C<I>}}\ \text{(T-NEW)}$$

$$\frac{\Gamma \vdash \mathtt{e}_1 : \mathtt{T}_1 \qquad \Gamma \vdash \mathtt{e}_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{e}_1; \mathtt{e}_2 : \mathtt{T}_2}\ \text{(T-SEQ)}$$

Figure 3.7: FIJ Typing Rules

an expression (`exp`) and a type (`ty`) as inputs. The header for `typing` is given below.

```
Inductive typing :
  env -> store_typing -> exp -> ty -> Prop :=
```

An expression variable `x` in environment Γ has type $\Gamma(\mathtt{x})$ (T-VAR). The Coq encoding is shown below as `T_Var`. Since the environment `Gamma` is actually a list of variable-type pairs representing variable to type mappings, for a unique `x` in `Gamma`, $\Gamma(\mathtt{x}) = \mathtt{T}$ translates to `In (x,T) Gamma`.

```
  | T_Var : forall Gamma Delta x T, env_ok Gamma ->
            In (x,T) Gamma -> ok_type T CT ->
```

```
            typing Gamma Delta (e_var x) T
```

A null expression is judged to have type `T` for all types `T` (T-NULL). This is captured by `T_Null` below. The only requirement being that `T` is well-formed (`ok_type T CT`).

```
| T_Null    : forall Gamma Delta T,
              ok_type T CT ->
              typing Gamma Delta e_null T
```

A location ι in a store typing Δ has type $\Delta(\iota)$ (T-LOC). `T_Loc` below, encodes typing for locations. Since locations are identified by an index `i` in a list, the type `T`, at index `i` in the store typing `Delta` is retrieved by `store_typing_lookup i Delta = T`. The premise `i < stLength Delta` is not strictly needed, and can be derived from `store_typing_lookup i Delta = T`. This is something that is not included in later encodings.

```
| T_Loc : forall Gamma Delta i T,
          i < stLength Delta -> ok_type T CT ->
          store_typing_lookup i Delta = T ->
          typing Gamma Delta (e_loc i) T
```

A field access `e.f` has type `T` if `e` has type $\texttt{T}_0$, and field `f` has type `T` in type $\texttt{T}_0$ (T-FIELD). This is encoded below as `T_Field`. `validField C0 fi Ti` corresponds to $\mathit{fType}(\texttt{f}, \texttt{T}_0) = \texttt{T}$, but `C0` is the class of type $\texttt{T}_0$. Since `validField` retrieves a field type for a class, and not a type, the mutability `mutC0` of type `C0 <<mutC0>>` has to be substituted into the field type `Ti`. `subst_ty mutC0 Ti` substitutes `mutC0` into `Ti`.

```
| T_Field : forall Gamma Delta e0 C0 fi Ti mutC0 T,
            typing Gamma Delta e0 C0 <<mutC0>> ->
            validField C0 fi Ti -> ok_type Ti CT ->
            T = subst_ty mutC0 Ti ->
            typing Gamma Delta (e_field e0 fi) T
```

A field assignment `e.f = e'` has type `T` if `e` has a mutable type, `f` has type `T` for that type, and the type of `e`, `T'`, subtypes `T` (T-ASSIGN). The encoding for T-ASSIGN, `T_Assign`, is given below. `typing Gamma Delta e0 C0 <<mutable>>` requires that the receiver `e0` have mutability `mutable`. As with `T_Field`, `validField C0 fi Ti` retrieves the type of field `fi` for class `C0`, `Ti`. `mutable` is then substituted into `Ti`.

```
| T_Assign : forall Gamma Delta e0 C0 fi Ti e T,
             typing Gamma Delta e0 C0 <<mutable>> ->
             validField C0 fi Ti ->
             subtype T (subst_ty mutable Ti) ->
             typing Gamma Delta e T ->
             typing Gamma Delta (e_assign e0 fi e) T
```

If `e` has type `C <<I>>`, `m` has type `p` $\overline{\texttt{D}}$ $\rightarrow$ `T` for a reciever of type `C0 <<I>>`, where `p` = `pure` if `I` $\neq$ `mutable`, and $\overline{\texttt{e}}$ subtypes $\overline{\texttt{D}}$, then `e.m`($\overline{\texttt{e}}$) has type `T` (T-INVK). The encoding for T-INVK is found below as `T_Invk`.

```
| T_Invk : forall Gamma Delta e0 C0 es
                  e T0 T m As mut0 mutM,
           typing Gamma Delta e0 C0 <<mut0>> ->
           method C0 (mDecl m T0 mutM As e) ->
           (mut0 <> mutable -> mutM = pure) ->
           subtypings Gamma Delta es
             (List.map (subst_ty mut0) (range As)) ->
           ok_types (range As) CT -> ok_type T0 CT ->
           T = (subst_ty mut0 T0) ->
           typing Gamma Delta (e_meth e0 m es) T
```

A new expression `new C<I>`($\overline{\texttt{e}}$) has type `C <I>` if the types of $\overline{\texttt{e}}$ subtype the field types of `C <I>` (T-NEW). The encoding of T-NEW is given below.

```
| T_New : forall Gamma Delta C es fs Ts mutC,
          mutability_defined mutC ->
          fields C fs -> range fs = Ts ->
          subtypings Gamma Delta es Ts ->
```

```
            ok_type C <<mutC>> CT ->
            typing Gamma Delta
                      (e_new C mutC es) C <<mutC>>
```

A sequence `e1; e2` has type `T2` if `e1` is well-typed with respect to some `T1`, and `e2` is well-typed with respect to `T2` (T-SEQ). The Coq encoding of T-SEQ is given below.

```
  | T_Seq : forall Gamma Delta e1 e2 T1 T2,
            typing Gamma Delta e1 T1 ->
            typing Gamma Delta e2 T2 ->
            typing Gamma Delta (e_seq e1 e2) T2
```

The only expression that is not well-typed with respect to some type is `e_err`. We cannot catch computation errors, all we can say is that they are not well-typed. In Section 3.3 we present the properties of the type system with the assumption that computation does not result in an error.

The predicate `subtyping` is defined along with `typing`, and is used instead of a combined typing and subtyping predicate. `subtyping Gamma Delta e T` holds if `typing Gamma Delta e T'` holds for some `T'`, and `T'` is a subtype of `T`. `subtypings` is defined as `subtyping` for lists of expressions and types.

```
with subtyping : env -> store_typing ->
                 exp -> ty -> Prop :=
  | T_Sub : forall Gamma Delta e T T',
            typing Gamma Delta e T ->
            subtype T T' -> ok_type T' CT ->
            subtyping Gamma Delta e T'
with subtypings : env -> store_typing ->
                  list exp -> list ty -> Prop :=
  | T_Nil  : forall Gamma Delta,
             subtypings Gamma Delta nil nil
  | T_Subs : forall Gamma Delta e T es Ts,
             subtypings Gamma Delta es Ts ->
```

$$\frac{mType(\mathtt{m},\mathtt{D}) = \mathtt{Q}\,\overline{\mathtt{U}} \to \mathtt{U}_0 \Rightarrow \overline{\mathtt{T}} = \overline{\mathtt{U}} \wedge \mathtt{T}_0 \mathtt{<:} \mathtt{U}_0 \wedge \mathtt{P} = \mathtt{Q}}{override(\mathtt{m},\mathtt{D},\mathtt{P}\,\overline{\mathtt{T}} \to \mathtt{T}_0)}$$

Figure 3.8: FIJ Override Function

```
subtyping Gamma Delta e T ->
subtypings Gamma Delta (e::es) (T::Ts).
```

The FIJ override function is given in Figure 3.8. $override(\mathtt{m},\mathtt{D},\mathtt{P}\,\overline{\mathtt{T}} \to \mathtt{T}_0)$ holds if the method `m` being defined in class `D` implies the method type $\mathtt{P}\,\overline{\mathtt{T}} \to \mathtt{T}_0$ overrides the method type of `m` in `D`. A method type overrides another method type if the parameter types are the same, the return type subtypes that of the overridden method, and they have the same `pure/mutating` annotation. There is no distinct encoding of this function as it is fairly simple, and is incorporated into the method typing discussed next.

The FIJ method and class typing rules are given in Figure 3.9. A method declaration $\mathtt{T}\ \mathtt{m}\ (\overline{\mathtt{T}}\,\overline{\mathtt{x}})\{\mathtt{return}\ \mathtt{e};\}$ is said to be well-formed in type `C<I>` if $\mathtt{T}\ \mathtt{m}\ (\overline{\mathtt{T}}\,\overline{\mathtt{x}})\{\mathtt{return}\ \mathtt{e};\}$ `OK IN C<I>` holds. A class declaration `class C<I>` `extends D<I>`$\{\overline{\mathtt{T}}\,\overline{\mathtt{f}};\ \overline{\mathtt{M}}\}$ is said to be well-formed if `class C<I> extends` `D<I>`$\{\overline{\mathtt{T}}\,\overline{\mathtt{f}};\ \overline{\mathtt{M}}\}$ `OK` holds.

A method declaration is well-formed in one of two cases, T-METH-PURE for pure methods, and T-METH-MUT for non-pure (or mutating) methods. A pure method is well-typed if it overrides any method by the same name in the super type and the body is well-typed for an immutable receiver and an empty store (T-METH-PURE). A mutating method is well-typed for a type `C<I>` if `I` is either `mut` or a variable, and the body is well-typed for a mutable receiver and an empty store (T-METH-MUT). As with T-METH-PURE, the method must correctly override any method by the same name in the super type.

$$\frac{\texttt{this} : \texttt{C<imm>}, [\texttt{imm/X}]\overline{\texttt{T}}\,\overline{\texttt{x}}; \emptyset \vdash [\texttt{imm/X}]\texttt{e} : [\texttt{imm/X}]\texttt{T} \quad \texttt{class C<I}_C\texttt{> extends D<I}_C\texttt{>}\{...\} \quad \mathit{override}(\texttt{m}, \texttt{D<I>}, \texttt{pure}\,\overline{\texttt{T}} \to \texttt{T})}{\texttt{T m pure}\,(\overline{\texttt{T}}\,\overline{\texttt{x}})\{\texttt{return e;}\}\ \texttt{OK IN C<I>}} \quad \text{(T-METH-PURE)}$$

$$\frac{\texttt{this} : \texttt{C<mut>}, [\texttt{mut/X}]\overline{\texttt{T}}\,\overline{\texttt{x}}; \emptyset \vdash [\texttt{mut/X}]\texttt{e} : [\texttt{mut/X}]\texttt{T} \quad \texttt{I} = \texttt{mut} \vee \texttt{I} = \texttt{X} \quad \texttt{class C<I}_C\texttt{> extends D<I}_C\texttt{>}\{...\} \quad \mathit{override}(\texttt{m}, \texttt{D<I>}, \texttt{mutating}\,\overline{\texttt{T}} \to \texttt{T})}{\texttt{T m mutating}\,(\overline{\texttt{T}}\,\overline{\texttt{x}})\{\texttt{return e;}\}\ \texttt{OK IN C<I>}} \quad \text{(T-METH-MUT)}$$

$$\frac{\overline{\texttt{T}}\,\overline{\texttt{f}} \cap \mathit{fields}(\texttt{D}) = \emptyset \qquad \forall \texttt{M} \in \overline{\texttt{M}} : \texttt{M OK IN C<I>}}{\texttt{class C<I> extends D<I>}\{\overline{\texttt{T}}\,\overline{\texttt{f}};\ \overline{\texttt{M}}\}\ \texttt{OK}} \quad \text{(T-CLASS)}$$

Figure 3.9: FIJ Method and Class Typing Rules

The FIJ method typing is encoded as `meth_ok`. `meth_ok` is a single definition that covers both T-METH-PURE and T-METH-MUT. `meth_ok decl C <<I>>` holds if method declaration `decl` is well-formed for type `C <<mutC>>`. `meth_ok` consists of a series of propositions joined by conjunction. Each proposition is given separately but is part of the same definition. First the header and premises for `meth_ok` is given below. The premises `decl = mDecl m T0 mutM As e0` and `T = C <<mutC>>` are added in order to extract the components of `decl` and `T`, the inputs of `meth_ok`.

```
Definition meth_ok (decl : MethDecl) (T : ty): Prop :=
  forall C mutC T0 m e0 mutM D As Cn,
     decl = mDecl m T0 mutM As e0 -> T = C <<mutC>> ->
```

To ensure that non-pure methods are not included in classes declared as `immutable`, the following proposition is added to `mneth_ok`. If `mutC` (the receiver's mutability) is `immutable`, then the method must be `pure`.

```
     (mutC = immutable -> mutM = pure) /\
```

Next, typing for the method body is given below. All method bodies are required to be well-typed for a `mutable` receiver. Below is the typing for `mutable` receivers. `mutable` is substituted for any mutability variable into the body (`e0`), the return type (`T0`) and the method parameters (`As`). The substituted body is then required to subtype the substituted return type with respect to an empty store (`nil`) and the substituted parameters along with the this variable mapped to a mutable `C`.

```
subtyping ((this,C<<mutable>>)::(map (subst_pair mutable) As))
            nil (subst_mut_exp mutable e0)
                                (subst_ty mutable T0) /\
```

If the method is annotated as `pure`, then the body must also be well-typed for an `immutable` receiver. This is done in the same manner as with a `mutable` receiver. An alternative to double the typing of method bodies (for both mutable and immutable receivers) would be to only require the bodies of pure methods to be well-typed with respect to an immutable receiver. It should not be hard to then show that pure methods are well-typed with respect to a mutable receiver. This is a consideration for future encodings.

```
(mutM = pure ->
  subtyping ((this,C<<immutable>>)::
                              (map (subst_pair immutable) As))
                nil (subst_mut_exp immutable e0)
                (subst_ty immutable T0)) /\
```

The prerequisites of the *override* function are included below, the return type must subtype the return type of any overridden methods, the parameter types must be the same and they must have the same pure/mutating annotation. The `range` function used below is not given here, but takes a list of pairs as an input, and returns a list composed of the second component of each pair.

```
    C = Cn extends D /\
    (forall T0' mutM' Bs e0',
```

```
        method D (mDecl m T0' mutM' Bs e0') ->
          (range As = range Bs) /\
          (subtype T0 T0') /\ (mutM = mutM')).
```

Finally, the notation "`decl 'OK_IN' C`" is used for `meth_ok decl T`.

```
Notation "decl 'OK_IN' T" := (meth_ok decl T) (at level 0).
```

Class typing is given in Figure 3.9 as T-CLASS. A class declaration is well-formed if there are no fields overriding those of the super class and if the list of method declarations are well-formed for the class. The Coq encoding of T-CLASS is given below as `class_ok`. `class_ok` is encoded as a `Definition` that takes a class declaration, and returns a proposition (`Prop`). The encoding is a straight encoding of T-CLASS, ensuring that the methods are well-formed, the fields do not override any super type fields and that all types are well-formed for that class declaration.

```
Definition class_ok (decl : ClassDecl): Prop :=
forall C ms fs mutC, decl = cDecl C mutC fs ms ->
 (ok_meths ms /\ (forall fC, fields C fC -> ok_fields fC)/\
 (forall Ci muti fi,
   (In (fi, Ci <<muti>>) fs -> ok_type Ci <<muti>> CT /\
   (muti = mutC \/ mutability_defined muti))) /\
 (forall m T0 As e0 mutM, (In (mDecl m T0 mutM As e0) ms ->
 (mDecl m T0 mutM As e0) OK_IN (C <<mutC>>) /\
 (forall C0 mut0, T0 = C0 <<mut0>> ->
mut0 = mutC \/ mutability_defined mut0) /\
 (ok_type T0 CT) /\
 (forall xi Ci muti, In (xi,Ci <<muti>>) As ->
 (muti = mutC \/ mutability_defined muti) /\
                                  ok_type Ci <<muti>> CT)))).

Notation "'CLASS' decl 'OK'" := (class_ok decl) (at level 0).
```

$$\frac{\mathcal{H}(\iota) = \texttt{new C<M>}(\overline{\texttt{v}}) \quad \mathit{fields}(\texttt{C}) = \overline{\texttt{C}}\,\overline{\texttt{f}}}{\iota.\texttt{f}_i|\mathcal{H} \longrightarrow \texttt{v}_i|\mathcal{H}} \text{ (R-FIELD)}$$

$$\frac{\mathcal{H}(\iota) = \texttt{new C<M>}(\overline{\texttt{v}}) \quad \mathit{fields}(\texttt{C}) = \overline{\texttt{C}}\,\overline{\texttt{f}} \quad \mathcal{H}' = \mathcal{H}[\iota \mapsto \texttt{new C<M>}(..., \texttt{v}_{i-1}, \texttt{v}, \texttt{v}_{i+1}, ...)]}{\iota.\texttt{f}_i = \texttt{v}|\mathcal{H} \longrightarrow \texttt{v}|\mathcal{H}'} \text{ (R-ASSIGN)}$$

$$\frac{\mathcal{H}(\iota) = \texttt{new C<M>}(...) \quad \mathit{mBody}(\texttt{m}, \texttt{C<M>}) = (\overline{\texttt{x}}; \texttt{e})}{\iota.\texttt{m}(\overline{\texttt{v}})|\mathcal{H} \longrightarrow [\iota/\texttt{this}, \overline{\texttt{v}}/\overline{\texttt{x}}]\texttt{e}|\mathcal{H}} \text{ (R-INVK)}$$

$$\frac{\iota \notin \mathit{dom}(\mathcal{H}) \quad \mathcal{H}' = \mathcal{H}, \iota \mapsto \texttt{new C<M>}(\overline{\texttt{v}})}{\texttt{new C<M>}(\overline{\texttt{v}})|\mathcal{H} \longrightarrow \iota|\mathcal{H}'} \text{ (R-NEW)}$$

$$\frac{}{\texttt{v; e}|\mathcal{H} \longrightarrow \texttt{e}|\mathcal{H}} \text{ (R-SEQ)}$$

Figure 3.10: FIJ Reduction Rules

3.2.6 FIJ Reduction

This section presents the reduction rules and their Coq encodings. FIJ reduction represents the evaluation of one FIJ expression-store pair to another. The reduction rules are given in Figure 3.10. An expression $\texttt{e}$ with store $\mathcal{H}$ reduces to $\texttt{e}'$ with store $\mathcal{H}'$ if $\texttt{e}|\mathcal{H} \longrightarrow \texttt{e}'|\mathcal{H}'$ holds. The core FIJ reductions are given, but the context and null reduction rules are not given. The FIJ context reduction rules, i.e. the reduction of subexpressions, are straightforward, and so are not given. The null reduction rules involve the reduction of field or method accesses on a null receiver, and result in error. As with context reduction, null reduction is straightforward, and so is not included. A field access $\iota.\texttt{f}_i$ reduces to the corresponding value $\texttt{v}_i$ in $\mathcal{H}(\iota)$ (R-FIELD). A field assignment $\iota.\texttt{f}_i = \texttt{v}$ with store $\mathcal{H}$ reduces to $\texttt{v}$, and replaces the value stored for $\texttt{f}_i$ in $\mathcal{H}$ with $\texttt{v}$ (R-ASSIGN). A method invoca-

tion ι.m($\overline{\text{e}}$) reduces to the body e of the method m with the parameters and the receiver substituted in for the appropriate variables (R-INVK). A new expression new C<M>($\overline{\text{v}}$) reduces to ι if $\iota \notin \mathcal{H}$ where ι contains an object of type C<I> with field values $\overline{\text{v}}$ and $\mathcal{H}$ is the store (R-NEW). A sequence v; e with store $\mathcal{H}$ reduces to e, and $\mathcal{H}$ remains unchanged (R-SEQ).

The following notation is used in Coq to represent expression reduction.

```
Reserved Notation "e1 '/' H1 '-->' e2 '/' H2"
  (at level 40, H1 at level 39, e2 at level 39).
```

An expression e1 with store H1 reduces to an expression e2 with store H2 if e1 / H1 --> e2 / H2 holds.

The Coq encodings of the reduction rules are given separately here for convenience, but form a single reduction rule in the encoding. The header for the reduction predicate is given below. The reduction predicate is encoded as an Inductive definition, taking two expression-store pairs and holds if the first reduces to the second.

```
Inductive reduction : exp * store -> exp * store -> Prop :=
```

R_Field encodes R-FIELD below. For a field access e_field (e_loc i) f with store H, the object (C <<mutC>>,vs) at location i is retrieved by store_lookup i H. The value v is retrieved from vs, the list of field values. The original field access reduces to v with an unchanged store.

```
  | R_Field  : forall C i H fs vs fv f v mutC,
               store_lookup i H = (C <<mutC>>,vs) ->
               fields C fs -> ok_fields fs ->
               zipFlds fs vs fv -> In (f,v) fv ->
               (e_field (e_loc i) f) / H --> v / H
```

R_Assign encodes R-ASSIGN below. A field assignment e_assign (e_loc i) f v reduces to the value v. The object (C <<mutC>>,vs) at location i is retrieved by store_lookup i H. The index of the field f in fs is identified by lookup_index n fs. The nth value in vs is then

replaced by `v` (`replace n v vs`). The new store is derived by replacing the object at the original location with a new object of the same type, but with the modified list of values `replace i (C <<mutC>>,vs') H`.

```
| R_Assign : forall H H' C i n fs vs vs' fv v f T mutC,
             store_lookup i H = (C <<mutC>>,vs) ->
             value v ->  fields C fs ->
             ok_fields fs -> zipFlds fs vs fv ->
             lookup_index n fs = Some (f, T) ->
             vs' = replace n v vs ->
             H' = replace i (C <<mutC>>,vs') H ->
             e_assign (e_loc i) f v / H --> v / H'
```

`R_Invk` encodes R-INVK below. For a method invocation `e_meth (e_loc i) m vs` with store `H` to be reduced, the following prerequisites must hold. `i` must point to an existing location in `H` containing an object `(C <<mutC>>,es)`. The parameters `vs` must be values (i.e. null expressions or locations), and this is captured by `values vs`. `values` is not given here, but is a predicate that holds for a list of expressions `vs` if all expressions in `vs` are values. `method C (mDecl m T0 mutM xs e)` retrieves the method parameter variables and body of `m` in `C`. We then have to construct a substitution mapping that maps method parameter variables to parameters. To do this, we use the `SubstRelZip` predicate. `SubstRelZip` takes three parameters `xs` a list of variable-type pairs, `vs` a list of expressions and `R` a list of variable-expression pairs (`SubstRel`), and holds if `xs` zips together with `vs` to form `R`. Finally we can say that `e_meth (e_loc i) m vs` with store `H` reduces to `(subst ((this,e_loc i)::R)(subst_mut_exp mutC e))` with store `H`. We add the mapping of `this` to `e_loc i` on to the `R` we constructed. We also substitute any mutability variable in `e` for `mutC`.

```
| R_Invk : forall H i C m xs vs e R es T0 mutC mutM,
           store_lookup i H = (C <<mutC>>,es) ->
           values vs -> SubstRelZip xs vs R ->
           method C (mDecl m T0 mutM xs e) ->
```

```
              (e_meth (e_loc i) m vs) / H -->
                   (subst ((this,e_loc i)::R)
                              (subst_mut_exp mutC e)) / H
```

R_New encodes R-NEW below. Since a store is encoded as a list of objects, new locations are appended to the end of the list. Therefore, the new store is derived by appending a new object to the end of H (stSnoc H (C <<mutC>>,vs)). The constructor parameters vs must be values (values vs). The reduced expression is e_loc i, where i is the final position of the reduced store.

```
  | R_New : forall H H' i C vs mutC,
           stLength H = i -> values vs ->
           H' = stSnoc H (C <<mutC>>,vs) ->
           (e_new C mutC vs) / H --> (e_loc i) / H'
```

R_Seq encodes R-SEQ below. A sequence v ;; e reduces to e, with an unchanged store H, if v is a value (value v).

```
  | R_Seq : forall v e H, value v -> v ;; e / H --> e / H
```

3.3 FIJ Soundness

The statements of the FIJ soundness theorems, *Preservation* and *Progress*, are given in this section. Before soundness can be proven, we need to define a mutual induction scheme for reduction and ListReduction, as well as one for typing, subtyping and subtypings. Mutual induction was introduced in Chapter 2, and it is tackled in the same way here. Below is the mutual induction scheme for reduction.

```
Scheme reduction_reduction_ind :=
                 Minimality for reduction Sort Prop
with reduction_listreduction_ind :=
                 Minimality for ListReduction Sort Prop.
```

```
Combined Scheme reduction_mutind from
      reduction_reduction_ind, reduction_listreduction_ind.
```

First, induction for `reduction` and `ListReduction` is defined. Secondly, these two induction schemes are joined to form the combined scheme `reduction_mutind` Below is the mutual induction scheme for typing.

```
Scheme typing_typings_ind := Minimality for
                        typing Sort Prop
with typing_subtyping_ind := Minimality for
                        subtyping Sort Prop
with typing_subtypings_ind := Minimality for
                        subtypings Sort Prop.

Combined Scheme typings_mutind from
         typing_typings_ind, typing_subtyping_ind,
 typing_subtypings_ind.
```

This is defined in a similar way to `reduction`, except adding an extra induction scheme for `subtyping`.

Preservation

Theorem 3.3.1 is the statement of the *Preservation* Theorem for FIJ. Loosely it states that for an expression `e` that is well-typed with respect to some type `T`, any reduction that does not result in an error (`err`) will be well-typed too. A field access on a null receiver `null.f` may be well-typed, but it reduces to `err`, which is not well-typed.

Theorem 3.3.1 (Preservation). *If* $\Gamma, \Delta, \vdash e : T$, $e|\mathcal{H} \longrightarrow e'|\mathcal{H}'$ *where* $e' \neq err$ *and* $\mathcal{H}$ *is well-typed with respect to* Δ *then* $\exists \Delta', T'$ *s.t.* Δ' *extends* Δ, $\Gamma, \Delta' \vdash e' : T'$ *and* $T' <: T$.

Below is the Coq encoding of *Preservation*. In order to be able to apply the mutual induction scheme for `reduction` and `ListReduction`, the

reduction notation `e / H --> e' / H'` defined earlier cannot be used. This is because the mutual induction scheme we defined expects two pairs, `p` and `p'`, not two expressions `e` and `e'` and two stores `H` and `H'`. For this reason, reduction is written below as `reduction p p'`, where `p = (e,H)` and `p' = (e',H')`. Similarly, reduction of lists is written below as `ListReduction p p'`, where `p = (es,H)` and `p' = (es',H')` (`es` and `es'` being lists of expressions). Our premises are that the store `H` must be well-typed with respect to the store typing `Delta` (`store_well_typed Delta H`), the environment `Gamma` must be well-formed (`env_ok Gamma`), and `e` must be well-typed with respect to some type `T`. If this is the case, then `e'` must be well-typed and subtype `T`, with respect to some `Delta'` that extends `Delta`, and `H'` must be well-typed with respect to `Delta'`. The mutual case for expression lists extends this for lists, `ListReduction` instead of `reduction` and `subtypings` instead of `typing`.

```
Theorem Preservation :
(forall p p', reduction p p' ->
  (forall Gamma Delta T e e' H H',
   (e,H) = p -> (e',H') = p' ->
   e' <> e_err ->
   store_well_typed Delta H -> env_ok Gamma ->
   typing Gamma Delta e T ->
     (exists Delta', ST_Extends Delta' Delta ->
                     store_well_typed Delta' H' ->
                     subtyping Gamma Delta' e' T))) /\
(forall p p', ListReduction p p' ->
  (forall Gamma Delta Ts es es' H H',
   (es,H) = p -> (es',H') = p' ->
   ~ In e_err es' ->
   store_well_typed Delta H -> env_ok Gamma ->
   subtypings Gamma Delta es Ts ->
 (exists Delta', ST_Extends Delta' Delta ->
                 store_well_typed Delta' H' ->
                 subtypings Gamma Delta' es' Ts))).
```

Progress

Theorem 3.3.2 is the statement of *Progress* for FIJ. Given a well typed expression `e`, either `e` is a value, or there exists `e'` such that `e` reduces to `e'`.

Theorem 3.3.2 (Progress)**.** *If* $\Gamma, \Delta \vdash e : T$*, then either*

(i) e *is a value, or*

(ii) $\forall \mathcal{H}$ *s.t.* $\mathcal{H}$ *is well-typed with respect to* Δ*,* $\exists e', \mathcal{H}'$ *s.t.* $e|\mathcal{H} \longrightarrow e'|\mathcal{H}'$

The Coq encoding of Progress is given below. *Progress* makes use of the mutual induction scheme `typings_mutind` defined at the beginning of this Section. Mutual induction needs *Progress* to be proven for `typing`, `subtyping` and `subtypings` at the same time, so the statement is broken up into three statements, joined by conjunctions. For a tying judgment `typing Gamma Delta e T`, if the environment `Gamma` is empty (`Gamma = nil`), then either `e` is a value (`value e`), or for all stores `H` that are well-formed with respect to `Delta`, there exists `e'` and `H'` such that `e / H --> e' / H'`. This is extended to the cases for `subtyping` and `subtypings`.

```
Theorem Progress :
(forall Gamma Delta e T,
   typing Gamma Delta e T -> Gamma = nil ->
     (value e \/ (forall H, store_well_typed Delta H ->
      exists e', exists H', e / H --> e' / H'))) /\
(forall Gamma Delta e T,
   subtyping Gamma Delta e T -> Gamma = nil ->
     (value e \/ (forall H, store_well_typed Delta H ->
      exists e', exists H', e / H --> e' / H'))) /\
(forall Gamma Delta es Ts,
  subtypings Gamma Delta es Ts -> Gamma = nil ->
  values es \/ (forall H, store_well_typed Delta H ->
  exists es', exists H', ListReduction (es, H) (es', H')))).
```

Immutability Guarantee

Theorem 3.3.3 gives the statement of the Immutability Guarantee of FIJ. For any expression `e` that is well-typed, if `e` with store $\mathcal{H}$ reduces to `e'` with store $\mathcal{H}'$, then all locations in $\mathcal{H}$ that have immutable types will have unchanged fields in $\mathcal{H}'$.

Theorem 3.3.3. *If* $e; \mathcal{H} \longrightarrow e'; \mathcal{H}'$, $\Gamma, \Delta \vdash e : T$, $e' \neq$ `err`, *and* $\mathcal{H}$ *is well-typed with respect to* Δ *then* $\forall \iota$ *if* $\Delta(\iota) =$ `C <imm>`, $\mathcal{H}(\iota) = (T, \overline{v})$ *and* $\mathcal{H}'(\iota) = (T', \overline{v'})$ *then* $\overline{v} = \overline{v'}$

Below is the Coq encoding of the Immutability Guarantee. As with *Preservation*, the Immutability Guarantee makes use of the mutual induction scheme for `reduction`. The statement is a straightforward encoding of Theorem 3.3.3.

```
(forall p p', reduction p p' ->
     (forall e H e' H' Delta Gamma T,
             (e, H) = p -> (e',H') = p' ->
             subtyping Gamma Delta e T ->
             e' <> e_err ->
             store_well_typed Delta H ->
             (forall i C T T' vs vs', i < stLength H ->
              store_typing_lookup i Delta = C <<immutable>>->
              store_lookup i H = (T, vs) ->
              store_lookup i H' = (T', vs') ->
              vs = vs'))) /\
(forall p p', ListReduction p p' ->
     (forall es H es' H' Delta Gamma Ts,
             (es, H) = p -> (es', H') = p' ->
             subtypings Gamma Delta es Ts ->
             ~ In e_err es' ->
             store_well_typed Delta H ->
             (forall i C T T' vs vs', i < stLength H ->
              store_typing_lookup i Delta = C<<immutable>>->
```

```
store_lookup i H = (T, vs) ->
store_lookup i H' = (T', vs') ->
vs = vs'))).
```

Chapter 4

Featherweight Generic Java

In this Chapter, we present the Featherweight Generic Java (FGJ) type system and its Coq encoding. Section 4.1 provides a brief introduction to FGJ and an overview of the type system, Section 4.2 presents the type system and its encoding, while Section 4.3 gives the an overview of the Soundness proofs of FGJ.

4.1 FGJ

Featherweight Generic Java extends Featherweight Java with *Generic Types* [6]. Generic types allow normal FJ classes to be written using types that are only defined at runtime. We would like to parametrize types, and delay defining those parameters until runtime in order to allow for code reuse in structurally identical classes. An example would be a normal `Pair` class that contains a pair of objects `fst` and `snd`. In different circumstances we would want an instance of `Pair` to contain fields of different types. `Pair` without generics is given below, along with two classes `C` and `D`.

```
1 class Pair extends Object{
2         Object fst, snd;
3         Pair(Object fst, Object snd){
```

```
4                   this.fst = fst; this.snd = snd;
5           }
6   }
7   class C extends Object{C(){}}
8   class D extends Object{D(){}}
```

A `Pair` instance can be initialized with fields of any type, however once initialized, those fields may only be treated as having type `Object` unless we use a cast. To illustrate this, the `Pair` object `p` below is initialized using fields of type `C` and `D`.

```
1 Pair p = new Pair(new C(), new D());
2 Object fst1 = p.fst; // type checks
3 C fst2 = (C) p.fst; // type checks
4 C fst3 = p.fst; // results in a type error
```

The field access `p.fst` has type `Object`, thus line 2 above type checks. We know that the value stored in `fst` has type `C` since we just initialized it, thus the cast on line 3 may be done and we can be sure it is safe. Line 4 however, does not type check since the type system cannot be sure that `p.fst` has type `C`. This is inconvenient if we want to create a `Pair` with elements of type `C` and `D`, and we do not want to lose that type information. We could define a new class, say `CDPair`, that has `fst` and `snd` fields of type `C` and `D` respectively.

```
1 class CDPair extends Object{
2           C fst; D snd;
3           Pair(C fst, D snd){
4                   this.fst = fst; this.snd = snd;
5           }
6   }
```

`CDPair` allows us to initialize a pair with fields of type `C` and `D` without losing any type information, however, this is not very efficient since both

`Pair` and `CDPair` have the same structure. It is more efficient to define one class that has fields of variable type that can be changed depending on the need. Generics in Java allows classes to be parametrized with variable types that are defined at runtime. Using *generics*, we can rewrite the `Pair` class.

```
1 class Pair<X extends Object, Y extends Object>
2                           extends Object{
3         X fst; Y snd;
4         Pair(X fst, Y snd){
5                 this.fst = fst; this.snd = snd;}
6 }
```

`Pair` is now parametrized with `X` and `Y`. Both `X` and `Y` are bound by `Object`, that is `X` and `Y` must both subtype `Object`. If we wanted to create a `Pair` object with elements of type `C` and `D` as in `CDPair`, we can make use of the new `Pair` class.

```
1 Pair <C,D> cd = new Pair<C,D>(new C(), new D());
```

4.2 FGJ Type System

4.2.1 FGJ Syntax

The syntax is shown in Figure 4.1. An expression (`e`) may be a null expression (`null`), a location (ι), a variable (`x`) an error (`err`), a field access (`e.f`), a field assignment (`e.f` = `e`), a method invocation (`e.m<`$\overline{\texttt{T}}$`>(`$\overline{\texttt{e}}$`)`), a new expression (`new N(`$\overline{\texttt{e}}$`)`) or a sequence (`e; e`). A value (`v`) is either `null` or ι. A type (`T`) can be either a type variable (`X`) or a non-variable type (`N`). A non-variable type is given by a class and list of types (`C <`$\overline{\texttt{T}}$`>`). A class declaration (`L`) is a class name, a list of type variables and their bounds, a super type, a list of fields and their types, a constructor and a list of methods. A constructor (`K`) calls the constructor of the superclass and then as-

e ::= null | ι | x | err | new N($\overline{\text{e}}$) | e.f | e.f = e | e.m<$\overline{\text{T}}$>($\overline{\text{e}}$) | e; e
v ::= null | ι
T ::= X | N
N ::= C<$\overline{\text{T}}$>
L ::= class C<$\overline{\text{X}}$ ◁ $\overline{\text{N}}$> extends N{$\overline{\text{T}}$ $\overline{\text{f}}$; K $\overline{\text{M}}$}
K ::= N ($\overline{\text{T}}$ $\overline{\text{f}}$){super($\overline{\text{f}}$); this.$\overline{\text{f}}$ = $\overline{\text{f}}$}
M ::= T m<$\overline{\text{X}}$ ◁ $\overline{\text{N}}$> ($\overline{\text{T}}$ $\overline{\text{x}}$){return e;}
Pr ::= $\overline{\text{L}}$; e

Figure 4.1: FGJ Syntax

signs values to the appropriate fields. A method declaration (M) is a return type, a list of generic type variables and their bounds, a method name, list of expression parameters and their types, and a method body. A program (Pr) is a list of class declarations and an expression.

The encoding for types is shown below. Types (T) and type lists ($\overline{\text{T}}$) are encoded mutually as ty and tys respectively. A type (ty) may be constructed in one of two ways, Ty_Var (X in Figure 4.1) or Ty_Class (N in Figure 4.1). A type variable Ty_Var X is constructed using the unique identifier X of type var. var is defined, but not shown. An object of var is just an identifier used to differentiate between variables. A non-variable type Ty_Class C Ts is constructed with a class C (type class), and a type list Ts, this corresponds to a type with class C and a generic parameters Ts. class like var is merely a unique identifier, this time for class names. A type list (tys) is constructed inductively as either an empty list (empty) or a type concatenated with another type list (Tys). Throughout the encoding (T ;; Ts) is the notation for a non-empty list of types, where T is the head and Ts is the tail. The notation C <<Ts>>is used for non-variable types, where C is the class, and Ts is the type list representing the generic parameters.

```
Inductive ty : Set :=
  | Ty_Var : var -> ty
  | Ty_Class : class -> tys -> ty
```

```
with tys : Set :=
  | empty : tys
  | Tys : ty -> tys -> tys.

Notation "T ';;' Ts" := (Tys T Ts)(at level 0).

Notation "C '<<' Ts '>>'" := (Ty_Class C Ts)(at level 0).
```

Given that there already is a Coq `list` type as described in Section 2.2, it may seem strange that a new list type `tys` was defined instead of simply using `list ty`. A large part of the FGJ encoding was devoted to the encoding of type substitution. Type substitution makes use of a function `subst_ty` defined in Section 4.2.2. Proving properties about both types the substitution of types requires mutual induction of types and type lists. This means that both types and type lists, and substitution and substitution of lists have to be defined inductively. While this is an advantage when mutually reasoning about both types and type lists, it does mean the predefined functions for list can not be used for type lists. For this reason, unless a case requires mutual induction, the Coq `list` type is used.

We often need to convert from a type list (`tys`) to a list of types (`list ty`). The functions `toList` and `toTys` are used for this purpose. The encodings are not given here as they are very simple functions. `toList` converts an instance of `tys` an instance of `list ty`, while `toTys` converts an instance of type `list tys` to one of type `tys`.

Below is the encoding for the expressions from Figure 4.1. These are largely unchanged from the FIJ expressions in Section 3.2.1. Differences can be seen in the `e_meth` and `e_new` expressions that both require a type parameter list as input of type `list ty` in the case of `e_meth`, and `tys` for `e_new`.

```
Inductive exp : Type :=
  | e_null    : exp
```

```
  | e_loc     : nat -> exp
  | e_var     : var -> exp
  | e_err     : exp
  | e_new     : class -> tys -> list exp -> exp
  | e_field   : exp -> field -> exp
  | e_assign  : exp -> field -> exp -> exp
  | e_meth    : exp -> meth -> list ty -> list exp -> exp
  | e_seq     : exp -> exp -> exp.
```

Values are encoded below as an inductively defined predicate. `value e` implies that expression `e` is a value. `v_null` provides the rule for null expressions (`e_null`) to be values, and `v_loc` for locations (`e_loc n`).

```
Inductive value : exp -> Prop :=
  | v_null : value e_null
  | v_loc : forall n, value (e_loc n).
```

4.2.2 FGJ Substitution

The addition of generic types to FJ [6] creates several complications for the encoding, the most obvious being the substitution of type parameters that have to be done for every type and method invocation. In this section, the encoding of type substitution in FGJ is given. Expression substitution is unchanged from that of FIJ in Section 3.2.2.

Substitution of types is similar to that of expressions, and substitutes a variable-type mapping into a type. We define variable-type mappings as environments (`env`) below as a list of variable-type pairs (`list (var * ty)`).

```
Notation env := (list (var * ty)).
```

`subst_ty` below takes an environment (`E`) and a type (`T`), and returns a type with all the variables in the type substituted for relevant mapping in `E`. For a type variable `Ty_Var x`, `E` is searched (using the function `get`

as in Section 3.2.2) for a mapping `(x,N)`, returning `N` if one is found, and `Ty_Var x` otherwise. In the case of a non-variable type `C <<Ts>>`, `subst_ty` recursively performs the substitution on `Ts` using the mutually defined function `subst_tys`. `subst_tys` is the substitution function defined for type lists (of type `tys`). `subst_tys` takes an environment `E` and a type list `Ts`, and returns a type list with the variables in the environment substituted for their mappings. Substitution into `empty` type lists returns `empty`. Substitution into non-empty type lists substitutes the environment into the head, and then recursively applies the substitution into the tail.

```
Function subst_ty (E : env) (T : ty) {struct T} : ty :=
  match T with
  | Ty_Var x => match get x E with
                | None => T
                | Some N => N
                end
  | Ty_Class C Ts => Ty_Class C (subst_tys E Ts)
end

with subst_tys (E : env) (Ts : tys) {struct Ts} : tys :=
  match Ts with
  | empty => empty
  | T;;Ts' => (subst_ty E T);;(subst_tys E Ts')
end.
```

Another function, `subst_pair`, is not shown here but is used often to apply type substitution to a pair where the second element is a type. `subst_pair R (x,T)` simply returns `(x,subst_ty R T)`. The notations `[R] T` and `[:R:] P` below are used instead of `subst_ty R T` and `subst_pair R P` respectively, throughout the encoding for simplification.

```
Notation "'[' R ']' T" := (subst_ty R T)(at level 0).

Notation "'[:' R ':]'" := (subst_pair R)(at level 0).
```

$$bound_E(\texttt{X}) = E(\texttt{X}) \qquad bound_E(\texttt{N}) = \texttt{N}$$

Figure 4.2: FGJ Bound Function

4.2.3 FGJ Functions

The bound function in Figure 4.2 is used to determine the bound of a type for a given environment. When checking the well-formedness of expressions and types when some types may be variables, we need to be able to determine the upper bound of a type. The bound of a type variable is used when type checking field or method call receivers. For a type variable X and an environment E, $bound_{\texttt{E}}(\texttt{X}) = \texttt{E}(\texttt{X})$. Since we will always be working with well-formed types and expressions, there will never be a case where X is not in E. Well-formedness of types and expressions are found in Sections 4.2.4 and 4.2.5 respectively. For a non-variable type C<$\overline{\texttt{T}}$> and an environment E, $bound(\texttt{C<}\overline{\texttt{T}}\texttt{>}) = \texttt{C<}\overline{\texttt{T}}\texttt{>}$. The encoding for the *bound* function (bound) is given below as an inductive predicate on an environment and two types. The predicate holds if the second type is the bound of the first in the supplied environment. B_Var and B_Class provide the two cases.

```
Inductive bound : env -> ty -> ty -> Prop :=
  | B_Var :   forall E x T,
              E maps x to T ->
              bound E (Ty_Var x) T
  | B_Class : forall E C Ts,
              bound E C <<Ts>> C<<Ts>>.
```

The field lookup functions are given in Figure 4.3. The function *fields* takes a type as input, and returns a list of fields and their respective types. *fType* takes a field name and a type and returns the field's type. The Coq encoding of these rules is shown below. fields and fType encode *fields* and *fType* respectively. fields takes a type T and a list of field-type pairs

$$\frac{}{fields(\texttt{Object<>}) = \emptyset}\ (\text{F-OBJ})$$

$$\frac{\texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{> extends N}\{\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{;}\ \texttt{K}\ \overline{\texttt{M}}\}}{fields(\texttt{C<}\overline{\texttt{T}}\texttt{>}) = [\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{S}}\ \overline{\texttt{f}} \cup fields([\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N})}\ (\text{F-CLASS}) \qquad \frac{\texttt{T f} \in fields(\texttt{C<}\overline{\texttt{T}}\texttt{>})}{fType(\texttt{f}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = \texttt{T}}$$

Figure 4.3: FGJ Field Lookup Function

`fs` as inputs, and holds if `fs` is the list of fields defined for type `T`. `F_Obj` and `F_Class` provide the encodings for F-OBJ and F-CLASS respectively. `fields` holds for a type `Object <<empty>>` and an empty list of fields (`F_Obj`). `fields C <<Ts>> (concat (map [:R:] Cf) Df)` holds if `C` is in the class table `CT`, `Cf` is the list fields declared in `C` and `Df` is the list of fields defined for the super type `[R] N`. `R` is the substitution map created by zipping the type parameters `Ts` and the type variables `Xs`. `R` is then substituted into the class field list (`map [:R:] Cf`) and the super type (`[R] N`).

```
Inductive fields : ty -> flds -> Prop :=
  | F_Obj   : fields Object <<empty>> nil
  | F_Class : forall C Cf Df ms Xs N Ts R,
              In (cDecl C Xs N Cf ms) CT ->
              zip Xs (toList Ts) R ->
              fields ([R] N) Df ->
              fields C <<Ts>> (concat (map [:R:] Cf) Df).
```

The `fType` predicate below encodes *fType*. `fType T fi Ti` holds if there exists a list of field-type pairs `fs` for type `T`, and `(fi,Ti)` is in `fs`.

```
Definition fType (T : ty)
                 (fi : field) (Ti : ty) : Prop :=
    exists fs, fields T fs /\ In (fi,Ti) fs.
```

The main difference between the method lookup functions given here in Figures 4.4 and 4.5 and those in the original FGJ type system [6], is the

order in which the type substitutions are made in *mType* and *mBody*. In the original FGJ type system, the substitution of the generic type parameters for the method was applied after retrieval of the method type and body, while the generic type parameters of the class were substituted during retrieval. This is not possible to do during the Coq encoding of *mType* and *mBody*, and so the method retrieval functions were changed to reflect the encoding.

Given a class declaration `class C<`$\overline{\texttt{X}}$ $\triangleleft$ $\overline{\texttt{N}}$`> extends N{`$\overline{\texttt{T}}$ $\overline{\texttt{f}}$; $\overline{\texttt{M}}$`}`, a method declaration $\texttt{T}_0$ `m<`$\overline{\texttt{Y}}$ $\triangleleft$ $\overline{\texttt{P}}$`>` $(\overline{\texttt{x}})$`{return` $\texttt{e}_0$`;}` $\in \overline{\texttt{M}}$, and a method call `e.m<`$\overline{\texttt{V}}$`>`$(\overline{\texttt{e}})$, where `e` has type `C<T>`, we have two substitutions that need to be applied; $[\overline{\texttt{T}}/\overline{\texttt{X}}]$ and $[\overline{\texttt{V}}/\overline{\texttt{Y}}]$. These substitutions can be applied to a type `T` in one of two ways; (i) concurrently as $[\overline{\texttt{T}}/\overline{\texttt{X}}, \overline{\texttt{V}}/\overline{\texttt{Y}}]\texttt{T}$, or (ii) consecutively as $[\overline{\texttt{T}}/\overline{\texttt{X}}][\overline{\texttt{V}}/\overline{\texttt{Y}}]\texttt{T}$. These two substitutions are not necessarily equal, in fact $[\overline{\texttt{T}}/\overline{\texttt{X}}, \overline{\texttt{V}}/\overline{\texttt{Y}}]\texttt{T} = [\overline{\texttt{T}}/\overline{\texttt{X}}][\overline{\texttt{V}}/\overline{\texttt{Y}}]\texttt{T} \iff \overline{\texttt{V}} \cap \overline{\texttt{X}} = \emptyset$. We cannot ensure that $\overline{\texttt{V}} \cap \overline{\texttt{X}} = \emptyset$, since $\overline{\texttt{X}}$ is declared within the class declaration and $\overline{\texttt{V}}$ is determined externally in some method call. For this reason, in order to ensure the substitution of type variables during method retrieval is done correctly, type substitution must be done concurrently. On paper the difference between these two is not addressed since the intent is clear, but Coq forces us to take these differences into account. In the original FGJ type system [6], these substitutions are applied consecutively. To apply them concurrently, they must both be applied either during or after method retrieval. In order to minimize the places type substitution needs to be applied, we decided to encapsulate the type substitution of methods within the method retrieval functions *mType* and *mBody*(see Figures 4.4 and 4.5).

Figure 4.4 gives the *mType* function. $mType(\texttt{m<}\overline{\texttt{V}}\texttt{>}, \texttt{C<}\overline{\texttt{T}}\texttt{>})$ retrieves the type of method `m` for a receiver of type `C<`$\overline{\texttt{T}}$`>` and generic type parameters `m<`$\overline{\texttt{V}}$`>`. MT-CLASS returns the type of a method call where the method is declared in the class declaration of the receiver. MT-SUPER returns the type of a method where the method is inherited from the super type of the receiver.

$$\frac{\texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{> extends N}\{\overline{\texttt{S}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \quad \texttt{T}_0\ \texttt{m<}\overline{\texttt{Y}} \lhd \overline{\texttt{P}}\texttt{>}\ (\overline{\texttt{U}}\ \overline{\texttt{x}})\{\texttt{return e;}\} \in \overline{\texttt{M}}}{mType(\texttt{m<}\overline{\texttt{V}}\texttt{>}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = [\overline{\texttt{T}}/\overline{\texttt{X}}, \overline{\texttt{V}}/\overline{\texttt{Y}}](\texttt{<}\overline{\texttt{Y}} \lhd \overline{\texttt{P}}\texttt{>}\overline{\texttt{U}} \rightarrow \texttt{T}_0)} \quad \text{(MT-CLASS)}$$

$$\frac{\texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{> extends N}\{\overline{\texttt{S}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \quad \forall\, \texttt{T}_0, \overline{\texttt{Y}}, \overline{\texttt{P}}, \overline{\texttt{U}}, \overline{\texttt{x}}, \texttt{e} : \texttt{T}_0\ \texttt{m<}\overline{\texttt{Y}} \lhd \overline{\texttt{P}}\texttt{>}\ (\overline{\texttt{U}}\ \overline{\texttt{x}})\{\texttt{return e;}\} \notin \overline{\texttt{M}}}{mType(\texttt{m<}\overline{\texttt{V}}\texttt{>}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = mType(\texttt{m<}\overline{\texttt{V}}\texttt{>}, [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N})} \quad \text{(MT-SUPER)}$$

Figure 4.4: FGJ Method Type Lookup Function

Below is the encoding of the *mtype* function. `MT_Class` and `MT_Super` correspond to MT-CLASS and MT-SUPER respectively in Figure 4.4. `mtype m T Vs (YP,xs,T0)` holds if `(YP,xs,T0)` is the type of method call `m` with generic type parameters `Vs` on a receiver of type `T`, where `YP` is generic type environment of `m`, `xs` the expression environment and `T0` is the return type of `m`.

```
Inductive mtype : meth -> ty -> list ty ->
                      env * env * ty -> Prop :=
  | MT_Class : forall C Xs N fs ms m YP T0 xs
                      e0 ZQ ys U0 Ts R1 R2 R Vs,
               In (cDecl C Xs N fs ms) CT ->
               In (mDecl m YP T0 xs e0) ms ->
               zip Xs (toList Ts) R1 ->
               zip YP Vs R2 ->
               R = concat R1 R2 ->
               (ZQ,ys,U0) =
                  (map [:R:] YP, map [:R:] xs, [R] T0) ->
               mtype m C <<Ts>> Vs (ZQ,ys,U0)
  | MT_Super : forall C Xs N fs ms m YP T0 xs Ts R Vs,
               In (cDecl C Xs N fs ms) CT ->
              (forall ZQ U0 ys e,
```

$$\frac{\texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{> extends N}\{\overline{\texttt{S}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \quad \texttt{T}_0\ \texttt{m<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{> (}\overline{\texttt{U}}\ \overline{\texttt{x}}\texttt{)}\{\texttt{return e;}\} \in \overline{\texttt{M}}}{mBody(\texttt{m<}\overline{\texttt{V}}\texttt{>}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = (\overline{\texttt{x}}, [\overline{\texttt{T}}/\overline{\texttt{X}}, \overline{\texttt{V}}/\overline{\texttt{Y}}]\texttt{e})} \quad \text{(MB-CLASS)}$$

$$\frac{\texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{> extends N}\{\overline{\texttt{S}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \quad \forall\, \texttt{T}_0, \overline{\texttt{Y}}, \overline{\texttt{P}}, \overline{\texttt{U}}, \overline{\texttt{x}}, \texttt{e} : \texttt{T}_0\ \texttt{m<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{> (}\overline{\texttt{U}}\ \overline{\texttt{x}}\texttt{)}\{\texttt{return e;}\} \notin \overline{\texttt{M}}}{mBody(\texttt{m<}\overline{\texttt{V}}\texttt{>}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = mBody(\texttt{m<}\overline{\texttt{V}}\texttt{>}, [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N})} \quad \text{(MB-SUPER)}$$

Figure 4.5: FGJ Method Body Lookup Function

```
  ~ In (mDecl m ZQ U0 ys e) ms) ->
zip Xs (toList Ts) R ->
mtype m ([R] N) Vs (YP, xs, T0) ->
mtype m C <<Ts>> Vs (YP, xs, T0).
```

Figure 4.5 shows the FGJ method body retrieval function. As with *mType*, the substitution of the method and class type variables is done during method retrieval. MB-CLASS retrieves the body of a method declared in the class of the receiver. MB-SUPER retrieves the body of a method inherited from the receiver's super type.

Below is the encoding for *mBody*; `mbody`. `mbody m Vs T (xs,e)` holds if `(xs,e)` is the body of of method call `m` with generic parameters `Vs` on a receiver of type `T`, where `xs` is the parameter list of `m`, and `e` is the body. `mbody`, as with `mtype`, is inductively defined, and has two cases, `MB_Class` corresponding to MB-CLASS and `MB_Super` corresponding to MB-SUPER.

```
Inductive mbody : meth -> list ty -> ty ->
                          env * exp -> Prop :=
  | MB_Class : forall C Xs N fs ms m YP T0 xs
                      e0 ys e1 Ts Vs R1 R2 R,
               In (cDecl C Xs N fs ms) CT ->
```

$$\frac{mType(\mathtt{m}\!<\!\overline{\mathtt{Y}}\!>, \mathtt{N}) = \!<\!\overline{\mathtt{Z}} \lhd \overline{\mathtt{Q}}\!>\!\overline{\mathtt{U}} \to \mathtt{U}_0 \Rightarrow \quad (\overline{\mathtt{P}} = \overline{\mathtt{Q}} \qquad \overline{\mathtt{T}} = \overline{\mathtt{U}} \qquad \overline{\mathtt{Y}} \lhd \overline{\mathtt{P}} \vdash \mathtt{T}_0 <: \mathtt{U}_0)}{override(\mathtt{m}, \mathtt{N}, \!<\!\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\!>\!\overline{\mathtt{T}} \to \mathtt{T}_0)}$$

Figure 4.6: FGJ Override Function

```
                In (mDecl m YP T0 xs e0) ms ->
                zip Xs (toList Ts) R1 ->
                zip YP Vs R2 ->
                R = concat R1 R2 ->
                (ys,e1) = (map [:R:] xs, subst_exp R e0) ->
                mbody m Vs C <<Ts>> (ys, e1)
  | MB_Super : forall C Xs N fs ms m xs e0 Ts Vs R,
                In (cDecl C Xs N fs ms) CT ->
               (forall ZQ U0 ys e,
                      ~ In (mDecl m ZQ U0 ys e) ms) ->
                zip Xs (toList Ts) R ->
                mbody m Vs ([R] N) (xs,e0) ->
                mbody m Vs C <<Ts>> (xs,e0).
```

Figure 4.6 shows the FGJ override function. $override(\mathtt{m}, \mathtt{N}, \!<\!\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\!>\!\overline{\mathtt{T}} \to \mathtt{T}_0)$ holds if method $\mathtt{m}$ with type $\!<\!\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\!>\!\overline{\mathtt{T}} \to \mathtt{T}_0$ overrides $\mathtt{m}$ in type $\mathtt{N}$. That is, if $\mathtt{m}\!<\!\overline{\mathtt{Y}}\!>$ has type $\!<\!\overline{\mathtt{Z}} \lhd \overline{\mathtt{Q}}\!>\!\overline{\mathtt{U}} \to \mathtt{U}_0$ for a receiver type $\mathtt{N}$, then $\overline{\mathtt{P}} = \overline{\mathtt{Q}}$, $\overline{\mathtt{T}} = \overline{\mathtt{U}}$ and $\mathtt{T}_0$ subtypes $\mathtt{U}_0$. The Coq encoding of $override$ is given below.

```
Definition override (m : meth)(N : ty)(YP : env)
                    (Ts : list ty)(T0 : ty) : Prop :=
  meth_def m N ->
    exists ZQ ys U0,
    mtype m N (toVars (dom YP)) (ZQ,ys,U0) /\
    (range YP = range ZQ /\
     Ts = range ys /\
     subtype YP T0 U0).
```

$$E \vdash \texttt{T} <: \texttt{T} \quad \text{(S-REFL)} \qquad \frac{E \vdash \texttt{S} <: \texttt{T} \qquad E \vdash \texttt{T} <: \texttt{U}}{E \vdash \texttt{S} <: \texttt{U}} \quad \text{(S-TRANS)}$$

$$E \vdash \texttt{X} <: \texttt{E(X)} \quad \text{(S-VAR)} \qquad \frac{\texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{> extends N}\{\overline{\texttt{S}}\ \overline{\texttt{f}};\ \overline{\texttt{M}}\}}{E \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} <: [\overline{\texttt{X}}/\overline{\texttt{T}}]\texttt{N}} \quad \text{(S-EXTEND)}$$

Figure 4.7: FGJ Subtyping Rules

`override m N YP Ts T0` holds if a method with generic type environment `YP`, parameter type `Ts` and return type `T0` overrides `m` in type `N`. There is a difference between the encoded `override` function, and *override* from Figure 4.6. The predicate `meth_def` is defined, but not shown. `meth_def m N` holds if method `m` is defined for type `N`, i.e. if `m` is a valid method call on a receiver type `N`. `meth_def` is used instead of `mtype` (as in Figure 4.6) due to a peculiarity in the encoding. The *override* function uses a logical implication since we want to say, if a method is defined for a super class, then the type of the method in the subclass conforms to some restrictions. In the encoding it is possible for a method `m` to be defined for a type `N`, but for `mtype m N ...` not to hold. This is because the `mtype` predicate includes a premise `zip YP Vs R2`, which requires that the method type parameters `YP` and the method call type parameters `Vs` have the same cardinality. This means that you could define a method in a subclass with the same name as that in a super class but with a different number of type parameters, and the `mtype m N ...` premise would not hold, which would imply `mtype m N ...` $\Rightarrow P$ would hold for all P. To get around this, we use `meth_def` instead of `mtype`.

4.2.4 FGJ Subtyping and Well-Formedness

The FGJ subtyping rules are given in Figure 4.7, and the Coq encoding is given below. The subtype rules are the same as those in FJ [6], except for

$$E \vdash \texttt{Object<>} : \texttt{ok} \quad \text{(WF-OBJECT)} \qquad \frac{\texttt{X} \in E(\texttt{X})}{E \vdash \texttt{X} : \texttt{ok}} \quad \text{(WF-VAR)}$$

$$\frac{\begin{array}{c}\texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{> extends N}\{\overline{\texttt{S}}\ \overline{\texttt{f}};\ \overline{\texttt{M}}\} \\ E \vdash \overline{\texttt{T}} : \texttt{ok} \qquad E \vdash [\overline{\texttt{X}}/\overline{\texttt{T}}]\texttt{N} : \texttt{ok} \qquad E \vdash \overline{\texttt{T}} <: [\overline{\texttt{X}}/\overline{\texttt{T}}]\overline{\texttt{N}}\end{array}}{E \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} : \texttt{ok}} \quad \text{(WF-CLASS)}$$

Figure 4.8: FGJ Type Well-Formedness

the addition of the S-VAR type rule. A type variable `X` subtypes `N` in E, if $E(\texttt{X}) = \texttt{N}$. The encoding of the subtype judgment is also very similar to the encoded FIJ subtype judgment, but now takes an environment `E` as input along with the types `S` and `T`. `subtype E S T` holds if `S` is a subtype of `T` in environment `E`. `S_Var` corresponds to S-VAR.

```
Inductive subtype : env -> ty -> ty -> Prop :=
  | S_Refl  : forall E T,
                subtype E T T
  | S_Trans : forall E S T U,
                subtype E S T ->
                subtype E T U ->
                subtype E S U
  | S_Var   : forall E x N,
                E maps x to N ->
                subtype E (Ty_Var x) N
  | S_Class : forall E C Xs N fs ms Ts Rs,
                In (cDecl C Xs N fs ms) CT ->
                zip Xs (toList Ts) Rs ->
                subtype E (C <<Ts>>) ([Rs] N).
```

Figure 4.8 provides the well-formedness rules for FGJ, and the encoding is found below as `WF_ty`. The predicate `WF_tys` is encoded along with `WF_ty` as a well-formedness predicate for type lists. `WF_ty` takes

three inputs; an environment, a type and a class table. `WF_ty E T CTbl` holds if type `T` is a well-formed type with respect to environment `E` and class table `CTbl`. `WF_tys E Ts CTbl` holds if type list `Ts` is well-formed with respect to environment `E` and class table `CTbl`. `WF_ty` is the only place a `ClassTable` is used as a parameter. This is something that was tried in a previous encoding, was found to not be useful, and is not passed as an input in any other predicate. In later encodings it may be useful to remove `CTbl` as a parameter to `WF_ty` altogether, but for now it is just a relic of an older encoding. In practice, the `ClassTable` parameter is always replaced with the generic `ClassTable` instance `CT` discussed in Chapter 3.2.1.

The predicate `WF_ty` has three cases, each corresponding to a rule in Figure 4.8. `WF_Obj` encodes WF-OBJECT, and specifies that a type, of class `Object`, is well-formed if the generic parameters list is empty. `WF_Var` encodes WF-VAR. A type variable `X` is well-formed in environment `E` if there exists `T` such that `E` maps `X` to `T` (in other words, if `X` is in the domain of `E`). `WF_Class` encodes WF-CLASS. A non-variable type `C<<Ts>>` is well-formed for an environment `E` and a class table `CTbl` if the following holds: `C` is defined in `CTbl` (`(cDecl C Xs N fs ms)`), the substitution `R` can be constructed from generic parameters `Ts` and the generic class parameters `Xs` (`zip Xs (toList Ts) R`), `Ts` is well-formed with respect to `E` (`WF_tys E Ts CTbl`), `Ts` subtypes the generic type parameter bounds of class `C` (`subtypes E (toList Ts) (map [R] (range Xs))`), and super type `[R] N` must also be well-formed with respect to `E`.

The predicate `WF_tys` is not given, but recursively performs wellformedness checks on a type list (`tys`). `WF_tys E Ts CTbl` holds if all types in `Ts` are well-formed with respect to `E` and `CTbl`.

```
Inductive WF_ty : env -> ty -> ClassTable-> Prop :=
  | WF_Obj   : forall E CTbl,
               WF_ty E Object <<empty>> CTbl
  | WF_Var   : forall E x CTbl,
```

$$\frac{E \vdash \mathtt{N} : \mathtt{ok} \qquad fields(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad E;\Gamma;\Delta \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}} \qquad E \vdash \overline{\mathtt{S}} <: \overline{\mathtt{T}}}{E;\Gamma;\Delta \vdash \mathtt{new\ N}(\overline{\mathtt{e}}) : \mathtt{N}} \text{ (T-NEW)}$$

$$\frac{E;\Gamma;\Delta \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad fType(\mathtt{f}, bound_{\mathrm{E}}(\mathtt{T}_0)) = \mathtt{T}}{E;\Gamma;\Delta \vdash \mathtt{e}_0.\mathtt{f} : \mathtt{T}} \text{ (T-FIELD)}$$

$$\frac{\begin{array}{c} E;\Gamma;\Delta \vdash \mathtt{e}_0 : \mathtt{T}_0 \\ E;\Gamma;\Delta \vdash \mathtt{e} : \mathtt{T} \qquad fType(\mathtt{f}, bound_{\mathrm{E}}(\mathtt{T}_0)) = \mathtt{S} \qquad E \vdash \mathtt{T} <: \mathtt{S} \end{array}}{E;\Gamma;\Delta \vdash \mathtt{e}_0.\mathtt{f} = \mathtt{e} : \mathtt{T}} \text{ (T-ASSIGN)}$$

$$\frac{\begin{array}{c} E;\Gamma;\Delta \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad E \vdash \overline{\mathtt{V}} : \mathtt{ok} \qquad E;\Gamma;\Delta \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}} \qquad E \vdash \overline{\mathtt{T}} <: \overline{\mathtt{U}} \\ E \vdash \overline{\mathtt{V}} <: \overline{\mathtt{P}} \qquad mType(\mathtt{m}\texttt{<}\overline{\mathtt{V}}\texttt{>}, bound_{\mathrm{E}}(\mathtt{T}_0)) = \texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\texttt{>}\overline{\mathtt{U}} \to \mathtt{T} \end{array}}{E;\Gamma;\Delta \vdash \mathtt{e}_0.\mathtt{m}\texttt{<}\overline{\mathtt{V}}\texttt{>}(\overline{\mathtt{e}}) : \mathtt{T}} \text{ (T-INVK)}$$

Figure 4.9: FGJ Expression Typing Rules

```
            (exists T, E maps x to T) ->
            WF_ty E (Ty_Var x) CTbl
| WF_Class : forall E C Xs N fs ms CTbl Ts R,
            In (cDecl C Xs N fs ms) CTbl ->
            zip Xs (toList Ts) R ->
            WF_tys E Ts CTbl ->
            subtypes E (toList Ts) (map [R](range Xs)) ->
            WF_ty E ([R] N) CTbl ->
            WF_ty E (C <<Ts>>) CTbl.
```

4.2.5 FGJ Expression Typing

The FGJ expression typing rules are given in Figure 4.9. An expression `e` is said to be well-typed with respect to type `T`, type environment E, environment Γ and store typing Δ if $E;\Gamma;\Delta \vdash \mathtt{e} : \mathtt{T}$ holds. Typing for `null` expressions (T-NULL), variables (T-VAR), locations (T-LOC) and sequences (T-SEQ) are unchanged from Chapter 3, and so are omitted. A

new expression `new N(`$\overline{\texttt{e}}$`)` has type `N` in E, Γ and Δ if `N` is well-formed in E, Γ and Δ, and the types of $\overline{\texttt{e}}$ subtype the field types $\overline{\texttt{T}}$ of `N`. A field access $\texttt{e}_0\texttt{.f}$ has type `T` in E, Γ and Δ if $\texttt{e}_0$ has type $\texttt{T}_0$, and the field type of `f` for the bound of $\texttt{T}_0$ in E has type `T` (T-FIELD). A field assignment $\texttt{e}_0\texttt{.f} = \texttt{e}$ has type `T` in E, Γ and Δ if $\texttt{e}_0$ has type $\texttt{T}_0$, the type of `e` subtypes the field type of `f` for the bound of $\texttt{T}_0$ in E (T-ASSIGN). A method invocation `e.m<`$\overline{\texttt{V}}$`>(`$\overline{\texttt{e}}$`)` has type `T` in E, Γ and Δ if $\texttt{e}_0$ has type $\texttt{T}_0$, method `m` has return type `T` for the bound of $\texttt{T}_0$, the generic type parameters $\overline{\texttt{V}}$ are well-formed and subtype the generic type parameter bounds of `m` and the types of $\overline{\texttt{e}}$ subtype the parameter types of `m` (T-INVK).

The encodings of the type rules are given next. `typing E Gamma Delta e T` holds if expression `e` has type `T` with respect to type environment `E`, environment `Gamma` and store typing `Delta`. Apart from the addition of the type environment `E`, the rules for `null` (T-NULL), variables (T-VAR), locations (T-LOC) and sequences (T-SEQ) are unchanged from Chapter 3, and so are omitted. Descriptions for the `subtyping` and `subtypings` rules can also be found in Chapter 3.

`T_New` below encodes T-NEW. An expression `e_new C Ts es` has type `C <<Ts>>` if the parameters `es` are well-typed with respect to the field types of `C <<Ts>>`. In other words, if `fs` is the list of field-type pairs associated with type `C <<Ts>>`; then the types of `es` must subtype `range fs` (`subtypings E Gamma Delta es (range fs)`). `C <<Ts>>` must also be a well-formed type (`WF_ty E (C <<Ts>>) CT`).

```
Inductive typing : env -> env ->
           store_typing -> exp -> ty -> Prop :=
  | T_New   : forall E Gamma Delta C es fs Us Ts,
              fields C <<Ts>> fs ->
              subtypings E Gamma Delta es (range fs) ->
              WF_ty E (C <<Ts>>) CT ->
              typing E Gamma Delta (e_new C Ts es) C <<Ts>>
```

`T_Field` below, encodes T-FIELD. `T_Field` checks that the receiver's

type `T0` has a bound (`N0`) in type environment `E`, and that field `f` has type `T` for a receiver type `T0'` (`ftype N0 f T`).

```
| T_Field    : forall E Gamma Delta e0 T0 T0' f T,
               typing E Gamma Delta e0 T0 ->
               bound E T0 N0 ->
               ftype N0 f T ->
               typing E Gamma Delta (e_field e0 f) T
```

T-ASSIGN is encoded below by `T_Assign`. This features the same prerequisites as `T_Field`, except the requirement that the assigned expression `e` is well-typed, and subtypes the field type `Ti`.

```
| T_Assign   : forall E Gamma Delta e0 fi Ti e T T0 N0,
               typing E Gamma Delta e0 T0 ->
               typing E Gamma Delta e T ->
               bound E T0 N0 ->
               fType N0 fi Ti ->
               subtype E T Ti ->
               typing E Gamma Delta (e_assign e0 fi e) T
```

`T_Invk` encodes T-INVK. A method invocation `e_meth e0 m Vs es` has type `T` if the receiver `e0` has type `T0`, and method `m` with type parameters `Vs` has type `(YP,xs,T)` in type `N`, where `N` is the bound of `T0` in type environment `E`. Since the substitution of type parameters is handled by `mtype`, no substitution is needed here. Therefore, `Vs` must subtype the range of `YP` (`subtypes E Vs (range YP)`), i.e. the generic type bounds of `m`. The expression parameters `es` must subtype `range xs`, or the expression parameter types (`subtypings E Gamma Delta es (range xs)`).

```
| T_Invk     : forall E Gamma Delta e0 T0 N es YP T m xs Vs,
               typing E Gamma Delta e0 T0 ->
               bound E T0 N ->
               (forall V, In V Vs -> WF_ty E V CT) ->
               mtype m N Vs (YP,xs,T) ->
```

$$\frac{\forall \mathtt{X}, : \exists \mathtt{N}, \ \mathtt{N} = E(\mathtt{X}) \wedge E \vdash \mathtt{N} : \mathtt{ok}}{E \ \mathtt{ok}} \quad \text{(T-TYPE-ENV)}$$

$$\frac{\forall \mathtt{x}, : \ E \vdash \Gamma(\mathtt{x}) : \mathtt{ok}}{E \vdash \Gamma : \mathtt{ok}} \quad \text{(T-ENV)}$$

Figure 4.10: FGJ Method and Class Typing Rules

```
subtypes E Vs (range YP) ->
subtypings E Gamma Delta es (range xs) ->
typing E Gamma Delta (e_meth e0 m Vs es) T
```

The well-formedness of environments is given in Figure 4.10. A type environment `E` is well-formed if the range of `E` is well-formed with respect to `E`, and contains only non-variable types (T-TYPE-ENV). An environment Γ is well-formed with respect to a type environment E if the range of Γ is well-formed with respect to E. The encodings of T-TYPE-ENV and T-ENV are given below.

```
Definition WF_type_env (E : env) (CTbl : ClassTable):Prop :=
  (ok E) /\ (forall x N, E maps x to N ->
              (nonvar_type N /\ WF_ty E N CTbl)).

Definition WF_env (E G : env): Prop :=
  (ok G) /\ (forall x T, In (x,T) G -> WF_ty E T CT).
```

`WF_type_env` encodes T-TYPE-ENV and `WF_env` encodes T-ENV. Both `WF_type_env` and `WF_env` make use of the `ok` predicate. `ok E` holds for a list of `A`-`B` pairs `E`, where `A` and `B` are generic Coq types, if each element of the domain of `E` is unique. Since `ok` works for any Coq types `A` and `B`, we can use it for any list of Coq pairs. Thus, for a type environment `E`, `ok E` ensures that `E` is a function. Similarly, `ok G` ensures that environment `G` is a function. E and Γ in Figure 4.10 are assumed to be functions, and so

$$\frac{\begin{array}{c}\texttt{class C<}\overline{\texttt{X}}\triangleleft\overline{\texttt{N}}\texttt{> extends N\{...\}} \qquad E = \overline{\texttt{X}}\triangleleft\overline{\texttt{N}}, \overline{\texttt{Y}}\triangleleft\overline{\texttt{P}} \\ E; \texttt{this} : \texttt{C<}\overline{\texttt{X}}\texttt{>}, \overline{\texttt{x}} : \overline{\texttt{T}}; \emptyset \vdash \texttt{e} : \texttt{S} \qquad E \vdash \texttt{T} : \texttt{ok} \qquad E \vdash \overline{\texttt{P}} : \texttt{ok} \\ E \vdash \overline{\texttt{T}} : \texttt{ok} \qquad E \vdash \texttt{S} <: \texttt{T} \qquad \mathit{override}(\texttt{m}, \texttt{N}, \texttt{<}\overline{\texttt{Y}}\triangleleft\overline{\texttt{P}}\texttt{>}\overline{\texttt{T}} \to \texttt{T})\end{array}}{\texttt{T m<}\overline{\texttt{Y}}\triangleleft\overline{\texttt{P}}\texttt{> (}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{)\{return e;\} OK IN C<}\overline{\texttt{X}}\triangleleft\overline{\texttt{N}}\texttt{>}} \text{ (T-METH)}$$

$$\frac{\begin{array}{c}\overline{\texttt{S}} \cap \mathit{fields}(\texttt{N}) = \emptyset \qquad \overline{\texttt{X}}\triangleleft\overline{\texttt{N}} \vdash \overline{\texttt{S}} : \texttt{ok} \\ \overline{\texttt{X}}\triangleleft\overline{\texttt{N}} \vdash \texttt{N} : \texttt{ok} \qquad \forall \texttt{M} \in \overline{\texttt{M}}, \texttt{ : M OK IN C<}\overline{\texttt{X}}\triangleleft\overline{\texttt{N}}\texttt{>}\end{array}}{\texttt{class C<}\overline{\texttt{X}}\triangleleft\overline{\texttt{N}}\texttt{> extends N\{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{; }\overline{\texttt{M}}\texttt{\} OK}} \text{ (T-CLASS)}$$

Figure 4.11: FGJ Method and Class Typing Rules

a premise equivalent to `ok` is not needed. Other than `ok`, `WF_type_env` and `WF_env` are direct encodings of the rules in Figure 4.10.

The FGJ method and class declaration typing rules are given in Figure 4.11. The judgment $\texttt{T m<}\overline{\texttt{Y}}\triangleleft\overline{\texttt{P}}\texttt{> (}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{)\{return e;\} OK IN C<}\overline{\texttt{X}}\triangleleft\overline{\texttt{N}}\texttt{>}$ holds if method declaration $\texttt{T m<}\overline{\texttt{Y}}\triangleleft\overline{\texttt{P}}\texttt{> (}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{)\{return e;\}}$ is well formed in class `C` with class type environment $\overline{\texttt{X}}\triangleleft\overline{\texttt{N}}$ (T-METH). The return type, `T`, the expression parameter types $\overline{\texttt{T}}$ and the generic type parameter bounds $\overline{\texttt{P}}$ must be well-typed with respect to the generic class ($\overline{\texttt{X}}\triangleleft\overline{\texttt{N}}$) and method parameters ($\overline{\texttt{Y}}\triangleleft\overline{\texttt{P}}$). The type of method body `e` must subtype the return type. Further, if $\texttt{C<}\overline{\texttt{X}}\texttt{>}$ extends some type `N`, then the method must override any method by the same name in `N`.

The judgment $\texttt{class C<}\overline{\texttt{X}}\triangleleft\overline{\texttt{N}}\texttt{> extends N\{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; }\overline{\texttt{M}}\texttt{\} OK}$ holds if the class declaration $\texttt{class C<}\overline{\texttt{X}}\triangleleft\overline{\texttt{N}}\texttt{> extends N\{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; }\overline{\texttt{M}}\texttt{\}}$ is well formed (T-CLASS). Since field overriding is not allowed, the intersection between the declared fields and the fields of the super type must be empty. The super type and the field types $\overline{\texttt{S}}$ must be well-formed with respect to the generic type parameters.

`meth_ok` below, encodes T-METH. It is encoded as an `Inductive` predicate instead of as a `Definition`, as in previous encodings, to make

it easier break up the predicate into its premises during proofs. `meth_ok decl C XN` holds if method declaration `decl` is well-typed in class `C` with respect to type environment `XN`. The types within the `decl` must be well-formed with respect to the class type environment `XN` and the method type environment. These are concatenated to form `E` (`E = concat XN YP`), which must be well-formed (`WF_type_env E CT`). The environment `G = ((this, C <<toTys (toVars (dom XN))>>)::xs)` is constructed from the method environment `xs` appended with `(this, C <<toTys (toVars (dom XN))>>)`, i.e. `this` has type `C <<toTys (toVars (dom XN))>>`, where `toTys (toVars (dom XN))` is the domain of `XN`, and corresponds to $\overline{X}$ in T-METH. `G` must be well formed. The method body `e0` must be well-typed with respect to the return type `T0`, or some subtype of `T0` (`subtyping E ((this,C<<toTys (toVars (dom XN))>>)::xs) nil e0 T0`). The method signature, `YP (range xs) T0`, must match that of any method defined for the super type `D <<Ys>>` (`override m D <<Ys>> YP (range xs) T0`).

```
Inductive meth_ok : MethDecl -> class ->
                                  list (var * ty) -> Prop :=
  | GT_Method : forall decl C N XN m YP T0 xs e0 E G fs ms,
      In (cDecl C XN N fs ms) CT ->
      decl = mDecl m YP T0 xs e0 ->
      E = concat XN YP ->
      G = ((this, C <<toTys (toVars (dom XN))>>)::xs) ->
      WF_type_env E CT -> WF_env E G -> WF_ty E T0 CT ->
      subtyping E G nil e0 T0 ->
      override m N YP (range xs) T0 ->
      meth_ok decl C XN.
```

Below is the encoding of T-CLASS, `class_ok`. `class_ok decl` holds if `decl` is well-formed in the class table `CT`. For a class declaration (`cDecl C XN Ys fs ms`) to be well-formed, the type environment `XN` must be well-formed (`WF_type_env XN CT`), and all types used in the declaration must be well-formed with respect to `XN`. That is, the super type `N` (`WF_ty XN N`

CT) and all field types (forall Ti fi, In (fi, Ti) fs -> WF_ty XN Ti CT). There must be no duplicate field or method names (ok fC and ok_meths ms). Lastly, all method declarations must be well formed for class C and type environment XN (M OK_IN C XN)).

```
Inductive class_ok : ClassDecl -> Prop :=
  |GT_Class :
     forall C XN N ms fs decl, decl = cDecl C XN N fs ms ->
       WF_type_env XN CT -> WF_ty XN N CT ->
       nonvar_type N -> ok_meths ms ->
       (forall Ts fC, fields C <<Ts>> fC -> ok fC) ->
       (forall Ti fi, In (fi, Ti) fs -> WF_ty XN Ti CT) ->
       (forall M, (In M ms -> M OK_IN C XN)) ->
       class_ok decl.
```

4.2.6 FGJ Reduction

The FGJ reduction rules are shown in Figure 4.12. The congruency reduction rules are not given, and are the usual congruency rules as in Chapter 2.1.5. Expression reduction takes place in the context of a store. $\texttt{e};\mathcal{H} \longrightarrow \texttt{e'};\mathcal{H}'$ asserts that an expression e with a store $\mathcal{H}$ reduces to e' with a store $\mathcal{H}'$. A new expression, new N($\overline{\texttt{v}}$) with store $\mathcal{H}$, reduces to some location ι with $\mathcal{H}'$ where ι is not in the domain of $\mathcal{H}$ and $\mathcal{H}'$ is the store derived by adding ι to $\mathcal{H}$ (R-NEW). A field access $\iota.\texttt{f}_i$ reduces to the field value $\texttt{v}_i$ stored in $\mathcal{H}(\iota)$ (R-FIELD). A field assignment $\iota.\texttt{f}_j = \texttt{v}$ with store $\mathcal{H}$, reduces to v with store $\mathcal{H}'$, where $\mathcal{H}'$ is the store derived from replacing the field value of f in $\mathcal{H}(\iota)$ with v (R-ASSIGN). A method invocation ι.m<$\overline{\texttt{V}}$>(e) with store $\mathcal{H}$ reduces to $[\iota/\texttt{this}, \overline{\texttt{v}}/\overline{\texttt{x}}]\texttt{e}$ with store $\mathcal{H}$, where $\overline{\texttt{x}}$ are the parameters of m (R-INVK). A sequence v; e with store $\mathcal{H}$ reduces to e with store $\mathcal{H}$ (R-SEQ). As with FIJ in Chapter 3 the congruency (the reduction of sub-expressions) and null (the reduction of field or method calls on null receivers) reduction rules are omitted.

$$\frac{\iota \notin dom(\mathcal{H}) \qquad \mathcal{H}' = \mathcal{H}, \iota \mapsto \texttt{new N}(\overline{\texttt{v}})}{\texttt{new N}(\overline{\texttt{v}})|\mathcal{H} \longrightarrow \iota|\mathcal{H}'} \quad \text{(R-NEW)}$$

$$\frac{\mathcal{H}(\iota) = \texttt{new N}(\overline{\texttt{v}}) \qquad fields(\texttt{N}) = \overline{\texttt{T}}\ \overline{\texttt{f}}}{\iota.\texttt{f}_i|\mathcal{H} \longrightarrow \texttt{v}_i|\mathcal{H}'} \quad \text{(R-FIELD)}$$

$$\frac{\mathcal{H}(\iota) = \texttt{new N}(\overline{\texttt{v}}) \qquad \begin{array}{c} fields(\texttt{N}) = \overline{\texttt{T}}\ \overline{\texttt{f}} \\ \mathcal{H}' = \mathcal{H}[\iota \mapsto \texttt{new N}(..., \texttt{v}_{j-1}, \texttt{v}, \texttt{v}_{j+1}, ...)] \end{array}}{\iota.\texttt{f}_j = \texttt{v}|\mathcal{H} \longrightarrow \texttt{v}|\mathcal{H}'} \quad \text{(R-ASSIGN)}$$

$$\frac{\mathcal{H}(\iota) = \texttt{new N}(...) \qquad mBody(\texttt{m<}\overline{\texttt{V}}\texttt{>}, \texttt{N}) = (\overline{\texttt{x}}; \texttt{e})}{\iota.\texttt{m<}\overline{\texttt{V}}\texttt{>}(\overline{\texttt{v}})|\mathcal{H} \longrightarrow [\iota/\texttt{this}, \overline{\texttt{v}}/\overline{\texttt{x}}]\texttt{e}|\mathcal{H}} \quad \text{(R-INVK)}$$

$$\frac{}{\texttt{v; e}|\mathcal{H} \longrightarrow \texttt{e}|\mathcal{H}} \quad \text{(R-SEQ)}$$

Figure 4.12: FGJ Reduction Rules

The encodings of the reduction rules are shown next. They are split up here for convenience, but, along with the reduction rules for congruence, make up a single predicate in the encoding. The notation for a reduction, `e / H --> e' / H'`, is interpreted as expression `e` with store `H` reduces to expression `e'` with store `H'`

`R_New` encodes R-NEW below. A `new` expression `e_new C Ts vs` with store `H` reduces to `e_loc i` with store `H'`. The constructor parameters `vs` must be values (`values vs`). The list of field value pairs `fv` is constructed by zipping `fs` and `vs` together (`zip fs vs fv`), where `fs` is the list of fields associated with type `C <<Ts>>` (`fields (C <<Ts>>) fs`). `H'` is constructed by appending `(C<<Ts>>, fv)` to the end of `H`. In R-NEW, ι must not be in the domain of $\mathcal{H}$. In the Coq encoding `R_New`, `H` (representing $\mathcal{H}$) is a list, that means new locations must be appended to the end of `H`. This is done by letting the position `i` of the new location be the length of `H` (`leng H = i`). The location `e_loc i` now points to the `i`th position

of H', and is the reduced expression.

```
Inductive reduction : exp * store -> exp * store -> Prop :=
  | R_New : forall H H' i C vs fs fv Ts,
            leng H = i ->
            values vs ->
            fields (C <<Ts>>) fs ->
            zip fs vs fv ->
            H' = stSnoc H (C <<Ts>>, fv) ->
            (e_new C Ts vs) / H --> (e_loc i) / H'
```

R_Field encodes R-FIELD below. For a field access e_field (e_loc i) f with store H, the location in H at position i is retrieved (lookup i H = Some (N,vs)). In this instance, N is the type, and vs is the list of field-value pairs, representing the field values. The value v corresponding to f is then retrieved from vs (vs maps f to v).

```
  | R_Field : forall N i H vs f v,
              lookup i H = Some (N,vs) ->
              vs maps f to v ->
              (e_field (e_loc i) f) / H --> v / H
```

R_Assign encodes R-ASSIGN below. A field assignment e_assign (e_loc i) f v with store H reduces to v with store H'. v is a value (value v). Location i is retrieved from store H (lookup i H = Some (N,vs)), and the index (j) of f in vs is found (lookup j vs = Some (f, e)). H' is constructed by replacing the object at i in H with a new object (N,vs') (H' = replace i (N,vs') H). vs' is constructed by replacing the field-value pair in vs at position j with (f,v) (vs' = replace j (f, v) vs).

```
  | R_Assign : forall H H' N i j e vs vs' v f,
               lookup i H = Some (N,vs) ->
               value v ->
               lookup j vs = Some (f, e) ->
               vs' = replace j (f, v) vs ->
```

```
            H' = replace i (N,vs') H ->
            e_assign (e_loc i) f v / H --> v / H'
```

R_Invk encodes R-INVK below. A method invocation e_meth (e_loc i) m Vs vs with store H reduces to (subst ((this,e_loc i)::R) e) with store H. vs must be values (values vs). The object at location i is retrieved, and the method body (xs, e) for the object's type N is retrieved (mbody m Vs N (xs, e)). zip xs vs R constructs the substitution function R by zipping the the method environment xs and the values vs. The reduced expression is now the method body with the parameters substituted into it, along with e_loc i substituted for this.

```
| R_Invk : forall H i N m xs vs e R es Vs,
           lookup i H = Some (N,es) ->
           values vs ->
           mbody m Vs N (xs, e) ->
           zip xs vs R ->
           (e_meth (e_loc i) m Vs vs) / H -->
                        (subst ((this,e_loc i)::R) e) / H
```

R_Seq encodes R-SEQ below. If v is a value (value v), then e_seq v e with store H reduces to e with store H.

```
| R_Seq    : forall v e H,
            value v ->
            e_seq v e / H --> e / H
```

Reductions on lists of expressions are needed for congruence reduction rules. ListReduction, defined below, is the Coq encoding of reduction for a list of expressions. ListReduction is defined mutually with the reduction predicate. ListReduction (es,H) (es',H') holds if list of expressions es with store H reduces to es' with store H'. R_Head is the rule describing a reduction of the head of a list, and R_Tail describes the reduction of the tail.

```
with ListReduction : (list exp) * store ->
```

```
                    (list exp) * store -> Prop :=
  | R_Head : forall H H' e e' E,
             e / H --> e' / H' ->
             ListReduction ((e::E), H) ((e'::E), H')
  | R_Tail : forall H H' e es es',
              ListReduction (es, H) (es', H') ->
              ListReduction ((e::es), H) ((e::es'), H')
```

The notations defined below are used to represent reductions of expressions and lists of expressions.

```
where "e1 '/' st1 '-->' e2 '/' st2" :=
                      (reduction (e1,st1) (e2,st2)).

Notation "es1 '//' st1 '==>' es2 '//' st2" :=
  (ListReduction (es1,st1) (es2,st2)) (at level 200).
```

4.3 FGJ Soundness

Soundness for FGJ is captured by two theorems *Preservation* and *Progress*, but in order for these to be proven, several lemmas involving type and expression substitution must be proven. Section 4.3.1 presents the substitution lemmas, while Sections 4.3.2 and 4.3.3 give the statements of *Preservation* and *Progress* respectively. The theorems are stated both formally and as a Coq formalism.

4.3.1 Substitution

The addition of generic types to FJ introduced type substitution to the type system. This section provides the statements of several useful lemmas needed to show that type substitution in FGJ preserves various judgements. The proofs of these Lemmas [6] are not given here. Provided here

are informal statements of proofs found in the Coq encoding. Lemma 4.3.1 states that a bound of a substituted type subtypes the substituted bound.

Lemma 4.3.1. *If* $E_2 \cup \overline{X} \triangleleft \overline{N} \vdash \texttt{T}\ ok$ *and* $E_1 \vdash \overline{U}\ ok$*, where* $E_1 \vdash \overline{U} <: [\overline{X}/\overline{U}]\overline{N}$*, then* $E_1 \cup [\overline{X}/\overline{U}]E_2 \vdash bound_{E_1 \cup [\overline{X}/\overline{U}]E_2}([\overline{X}/\overline{U}]\texttt{T}) <: [\overline{X}/\overline{U}]bound_{E_2 \cup \overline{X} \triangleleft \overline{N}}(\texttt{T})$

Proof. By case analysis on the bound function. The cases where T is a non-variable type and T is a variable type in $dom(E_2)$ are trivial. The case where T is some $\texttt{X}_i \in \overline{\texttt{X}}$ follows from $E_1 \vdash \overline{\texttt{U}} <: [\overline{\texttt{X}}/\overline{\texttt{U}}]\overline{\texttt{N}}$. □

```
Lemma type_substitution_preserves_bound :
  forall E1 XN E2 Us R,
    zip XN Us R ->
    subtypes E1 Us (map [R] (range XN)) ->
    (forall E T N, bound E T N ->
      E = (concat XN E2) ->
      (forall E',
        E' = (concat E1 (map [:R:] E2)) ->
        WF_type_env E CT ->
        WF_type_env E' CT ->
        WF_ty E T CT ->
        (forall U, In U Us -> WF_ty E1 U CT) ->
        (forall N0, bound E' ([R] T) N0 ->
                    subtype E' N0 ([R] N)))).
```

Lemma 4.3.2 states that the subtype relationship is preserved by type substitution.

Lemma 4.3.2. *If* $E_2 \cup \overline{X} \triangleleft \overline{N} \vdash \texttt{S} <: \texttt{T}$ *and* $E_1 \vdash \overline{U}\ ok$*, where* $E_1 \vdash \overline{U} <: [\overline{X}/\overline{U}]\overline{N}$*, then* $E_1 \cup [\overline{X}/\overline{U}]E_2 \vdash [\overline{X}/\overline{U}]\texttt{S} <: [\overline{X}/\overline{U}]\texttt{T}$

Proof. By induction on the derivation of $E_2 \cup \overline{\texttt{X}} \triangleleft \overline{\texttt{N}} \vdash \texttt{S} <: \texttt{T}$. Cases S-REFL, S-TRANS and S-CLASS are trivial. Case S-VAR where S is in $dom(E_2)$ is easy. In the case where $\texttt{S} = \texttt{X}_i \in \overline{\texttt{X}}$, the proposition follows from $E_1 \vdash \overline{\texttt{U}} <: [\overline{\texttt{X}}/\overline{\texttt{U}}]\overline{\texttt{N}}$. □

```
Lemma type_substitution_preserves_subtype :
  forall E1 XN E2 Us R,
    zip XN Us R ->
    subtypes E1 Us (map ([R]) (range XN)) ->
    (forall E S T, subtype E S T ->
          E = concat XN E2 ->
          (forall E',
            E' = (concat E1 (map ([:R:]) E2)) ->
            WF_type_env E CT ->
            (forall U, In U Us -> WF_ty E1 U CT) ->
            subtype E' ([R] S) ([R] T))).
```

Lemma 4.3.3 states that well-formedness of types is preserved by type substitution. The Coq proof requires mutual induction of type and type lists to be performed, rather than a straight induction on types. Without a mutual induction scheme, well-formedness of the type lists of class types would not be recognised as well-formed by the induction hypothesis. The statement of the Coq proof provided here only provides the statement of the proof for well-formedness of types, and not of type lists. The statement of the proof that makes use of mutual induction is omitted due to length, but can be found in the encoding.

Lemma 4.3.3. *If* $E_2 \cup \overline{X} \triangleleft \overline{N} \vdash$ `T ok` *and* $E_1 \vdash \overline{U}$ `ok`*, where* $E_1 \vdash \overline{U}$ `<:` $[\overline{X}/\overline{U}]\overline{N}$*, then* $E_1 \cup [\overline{X}/\overline{U}]E_2 \vdash [\overline{X}/\overline{U}]$`S ok`

Proof. By induction on the derivation of $E_2 \cup \overline{\texttt{X}} \triangleleft \overline{\texttt{N}} \vdash$ `T ok`. Cases WF-OBJECT and WF-CLASS are simple. Case WF-VAR (`T = X`) requires a case analysis on whether `X` $\in dom(E_2)$ or $\overline{\texttt{X}}$. `X` $\in dom(E_2)$ is easy, and follows from $E_1 \vdash \overline{\texttt{U}}$ `ok`. □

```
Lemma type_substitution_preserves_WF :
  forall E1 XN E2 Us R,
```

```
    zip XN Us R ->
    subtypes E1 Us (map ([R]) (range XN)) ->
    (forall E T CTbl, WF_ty E T CTbl ->
      CTbl = CT ->
      E = concat XN E2 ->
      (forall E',
        E' = (concat E1 (map ([:R:]) E2)) ->
        WF_type_env E CTbl ->
        (forall U, In U Us -> WF_ty E1 U CTbl) ->
        WF_ty E' ([R] T) CTbl)).
```

Lemma 4.3.4 states that expression typing is preserved by type substitution. As with Lemma 4.3.3, a mutual induction scheme is used in the Coq formalism. Mutual induction is perfomed on expression typing (`typing`), expression subtyping (`subtyping`) and expression list subtyping (`subtypings`).

Lemma 4.3.4. *If* $E_2 \cup \overline{X} \triangleleft \overline{N}; \Gamma; \Delta \vdash$ `e : T` *and* $E_1 \vdash \overline{U}$ `ok`*, where* $E_1 \vdash \overline{U}$ `<:` $[\overline{X}/\overline{U}]\overline{N}$*, then* $E_1 \cup [\overline{X}/\overline{U}]E_2; [\overline{X}/\overline{U}]\Gamma; \Delta \vdash [\overline{X}/\overline{U}]$`e : S` *and* $E_1 \cup [\overline{X}/\overline{U}]E_2 \vdash$ `S <:` $[\overline{X}/\overline{U}]$`T`*.*

Proof. By induction on the derivation of $E_2 \cup \overline{X} \triangleleft \overline{N}; \Gamma; \Delta \vdash$ `e : T` □

```
Lemma type_substitution_preserves_typing :
  forall E1 XN E2 Us R,
    zip XN Us R ->
    subtypes E1 Us (map ([R]) (range XN)) ->
    (forall E Gamma Delta e T,
      typing E Gamma Delta e T ->
      E = concat XN E2 ->
      (forall E',
        E' = (concat E1 (map ([:R:]) E2)) ->
        WF_type_env E CT ->
```

```
        WF_type_env E' CT ->
        (forall U, In U Us -> WF_ty E1 U CT) ->
        subtyping E' (map ([:R:]) Gamma) Delta
                     (subst_exp R e) ([R] T))).
```

Lemma 4.3.5 states that expression substitution preserves typing. A mutual induction scheme for expression typing, subtyping and expression list subtyping is also used in this Lemma. As before, the statement of the proof using mutual induction is omitted for brevity. Once the mutual induction scheme is applied, the proof is straight forward.

Lemma 4.3.5. *If* $E; \overline{x} \triangleleft \overline{B}; \Delta \vdash$ `e : T` *and* $E; \Gamma; \Delta \vdash \overline{e}$ `: A`, *where* $E \vdash \overline{A}$ `<: B`, *then* $E; \Gamma; \Delta \vdash [\overline{x}/\overline{e}]$`e : S` *and* $E \vdash$ `S <: T`

Proof. By induction on the derivation of $E; \overline{x} \triangleleft \overline{B}; \Delta \vdash$ `e : T`. □

```
Lemma Substitution_Preserves_Typing :
  forall E Gamma Delta xB e T es R,
    typing E xB Delta e T ->
    subtypings E Gamma Delta es (range xB) ->
    zip xB es R ->
    WF_type_env E CT ->
    subtyping E Gamma Delta (subst R e) T.
```

4.3.2 Preservation

The statement of the Subject Reduction Theorem, or *Preservation*, in FGJ is given in Theorem 4.3.1. As with previously discussed lemmas, the Coq proof of *Preservation* makes use of a mutual induction scheme. Induction is performed mutually on the `reduction` and the `ListReduction` predicates. Without this scheme, the induction hypothesis for expression lists would not be produced during induction. *Preservation* states that given a non-error, well-typed expression, any reduction of that expression would

result in an expression which is well-typed with regard to a type that is a subtype of the type of the original expression

Theorem 4.3.1 (Preservation). *If* $e; \mathcal{H} \longrightarrow e'; \mathcal{H}'$ *where* $e' \neq \texttt{err}$ *and* $E; \Gamma; \Delta \vdash \texttt{e : T}$*, then* $\exists \Delta'$ *s.t.* ($\Delta' \supseteq \Delta$ *and* $\exists S$*, s.t.* $E; \Gamma; \Delta' \vdash \texttt{e : S}$ *and* $E \vdash \texttt{S <: T}$)

Proof. By induction on the derivation of $\texttt{e}; \mathcal{H} \rightarrow \texttt{e}'; \mathcal{H}'$. □

Below is the statement of Theorem 4.3.1 in Coq. The theorem is presented as the conjunction of two propositions; one for reduction of expressions and one for the reduction of lists.

```
Theorem Preservation :
  (forall p p', reduction p p' ->
    (forall E Gamma Delta T e e' H H',
      (e,H) = p -> (e',H') = p' ->
      e' <> e_err ->
      WF_store Delta H -> ok Gamma ->
      WF_type_env E CT ->
      typing E Gamma Delta e T ->
      (exists Delta', ST_Extends Delta' Delta ->
              WF_store Delta' H' ->
              subtyping E Gamma Delta' e' T))) /\
  (forall p p', ListReduction p p' ->
    (forall E Gamma Delta Ts es es' H H',
      (es,H) = p -> (es',H') = p' ->
      ~ In e_err es' ->
      WF_store Delta H -> ok Gamma ->
      WF_type_env E CT ->
      subtypings E Gamma Delta es Ts ->
      (exists Delta', ST_Extends Delta' Delta ->
              WF_store Delta' H' ->
              subtypings E Gamma Delta' es' Ts))).
```

4.3.3 Progress

Theorem 4.3.2 is the statement of the *Progress* Theorem. *Progress* states that for all well-typed expressions, either that expression is a value, or there is some expression such that the original expression may be reduced to. Proof of the theorem is done by induction on the derivation of the typing predicate. As with *Preservation*, a mutual induction scheme is required in order to successfully perform the induction.

Theorem 4.3.2 (Progress). *$\forall e$, if $E;\Gamma;\Delta \vdash e : T$ then either;*

(a) e is a value, or

(b) $\forall \mathcal{H}$ s.t. $\mathcal{H}$ is well-typed with respect to Δ, $\exists e', \mathcal{H}'$ s.t. $e;\mathcal{H} \longrightarrow e';\mathcal{H}'$

Proof. By induction on the derivation of $E;\Gamma;\Delta \vdash$ `e : T`. □

```
Theorem Progress :
  (forall E Gamma Delta e T,
                typing E Gamma Delta e T ->
                Gamma = nil -> E = nil ->
                (value e \/
                (forall H, WF_store Delta H ->
                   exists e', exists H',
                       e / H --> e' / H'))) /\
  (forall E Gamma Delta e T,
                subtyping E Gamma Delta e T ->
                Gamma = nil -> E = nil ->
                (value e \/
                 (forall H, WF_store Delta H ->
                    exists e', exists H',
                       e / H --> e' / H'))) /\
  (forall E Gamma Delta es Ts,
                subtypings E Gamma Delta es Ts ->
```

```
Gamma = nil -> E = nil ->
(values es \/
 (forall H, WF_store Delta H ->
    exists es', exists H',
    ListReduction (es, H) (es', H')))).
```

Chapter 5

Featherweight Ownership Generic Java

Featherweight Ownership Generic Java (FOGJ) is presented in this Chapter. Section 5.1 gives an outline of the FOGJ type system, Section 5.2 describes the type system, while the FOGJ soundness proofs are described in Section 5.3.

5.1 FOGJ

Featherweight Ownership Generic Java is an extension of the FGJ in Chapter 4 with Ownership. Ownership is an object-oriented language feature that allows one object to *own* another. Ownership has several variations, but they all restrict access to owned objects to some degree based on their owner. A common variant of Ownership, and the one presented in this Chapter, is *Owners-as-Modifiers*. Under the Owners-as-Modifiers paradigm, modifications to an object (field accesses, field assignments and method invocations) may only be made by the owner of that object, while references may be held by any object.

Objects may be owned by either `World` or $\texttt{This}_l$. Objects owned by `World` may be modified by any other object, while objects owned by $\texttt{This}_l$

may only be modified by the object stored at location l.

In FOGJ, ownership information is contained as type information. The types of FGJ are extended to include owners. Therefore the owner of a type simply forms part of its generic parameter list. Including owners as types introduces two new kinds of type on top of those of FGJ: non-variable owner types $\texttt{N}_{\texttt{O}}$ and variable owner types $\texttt{X}_{\texttt{O}}$. To store owners of FOGJ types, class types are parametrised by an owner parameter: $\texttt{C<}\overline{\texttt{T}}\texttt{;O>}$ is a type of class `C`, generic parameters $\overline{\texttt{T}}$ and owner `O`. The class declaration of a class `C` is given below as an example.

```
1 class C <X extends Object<Xo>; Xo extends World>
2                                    extends Object<Xo>{
3    Object<Xo> f;}
4 ...
5 C <Object<World>;World> C_obj =
6                           new C<Object<World>;World>();
```

`C_obj` has type `C<Object<World>;World>`, with owner `World` and super type `Object<World>`. The field `f` has type `Object<Xo>` within class `C`. This shows that once we declare the owner parameter `Xo`, it may be used within the class `C`. The concept of scope is very important in FOGJ. While an object can only modify objects it owns, it may reference an object ι with owner `O` so long as `O` is in scope.

In the above example, the owner variable `Xo` is within the scope of `C`, and may be used to define fields (like `f`) and method declarations within `C`. Owners that are in scope in class declarations are `World`, any owner variables defined as generic parameters such as `Xo` above, and `This`. The `This` owner variable represents the current object, and at runtime for a specific object l, all occurrences of `This` are replaced by $\texttt{This}_l$. The `This` variable is used during type checking to prohibit the modification of objects by objects other that their owner. Field accesses, assignments and method invocations that use the `This` variable may only applied to the

$$
\begin{aligned}
\texttt{e} &::= \texttt{null} \mid \iota \mid \texttt{x} \mid \texttt{err} \mid \texttt{new N()} \mid \texttt{e.f} \mid \texttt{e.f} = \texttt{e} \mid \texttt{e.m<}\overline{\texttt{T}}\texttt{>}(\overline{\texttt{e}}) \mid \texttt{e; e} \mid \texttt{P} > \texttt{e} \\
\texttt{v} &::= \texttt{null} \mid \iota \\
\texttt{T} &::= \texttt{X} \mid \texttt{N} \\
\texttt{X} &::= \texttt{X}_{\texttt{t}} \mid \texttt{X}_{\texttt{O}} \\
\texttt{N} &::= \texttt{C<}\overline{\texttt{T}}\texttt{; O>} \mid \texttt{N}_{\texttt{O}} \\
\texttt{N}_{\texttt{O}} &::= \texttt{World} \mid \texttt{This} \mid \texttt{This}_{\iota} \\
\texttt{O} &::= \texttt{X}_{\texttt{O}} \mid \texttt{N}_{\texttt{O}} \\
\texttt{L} &::= \texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{; X}_{\texttt{O}} \lhd \texttt{N}_{\texttt{O}}\texttt{> extends N}\{\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{;}\ \overline{\texttt{M}}\} \\
\texttt{M} &::= \texttt{T m<}\overline{\texttt{X}} \lhd \overline{\texttt{T}}\texttt{>}\ (\texttt{T}\ \overline{\texttt{x}})\{\texttt{return e;}\} \\
\texttt{P} &::= \texttt{C} \mid \iota \\
\texttt{Pr} &::= \overline{\texttt{L}}\texttt{;}\ \texttt{e}
\end{aligned}
$$

Figure 5.1: FOGJ Syntax

receiver `this`. Therefore, a field with owner parameter `This` may only be accessed within the scope of the class where the field is defined.

5.2 FOGJ Type System

5.2.1 FOGJ Syntax

Figure 5.1 is the FOGJ syntax. The syntax is much the same as that of FGJ, extending types to include owners. Expressions remain the same as in FGJ. There is a single addition of a context expression `P > e` used to give a context of permission `P` to an expression `e`. A permission `P` can be either a class `C` or a location ι. Permissions are used to give an ownership context during typing and method reduction in order to identify the `This` owner. For a context expression `P > e`, `e` is within the scope of some class or object represented by `P`. The FOGJ `new` expression does not take any parameters, and all fields are initialized as `null`. This is because of the introduction of ownership. Any constructor parameters must be fully initialised before the new object is initialised. Since fields may be owned

by the new object, this can lead to a contradiction. For this reason fields must be set after the object is initialised. A value `v` is the same as in FGJ, nulls and locations. A type variable `X` can be either a class type variable $\texttt{X}_\texttt{t}$ as in FGJ, or an owner type variable $\texttt{X}_\texttt{O}$. A non-variable type may be either a class type $\texttt{C<}\overline{\texttt{T}}\texttt{;O>}$, or a non-variable owner $\texttt{N}_\texttt{O}$. The class type differs from that of FGJ by the addition of an owner type `O`, this is the owner of the class type. A non-variable owner may be `World`, `This` or $\texttt{This}_l$. `World` is the root of the ownership tree, `This` is the current object, and $\texttt{This}_l$ is the object at location l. An owner `O` may be either an owner type variable, or a non-variable owner. A class declaration `L` adds an explicit owner variable $\texttt{X}_\texttt{O}$ and bound $\texttt{N}_\texttt{O}$ to that of FGJ. Method declarations are unchanged from FGJ.

The addition of ownership types requires the introduction of three types on top of those in the FGJ encoding: owner type variables, concrete owner types and an undefined type. The Coq encoding of types can be found below.

```
Inductive ty : Type :=
  | Ty_Var   : Var -> ty
  | Ty_Class : class -> tys -> ty -> ty
  | Ty_Own   : own -> ty
  | Ty_Undf  : ty
with tys : Type :=
  | empty : tys
  | Tys : ty -> tys -> tys.
```

As in FGJ, type variables and class types are given by `Ty_Var` and `Ty_Class` respectively. A class type `Ty_Class C Ts O` has class `C`, generic parameters `Ts` and owner parameter `O`. Type variables are represented by a `Var`, a unique variable identifier. In FGJ the identifier was simply a natural number, but in FOGJ there are two types of type variables, class and owner type variables. This is captured by the `Var` definition given below.

```
Inductive Var : Type :=
 | t_Var : nat -> Var
 | o_Var : nat -> Var.
```

A `Var` is constructed in one of two ways, `t_Var` and `o_Var`, corresponding to class and owner type variables respectively. This distinction is needed throughout the encoding, and is assumed in the original OGJ type system. For ease, the following notation is used to identify class and owner type variables.

```
Notation "'X_' x" := (Ty_Var (t_Var x))(at level 0).
Notation "'O_' x" := (Ty_Var (o_Var x))(at level 0).
```

A normal type variable `Ty_Var (t_Var x)` uses the notation `X_ x`, while `Ty_Var (o_Var x)` is written as `O_ x`. The `This` variable is defined as `o_Var 0` in the encoding.

```
Definition This := o_Var 0.
```

For simplification, the following notation is used for type lists (`tys`) class types.

```
Notation "T ';;' Ts" := (Tys T Ts)(at level 0).
Notation "C '<<' Xs ';' O '>>'" :=
                        (Ty_Class C Xs O)(at level 0).
```

Under the above notation, `Tys T Ts` becomes `T ;; Ts`, while `Ty_Class C Ts O` becomes `C<<Ts;O>>`. There are two further types available in FOGJ; owner types `Ty_Own` and an undefined type `Ty_Undf`. Owner types correspond to N_O in Figure 5.1, and take an `own` parameter.

```
Inductive own : Type :=
  | O_World : own
  | O_This_ : nat -> own.
```

`own` may either be `O_World` or `O_This_ l`. `O_World` encodes `World` in the syntax, and `O_This_ l` encodes $\texttt{This}_l$, where `l` is the position of the

owner object in the store. As with class and variable types, owner types are given a notation for ease of use.

```
Notation "'World'":=(Ty_Own O_World).
Notation "'This_' l":=(Ty_Own (O_This_ l))(at level 0).
```

The final type `Ty_Undf` represents an undefined type. An undefined type is not an actual type, and is an encoding of $\perp$ to allow for type checking to catch ownership violations. The use of undefined types is discussed in Section 5.2.3 during the encoding of encoding of the $this$ function.

FOGJ expressions are very similar to those in FGJ, and are given below.

```
Inductive exp:Type :=
  | e_null   :exp
  | e_loc    :nat -> exp
  | e_var    :var -> exp
  | e_err    :exp
  | e_new    :class -> tys -> ty -> exp
  | e_field  :exp -> field -> exp
  | e_assign :exp -> field -> exp -> exp
  | e_meth   :exp -> meth -> list ty -> list exp -> exp
  | e_seq    :exp -> exp -> exp
  | e_context:nat -> exp -> exp.
```

The FOGJ context expression is encoded by `e_context l e`, where `l` is the position in the store of the current `This` object. The rest of the expressions are taken directly from FGJ, except for an owner type parameter for the new expression.

The only other peculiarity of the syntax is the addition of permissions. A permission `P` can be either a location `l` or a class `C`.

```
Inductive Permission : Type :=
  | P_Class : class -> Permission
  | P_loc   : nat -> Permission.
```

`P_Class C` encodes the class permission `C`, and `P_loc l` encodes the location permission `l`.

5.2.2 FOGJ Substitution

Substitution in FOGJ inherits much from FIJ and FGJ. FOGJ expression substitution is inherited from FIJ, and is unchanged. FOGJ type substitution is inherited from FGJ with minor changes.

In FGJ, type variables were identified in the same way as expression variables, using `var` objects (Section 4.2), that is an FGJ type variable is written `Ty_Var x`, where `x` is of type `var`. During FGJ type substitution we used the `get` function to retrieve a mapping of `x` in some environment. In Section 5.2.1 we presented the encoding for FOGJ type variables. FOGJ type variables are identified using `Var` objects. `Var` is different from `var` in that there is more than one way to construct an object of type `Var`; `t_Var` for normal type variables and `o_Var` for owner type variables. Since type variables are not defined using `var`, we can no longer use the `get` function to retrieve a type variable's mapping. For this reason, `get_Var` below was written to replace it.

```
Fixpoint get_Var {A : Type}(X : Var)
                    (E : list (Var * A)) : option A:=
  match E with
  | nil => None
  | (X',a)::E' => if (beq_Var X X') then
                    (Some a) else (get_Var X E')
end.
```

`get_Var X E` recursively searches environment `E` for a mapping of variable `X` (`Var`). If `E` is a `nil` list (empty), `get_Var X E` returns `None`. If `E` is non-empty list with head `(X',a)` and tail `E'`, then `get_Var X E` returns `Some a` if `beq_Var X X'` is true, and `get_Var X E'` otherwise. `beq_Var X X'` is an equality function that returns `true` if `X = X'`, and

$$\frac{}{fields(\texttt{Object<World>}, \texttt{T}_\texttt{O}) = \emptyset} \quad \text{(F-OBJECT)}$$

$$\frac{\texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{; X}_\texttt{O} \lhd \texttt{N}_\texttt{O}\texttt{> extends N}\{\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{;}\ \overline{\texttt{M}}\}}{\begin{array}{c} fields(\texttt{C<}\overline{\texttt{T}}\texttt{; O>}, \texttt{T}_\texttt{O}) = \\ [\texttt{T}_\texttt{O}/\texttt{This}, \texttt{O}/\texttt{X}_\texttt{O}, \overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{S}}\ \overline{\texttt{f}} \cup fields([\texttt{T}_\texttt{O}/\texttt{This}, \texttt{O}/\texttt{X}_\texttt{O}, \overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N}) \end{array}} \quad \text{(F-CLASS)}$$

$$\frac{\texttt{T f} \in fields(\texttt{C<}\overline{\texttt{T}}\texttt{; O>}, \texttt{T}_\texttt{O})}{fType(\texttt{f}, \texttt{C<}\overline{\texttt{T}}\texttt{; O>}, \texttt{T}_\texttt{O}) = \texttt{T}}$$

Figure 5.2: FOGJ Field Lookup Function

`false` if not.

5.2.3 FOGJ Functions

In Section 4.2.3, we discussed the importance of applying type substitutions concurrently. In FOGJ we still need to perform all type substitutions during field and method retrieval concurrently, and this includes the substitution of the `This` owner variable. The FOGJ field and method retrieval functions given in this section, both have an extra parameter on top of those in the original FGJ type system, the substitution of the `This` owner variable.

The FOGJ field retrieval functions *fields* and *fType* are given in Figure 5.2. The FOGJ *fields* function is much like the *fields* function of FGJ except for the addition on the $\texttt{T}_\texttt{O}$ parameter. $\texttt{T}_\texttt{O}$ is substituted for the `This` variable along with all other type variables. Similarly, *fType* also has a parameter $\texttt{T}_\texttt{O}$.

The FOGJ method type and body retrieval functions are given in Figure 5.3 and 5.4 respectively. As with the field retrieval functions they are unchanged from FGJ except for the addition of the $\texttt{T}_\texttt{O}$ parameter to substituted for the `This` variable.

$$\frac{\begin{array}{c}\texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{; X}_\texttt{O} \lhd \texttt{N}_\texttt{O}\texttt{> extends N}\{\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{; }\overline{\texttt{M}}\} \\ \texttt{T}_0\ \texttt{m<}\overline{\texttt{Y}} \lhd \overline{\texttt{Q}}\texttt{>}\ (\overline{\texttt{U}}\,\overline{\texttt{x}})\{\texttt{return e;}\} \in \overline{\texttt{M}}\end{array}}{\begin{array}{c}mType(\texttt{m}, \texttt{C<}\overline{\texttt{T}}\texttt{; O>}, \texttt{T}_\texttt{O}) = \\ [\texttt{T}_\texttt{O}/\texttt{This}, \texttt{O}/\texttt{X}_\texttt{O}, \overline{\texttt{T}}/\overline{\texttt{X}}, \overline{\texttt{V}}/\overline{\texttt{Y}}](\texttt{<}\overline{\texttt{Y}} \lhd \overline{\texttt{Q}}\texttt{>}\overline{\texttt{U}} \to \texttt{T}_0)\end{array}} \quad \text{(MT-CLASS)}$$

$$\frac{\begin{array}{c}\texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{; X}_\texttt{O} \lhd \texttt{N}_\texttt{O}\texttt{> extends N}\{\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{; }\overline{\texttt{M}}\} \\ \forall\, \texttt{T}_0, \overline{\texttt{Y}}, \overline{\texttt{Q}}, \overline{\texttt{U}}, \overline{\texttt{x}}, \texttt{e} : \texttt{T}_0\ \texttt{m<}\overline{\texttt{Y}} \lhd \overline{\texttt{Q}}\texttt{>}\ (\overline{\texttt{U}}\,\overline{\texttt{x}})\{\texttt{return e;}\} \notin \overline{\texttt{M}}\end{array}}{mType(\texttt{m}, \texttt{C<}\overline{\texttt{T}}\texttt{; O>}, \texttt{T}_\texttt{O}) = mType(\texttt{m}, [\texttt{O}/\texttt{X}_\texttt{O}, \overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N}, \texttt{T}_\texttt{O})} \quad \text{(MT-SUPER)}$$

Figure 5.3: FOGJ Method Type Lookup Function

$$\frac{\begin{array}{c}\texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{; X}_\texttt{O} \lhd \texttt{N}_\texttt{O}\texttt{> extends N}\{\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{; }\overline{\texttt{M}}\} \\ \texttt{T}_0\ \texttt{m<}\overline{\texttt{Y}} \lhd \overline{\texttt{Q}}\texttt{>}\ (\overline{\texttt{U}}\,\overline{\texttt{x}})\{\texttt{return e;}\} \in \overline{\texttt{M}}\end{array}}{mBody(\texttt{m<}\overline{\texttt{V}}\texttt{>}, \texttt{C<}\overline{\texttt{T}}\texttt{; O>}, \texttt{T}_\texttt{O}) = (\overline{\texttt{x}}; [\texttt{T}_\texttt{O}/\texttt{This}, \texttt{O}/\texttt{X}_\texttt{O}, \overline{\texttt{T}}/\overline{\texttt{X}}, \overline{\texttt{V}}/\overline{\texttt{Y}}]\texttt{e})} \quad \text{(MB-CLASS)}$$

$$\frac{\begin{array}{c}\texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{; X}_\texttt{O} \lhd \texttt{N}_\texttt{O}\texttt{> extends N}\{\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{; }\overline{\texttt{M}}\} \\ \forall\, \texttt{T}_0, \overline{\texttt{Y}}, \overline{\texttt{Q}}, \overline{\texttt{U}}, \overline{\texttt{x}}, \texttt{e} : \texttt{T}_0\ \texttt{m<}\overline{\texttt{Y}} \lhd \overline{\texttt{Q}}\texttt{>}\ (\overline{\texttt{U}}\,\overline{\texttt{x}})\{\texttt{return e;}\} \notin \overline{\texttt{M}}\end{array}}{mBody(\texttt{m<}\overline{\texttt{V}}\texttt{>}, \texttt{C<}\overline{\texttt{T}}\texttt{; O>}, \texttt{T}_\texttt{O}) = mBody(\texttt{m<}\overline{\texttt{V}}\texttt{>}, [\texttt{O}/\texttt{X}_\texttt{O}, \overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N}, \texttt{T}_\texttt{O})} \quad \text{(MB-SUPER)}$$

Figure 5.4: FOGJ Method Body Lookup Function

The FOGJ *override* and *bound* functions can be found in Figures 5.5 and 5.6, and are unchanged from FGJ.

The *owner* function is given in Figure 5.7. $owner_E(\texttt{T})$ returns the owner of type `T` in type environment E. If `T` is a non-owner type variable $\texttt{X}_\texttt{t}$, *owner* returns the owner of $\texttt{X}_\texttt{t}$'s bound in E. The owner of a class type $\texttt{C<}\overline{\texttt{T}}\texttt{; O>}$, is `O`. The owner of an ownership type `O` is `O`. The *owner* function is encoded as `owner` below. `owner E T O` holds if `O` is the owner of type `T` in environment `E`. `O_TVar` encodes the *owner* function for non-owner type variables. For a type variable `X_ x`, with bound `N` in environment `E`, if `O` is the owner of `N` in `E`, then `O` is the owner of `X_ x` in `E`.

$$\frac{mType(\texttt{m<}\overline{\texttt{Y}}\texttt{>}, \texttt{N}, \texttt{This}) = (\texttt{<}\overline{\texttt{Z}} \lhd \overline{\texttt{R}}\texttt{>}\overline{\texttt{U}} \to \texttt{U}_0) \Rightarrow \\ (\overline{\texttt{Q}}, \overline{\texttt{T}}) = [\texttt{Y}/\texttt{Z}](\overline{\texttt{R}}, \overline{\texttt{U}}) \;\&\; \overline{\texttt{Y}} \lhd \overline{\texttt{Q}}; \emptyset \vdash \texttt{T}_0 <: [\texttt{Y}/\texttt{Z}]\texttt{U}_0}{override(\texttt{m}, \texttt{N}, \texttt{<}\overline{\texttt{Y}} \lhd \overline{\texttt{Q}}\texttt{>}\overline{\texttt{T}} \to \texttt{T}_0)}$$

Figure 5.5: FOGJ Override Function

$$bound_E(\texttt{X}_\texttt{t}) = E(\texttt{X}_\texttt{t}) \qquad bound_E(\texttt{C<}\overline{\texttt{T}}\texttt{;O>}) = \texttt{C<}\overline{\texttt{T}}\texttt{;O>}$$

Figure 5.6: FOGJ Bound Function

```
Inductive owner : ty_env -> ty -> ty -> Prop :=
  | O_TVar  : forall E x N O,
              bound E X_ x N ->
              owner E N O ->
              owner E X_ x O
```

The rest of the rules are direct encodings of the rest of the cases in Figure 5.7.

```
  | O_OVar  : forall E x,
              owner E O_ x O_ x
  | O_Class : forall E C Ts O,
              owner E C <<Ts;O>> O
  | O_Own   : forall E O,
              owner E (Ty_Own O) (Ty_Own O).
```

The various cases; `O_OVar`, `O_Class` and `O_Own` correspond to the owner function for owner type variables, class types and owner types respectively. The encodings are straightforward.

The *this* function is given in Figure 5.8. The *this* function is used to ensure ownership is not violated. Ownership restrictions in FOGJ are en-

$$owner_E(\texttt{X}_\texttt{t}) = owner_E(bound_E(\texttt{X}_\texttt{t})) \qquad owner_E(\texttt{C<}\overline{\texttt{T}}\texttt{;O>}) = \texttt{O}$$
$$owner_E(\texttt{O}) = \texttt{O}$$

Figure 5.7: FOGJ Owner Function

$$this_\texttt{C}(this) = \texttt{This} \qquad this_\iota(\iota) = \texttt{This}_\iota \qquad this_\texttt{P}(...) = \bot$$

Figure 5.8: FOGJ $this$ Function

sured by restricting field accesses, assignments and method calls involving the `This` type variable to the `this` receiver. $this_\texttt{P}(\texttt{e})$ returns the value of the `This` type variable for a receiver `e` in context with permission `P`. When the permission is a class `C`, and the receiver is `this`, `This` has value `This`, or $this_\texttt{C}(\texttt{this}) = \texttt{This}$. When the permission is an object ι, and the receiver is ι, `This` has a value of $\texttt{This}_\iota$, or $this_\iota(\iota) = \texttt{This}_\iota$. For all other cases, `This` has an undefined value, or $this_\texttt{P}(...) = \bot$.

$this$ is encoded below as `this_P`, a Fixpoint function that takes a permission (`Permission`) and an expression (`exp`), and returns a type (`ty`). It is a simple encoding that makes use of `Ty_Undf` for $\bot$. The purpose of `Ty_Undf` is solely to be ill-formed, to allow ownership violations to be caught during type checking.

```
Fixpoint this_P (P : Permission) (e : exp) : ty :=
  match P with
  | P_Class C => if (eq_exp e (e_var this))
                   then (Ty_This) else Ty_Undf
  | P_loc l => if (eq_exp e (e_loc l))
                 then (This_ l) else Ty_Undf
end.
```

`this_P P e` can return one of three types. If the permission `P` is a class

$$E; \Delta \vdash \mathtt{T} <: \mathtt{T} \quad \text{(S-REFL)} \qquad \frac{E; \Delta \vdash \mathtt{S} <: \mathtt{T} \qquad E; \Delta \vdash \mathtt{T} <: \mathtt{U}}{E; \Delta \vdash \mathtt{S} <: \mathtt{U}} \quad \text{(S-TRANS)}$$

$$E; \Delta \vdash \mathtt{X} <: E(\mathtt{X}) \quad \text{(S-VAR)}$$

$$\frac{\mathtt{class\ C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}; \mathtt{X_O} \lhd \mathtt{N_O}\mathtt{>\ extends\ N}\{\overline{\mathtt{S}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\}}{E; \Delta \vdash \mathtt{C<}\overline{\mathtt{T}}; \mathtt{O>} <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}, \mathtt{O}/\mathtt{X_O}]\mathtt{N}} \quad \text{(S-EXTEND)}$$

$$\frac{\Delta(l) = \mathtt{T} \qquad \mathit{owner}_E(\mathtt{T}) = \mathtt{O}}{E; \Delta \vdash \mathtt{This}_l <: \mathtt{O}} \quad \text{(S-OWNER)}$$

Figure 5.9: FOGJ Subtyping Rules

permission, and matches some `P_Class C`, then `Ty_This` is returned if the receiver `e` equals the `this` variable `e_var this`, and `Ty_Undf` otherwise. If `P` is a location permission `P_loc l`, then `This_ l` is returned if `e` equals the location `e_loc l`, and `Ty_Undf` otherwise.

`this_P` is defined as a **`Fixpoint`** definition rather than an inductive one for ease of use later. A **`Fixpoint`** function can be used instead of its result. Since `this_P` is a relatively simple function as compared to `mtype`, or even `bound`, it is much easier to use a **`Fixpoint`** definition.

5.2.4 FOGJ Subtyping and Well-Formedness

Subtyping in FOGJ is given in Figure 5.9. $E; \Delta \vdash \mathtt{S} <: \mathtt{T}$ holds if `S` subtypes `T` with respect to type environment E and store type environment Δ. The subtyping rules are as in FGJ, except for the addition of S-OWNER and a store type environment as context. An object owner $\mathtt{This}_l$ subtypes `O` in type environment E and store type environment Δ if `O` is the owner of `T` in E, and `T` is the type of location l in Δ. As in Figure 5.9, the Coq encoding of the FOGJ subtype rules is largely unchanged from FGJ. The encoding of S-OWNER is given below, but is part of the larger `subtype` predicate.

```
| S_Owner : forall E D l O T,
            lookup l D = Some T ->
            owner E T O ->
            subtype E D (This_ l) O
```

For a location `l` with type `T` in store type environment `D`, where `T` has owner `O` in type environment `E`, `subtype E D (This_ l) O` holds.

Before well-formedness can be encoded, a secondary predicate must be defined, `is_owner`.

```
Inductive is_owner : ty -> Prop :=
  | Is_Var : forall x,
             is_owner O_ x
  | Is_Own : forall O,
             is_owner (Ty_Own O).
```

`is_owner O` holds if `O` is an owner, i.e. if `O` is a owner variable, or a non-variable owner. Since they are defined separately and not as a single "owner" type in the syntax, to identify an owner as separate from other types, this predicate is defined. In OGJ [17] it is assumed that owners can be distinguished from normal types.

Type well-formedness in FOGJ is defined in Figure 5.10. The Coq encoding of type well-formedness is the predicate `WF_ty`, and is encoded in the same manner as well-formedness in FGJ. The header for `WF_ty` is:

```
Inductive WF_ty : ty_env -> list ty ->
                      ty -> ClassTable-> Prop :=
```

`WF_ty E D T CTbl` holds if type `T` is well-formed in type environment `E`, and store type environment `D`, with class table `CTbl`. `Object<O>` is well-formed if `O` is well-formed (WF-OBJECT). `WF_Obj` below encodes WF-OBJECT. `O` must be an owner (`is_owner O`), and must be well-formed (`WF_ty E D O CTbl`).

$$\frac{E;\Delta \vdash \mathtt{O} : \mathtt{ok}}{E;\Delta \vdash \mathtt{Object<O>} : \mathtt{ok}}\ (\text{WF-OBJECT}) \qquad \frac{\mathtt{X} \in dom(E)}{E;\Delta \vdash \mathtt{X} : \mathtt{ok}}\ (\text{WF-VAR})$$

$$\frac{\begin{array}{c}\mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}; \mathtt{X_O} \triangleleft \mathtt{N_O}\mathtt{>\ extends\ N\{}\overline{\mathtt{S}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\mathtt{\}} \\ E;\Delta \vdash [\overline{\mathtt{T}}/\overline{\mathtt{X}}, \mathtt{O}/\mathtt{X_O}]\mathtt{N} <: \mathtt{Object<O>} \\ E;\Delta \vdash \overline{\mathtt{T}}, \mathtt{O}, [\overline{\mathtt{T}}/\overline{\mathtt{X}}, \mathtt{O}/\mathtt{X_O}]\mathtt{N} : \mathtt{ok} \qquad E;\Delta \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}, \mathtt{O}/\mathtt{X_O}]\overline{\mathtt{N}} \\ E;\Delta \vdash \mathtt{O} <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}, \mathtt{O}/\mathtt{X_O}]\mathtt{N_O} \qquad \forall\, \mathtt{T} \in \overline{\mathtt{T}},\ E;\Delta \vdash \mathtt{O} <: \mathit{owner}(\mathtt{T})\end{array}}{E;\Delta \vdash \mathtt{C<}\overline{\mathtt{T}}, \mathtt{O>} : \mathtt{ok}}\ (\text{WF-CLASS})$$

$$\frac{E;\Delta \vdash \mathtt{N_O} <: \mathtt{World}}{E;\Delta \vdash \mathtt{N_O} : \mathtt{ok}}\ (\text{WF-OWNER})$$

Figure 5.10: FOGJ Type Well-Formedness

```
| WF_Obj    : forall E D O CTbl,
              is_owner O ->
              WF_ty E D O CTbl ->
              WF_ty E D Object <<empty;O>> CTbl
```

A type variable X is well-formed in a type environment E if X is in the domain of E (WF-VAR). WF_Var is the Coq encoding of WF-VAR. For a type variable Ty_Var x to be well-formed, WF_Var simply checks if there is a mapping in the type environment E to some type.

```
| WF_Var    : forall E D x CTbl,
              (exists T, E maps x to T) ->
              WF_ty E D (Ty_Var x) CTbl
```

A class type C<$\overline{\mathtt{T}}$; O> (WF-CLASS) is well-formed if the type parameters $\overline{\mathtt{T}}$ and O subtype the bounds of the type parameters in the class declaration, and are well-formed themselves. The super type with the substituted type parameters must subtype Object<O>. The final prerequisite is that the

primary owner `O` must subtype (be "inside") the owners of all other type parameters $\overline{\texttt{T}}$. `WF_Class` encodes WF-CLASS below and checks that all subtypes are well-formed, as well as the super type. `(forall T, In T (toList Ts)->forall Q, owner E T Q->subtype E D O Q)` ensures that the primary owner `O` is nested inside the owners of the other parameters as per Section 5.1.

```
| WF_Class : forall E D C XN Xo No N fs ms R CTbl Ts O,
             In (cDecl C XN (Xo,No) N fs ms) CTbl ->
             is_owner O ->
             zip ((Xo,No)::XN) (O::(toList Ts)) R ->
             WF_tys E D O;;Ts CTbl ->
             WF_ty E D ([R] N) CTbl ->
             (forall T, In T (toList Ts) ->
                        forall Q, owner E T Q ->
                        subtype E D O Q) ->
             subtypes E D (O::(toList Ts))
                    (map [R] (range ((Xo,No)::XN))) ->
             WF_ty E D C <<Ts;O>> CTbl
```

Well-formedness for owners is one of the few differences between the FOGJ presented here and the OGJ type system. In OGJ, types were often checked to be either well-formed or to subtype `World`, while FOGJ requires non-variable owners to subtype `World` to be well-formed (WF-OWNER). This simplifies checks in the type system and cases in several lemmas when proving soundness. The Coq encoding of WF-OWNER, `WF_Owner` is given below.

```
| WF_Owner : forall E D O,
             subtype E D (Ty_Own O) World ->
             WF_ty E D (Ty_Own O) CT
```

$$\frac{E;\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{,O>} : \texttt{ok}}{E;\Gamma;\Delta;\texttt{P} \vdash \texttt{new C<}\overline{\texttt{T}}\texttt{,O>}() : \texttt{C<}\overline{\texttt{T}}\texttt{,O>}} \quad \text{(T-NEW)}$$

$$\frac{E;\Gamma;\Delta;\texttt{P} \vdash \texttt{e}_0 : \texttt{T}_0 \qquad fType(\texttt{f}, bound_E(\texttt{T}_0), this_{\texttt{P}}(\texttt{e}_0)) = \texttt{T}}{E;\Gamma;\Delta;\texttt{P} \vdash \texttt{e}_0\texttt{.f} : \texttt{T}} \quad \text{(T-FIELD)}$$

$$\frac{\begin{array}{c}E;\Gamma;\Delta;\texttt{P} \vdash \texttt{e}_0 : \texttt{T}_0 \qquad E;\Gamma;\Delta;\texttt{P} \vdash \texttt{e} : \texttt{T} \\ fType(\texttt{f}_i, bound_E(\texttt{T}_0), this_{\texttt{P}}(\texttt{e}_0)) = \texttt{T}_i \qquad E;\Delta \vdash \texttt{T} <: \texttt{T}_i\end{array}}{E;\Gamma;\Delta;\texttt{P} \vdash \texttt{e}_0\texttt{.f} = \texttt{e} : \texttt{T}} \quad \text{(T-ASSIGN)}$$

$$\frac{\begin{array}{c}E;\Gamma;\Delta;\texttt{P} \vdash \texttt{e}_0 : \texttt{T}_0 \\ E;\Delta \vdash \overline{\texttt{V}} : \texttt{ok} \qquad mType(\texttt{m<}\overline{\texttt{V}}\texttt{>}, bound_E(\texttt{T}_0), this_{\texttt{P}}(\texttt{e}_0)) = \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{Q}}\texttt{>}\overline{\texttt{U}} \to \texttt{T}_0 \\ (\forall \texttt{V} \in \overline{\texttt{V}} \Rightarrow E;\Delta \vdash owner_E(\texttt{T}_0) <: owner_E(\texttt{V})) \\ E;\Delta \vdash \overline{\texttt{V}} <: \overline{\texttt{Q}} \qquad E;\Delta;\texttt{P} \vdash \overline{\texttt{e}} : \overline{\texttt{S}} \qquad E;\Delta \vdash \overline{\texttt{S}} <: \overline{\texttt{U}}\end{array}}{E;\Gamma;\Delta;\texttt{P} \vdash \texttt{e}_0 : \texttt{T}_0} \quad \text{(T-INVK)}$$

$$\frac{E;\Gamma;\Delta;\iota \vdash \texttt{e} : \texttt{T}}{E;\Gamma;\Delta;\texttt{P} \vdash \iota > \texttt{e} : \texttt{T}} \quad \text{(T-CONTEXT)}$$

Figure 5.11: FOGJ Typing Rules

5.2.5 FOGJ Expression Typing

The FOGJ type rules can be found in Figure 5.11. The FOGJ typing judgment $E;\Gamma;\Delta;\texttt{P} \vdash \texttt{e} : \texttt{T}$ holds if expression `e` has type `T` in type environment E, environment Γ, store type environment Δ and with permission `P`. The type judgment is encoded in this Section as `typing`. Below is the header for `typing`. `typing E G D P e T` holds if expression `e` has type `T` in type environment `E`, environment `G`, store type environment `D` with permission `P`.

```
Inductive typing : ty_env -> env -> list ty ->
                   Permission -> exp -> ty -> Prop :=
```

Except for the inclusion of a permission, typing for null expressions (T-NULL), variables (T-VAR), locations (T-LOC) and sequences (T-SEQ) are unchanged from FGJ, and so are omitted.

Typing for a `new` expression is greatly simplified in FOGJ from FGJ since all fields are initialized as `null`, and thus takes no parameters. A new expression `new C<`$\overline{\texttt{T}}$`;O>()` has type `C<`$\overline{\texttt{T}}$`;O>` if `C<`$\overline{\texttt{T}}$`;O>` is a well-formed type. This is encoded below as `T_New`.

```
| T_New : forall E G D P C Ts O,
        WF_ty E D (C <<Ts;O>>) CT ->
        typing E G D P (e_new C Ts O) C <<Ts;O>>
```

A field access $\texttt{e}_0\texttt{.f}$ has type `T`, where `T` is the the type of field `f` with $this_{\texttt{P}}(\texttt{e}_0)$ substituted for `This` (T-FIELD). The Coq encoding of T-FIELD is given below, `T_Field`. We also make use of the predicate `defn_ty`, which is not given, but checks that a type does not contain any undefined types (`Ty_Undf`). In other words, `defn_ty Ti` holds if `Ti` does not contain `Ty_Undf` as a sub-expression. This is to ensure that the substitution of $this_{\texttt{P}}(\texttt{e}_0)$ (`this_P P e0`) into the field type does not result in an ill-formed type.

```
| T_Field : forall E G D P e0 T0 N0 fi Ti,
    typing E G D P e0 T0 -> bound E T0 N0 ->
    ftype N0 (this_P P e0) fi Ti -> defn_ty Ti ->
    typing E G D P (e_field e0 fi) Ti
```

A field assignment $\texttt{e}_0\texttt{.f} = \texttt{e}$ has type `T`, where `T` is the type of `e`, and `T` subtypes the type of `f` substituting $this_{\texttt{P}}(\texttt{e}_0)$ for `This` (T-ASSIGN). `T_Assign` below is the Coq code for T-ASSIGN. Apart from the substitution of `this_P P e0` into the field type, and the subsequent defined type check (`defn_ty Ti`), it is unchanged from the assignment typing of FGJ.

```
| T_Assign : forall E G D P e0 T0 N0 fi Ti e T,
    typing E G D P e0 T0 -> bound E T0 N0 ->
```

```
    ftype N0 (this_P P e0) fi Ti -> defn_ty Ti ->
    subtype E D T Ti -> typing E G D P e T ->
    typing E G D P (e_assign e0 fi e) T
```

A method invocation $\texttt{e}_0\texttt{.m<}\overline{\texttt{V}}\texttt{>}(\overline{\texttt{e}})$ has type `T`, where `T` is the return type of `m` with $this_{\texttt{P}}(\texttt{e}_0)$ substituted for `This` (T-INVK). To ensure that methods do not violate ownership, the owner of the receiver must be inside the owners of the method type parameters. The encoding is given below as `T_Invk`, and has little changed from FGJ with the exception of the substitution of `this_P P e0` and the nesting of parameter owners. The nesting of owners is accomplished by a predicate `nested_owners` that is omitted. `nested_owners E D T1 T2` holds if the owner of `T1` is "inside" (subtypes) the owner of `T2` for type environment `E` and store type environment `D`.

```
| T_Invk : forall E G D P e0 T0 N0 es m YP xs T Vs,
    typing E G D P e0 T0 -> bound E T0 N0 ->
    mtype m Vs N0 (this_P P e0) (YP,xs,T) ->defn_ty T->
    (forall x U, In (x,U) xs -> defn_ty U) ->
    (forall Y P, In (Y,P) xs -> defn_ty P) ->
    WF_types E D Vs CT ->
    (forall V, In V Vs -> nested_owners E D T0 V) ->
    subtypes E D Vs (range YP) ->
    subtypings E G D P es (range xs) ->
    typing E G D P (e_meth e0 m Vs es) T
```

A context expression l `> e`, for a location l, has type `T` if `e` has type `T` given permission `P`. This has a straight forward encoding below as `T_Context`.

```
| T_Context : forall E G D P l e T,
               typing E G D (P_loc l) e T ->
               typing E G D P (e_context l e) T
```

$$\frac{visible_E(owner_E(\texttt{T}), \texttt{C})}{visible_E(\texttt{T}, \texttt{C})} \quad \text{(V-TYPE)}$$

$$\frac{\begin{array}{c}\texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}, \texttt{X}_\texttt{O} \triangleleft \texttt{N}_\texttt{O}\texttt{> extends N}\{\overline{\texttt{S}}\ \overline{\texttt{f}};\ \overline{\texttt{M}}\} \\ \texttt{O} \in \{\texttt{X}_\texttt{O}, \texttt{This}, \texttt{World}\} \cup \overline{\texttt{X}}\end{array}}{visible_E(\texttt{O}, \texttt{C})} \quad \text{(V-OWNER)}$$

Figure 5.12: FOGJ Type Visibility

Visibility

The visibility predicate is used to determine if types and expressions are visible within a certain context such as a class or method declaration. Type visibility is given by Figure 5.12. Informally stated, a type `T` is *visible* in a class `C` if the owner of `T` is within the scope of `C`. This captures the difference between referencing an object, and modifying an object. If permissions and the *this* function are used to determine if receivers may be modified within a certain context, visibility determines if objects or types may be referenced within a context. Formally, a type `T` is visible in a class declaration `C` if the owner of `T` is visible in `C` (V-TYPE). An owner type `O` is visible in a class declaration `C` if `O` is one of the defined class parameters, or if it is either `World` or `This`. The encoding of type visibility is given below as `visible`.

```
Inductive visible : ty_env -> ty -> class -> Prop :=
  | V_Type  : forall E T C O,
              owner E T O ->
              visible E O C ->
              visible E T C
  | V_Owner : forall E O C XN ZM N fs ms,
              In (cDecl C XN ZM N fs ms) CT ->
              is_owner O ->
```

```
            (In O (toVars (dom XN)) \/
             In O (Ty_This::(World::nil))) ->
            visible E O C.
```

`visible E T C` holds if type `T` has owner `O` in type environment `E`, and `O` is visible in class `C` (`V_Type`). `visible E O C` holds if owner type `O` has a mapping in the class type environment of `C`, `XN`, or `O` is either the `This` type variable (`Ty_This`) or `World`.

Expression visibility is given in Figure 5.13. An expression is visible if all sub-expressions and types within the expression are visible. As with type visiblility, an expression is judged visible with regards to some class or method declaration. The Coq encoding for expression visibility is omitted as it is a simple encoding of Figure 5.13, and simply checks the visibility of sub-expressions and types. The header is given below.

```
Inductive visible_exp :ty_env->env->class->exp->Prop:=
```

`visible_exp E G C e` holds if expression `e` is visible in class `C`, with type environment `E`, and expression variable environment `G`.

The well-formedness rules for method and class declarations are given in Figure 5.14. The difference from those in FGJ is the addition of ownership nesting in both method and class declarations. For a class or method declaration to access a type or expression, it must be visible. If the primary owner parameter of a class is not ”inside” other owner parameters, the owners-as-modifiers property may be broken by allowing multiple heirarchical paths from `World` to the object. The other extension from FGJ is the requirement of all subexpressions and types to be visible.

The Coq encoding of T-METHOD is given below as `meth_ok`. `meth_ok` is defined as an `Inductive` definition with a single case, `T_Method`. `meth_ok decl C` holds if method declaration `decl` is well-formed for class `C`. All types must be well-formed (`WF_ty`), as well as the body. All typing is done with respect to the type environment `concat E E'`. `E` is the class type parameters, method type parameters and the `This` type

$$\frac{E;\Gamma;\mathtt{C} \vdash \mathit{visible}(\mathtt{e}_0) \qquad E;\Gamma;\emptyset;\mathtt{C} \vdash \mathtt{e}_0.\mathtt{f} : \mathtt{T} \qquad \mathit{visible}_E(\mathtt{T},\mathtt{C})}{E;\Gamma;\mathtt{C} \vdash \mathit{visible}(\mathtt{e}_0.\mathtt{f})} \text{ (V-FIELD)}$$

$$\frac{\begin{array}{c}E;\Gamma;\mathtt{C} \vdash \mathit{visible}(\mathtt{e}_0) \qquad E;\Gamma;\emptyset;\mathtt{C} \vdash \mathtt{e}_0.\mathtt{f}_i : \mathtt{T}_i \\ \mathit{visible}_E(\mathtt{T}_i,\mathtt{C}) \qquad E;\Gamma;\emptyset;\mathtt{C} \vdash \mathtt{e} : \mathtt{T} \qquad \mathit{visible}_E(\mathtt{T},\mathtt{C})\end{array}}{E;\Gamma;\mathtt{C} \vdash \mathit{visible}(\mathtt{e}_0.\mathtt{f}_i = \mathtt{e})} \text{ (V-ASSIGN)}$$

$$\frac{\begin{array}{c}E;\Gamma;\emptyset;\mathtt{C} \vdash \mathtt{e}_0.\mathtt{m}\texttt{<}\overline{\mathtt{V}}\texttt{>}(\overline{\mathtt{e}}) : \mathtt{T} \qquad \mathit{visible}_E(\mathtt{T},\mathtt{C}) \\ \mathit{visible}_E(\overline{\mathtt{V}},\mathtt{C}) \qquad E;\Gamma;\mathtt{C} \vdash \mathit{visible}(\overline{\mathtt{e}}) \qquad E;\Gamma;\mathtt{C} \vdash \mathit{visible}(\mathtt{e}_0)\end{array}}{E;\Gamma;\mathtt{C} \vdash \mathit{visible}(\mathtt{e}_0.\mathtt{m}\texttt{<}\overline{\mathtt{V}}\texttt{>}(\overline{\mathtt{e}}))} \text{ (V-INVK)}$$

$$\frac{E;\Gamma;\emptyset;\mathtt{C} \vdash \mathtt{x} : \mathtt{T} \qquad \mathit{visible}_E(\mathtt{T},\mathtt{C})}{E;\Gamma;\mathtt{C} \vdash \mathit{visible}(\mathtt{x})} \text{ (V-VAR)}$$

$$\frac{\mathit{visible}_E(\mathtt{C}'\texttt{<}\overline{\mathtt{T}};\mathtt{O}\texttt{>},\mathtt{C})}{E;\Gamma;\mathtt{C} \vdash \mathit{visible}(\texttt{new } \mathtt{C}'\texttt{<}\overline{\mathtt{T}};\mathtt{O}\texttt{>}())} \text{ (V-NEW)}$$

$$\frac{E;\Gamma;\mathtt{C} \vdash \mathit{visible}(\mathtt{e}_1) \qquad E;\Gamma;\mathtt{C} \vdash \mathit{visible}(\mathtt{e}_2)}{E;\Gamma;\mathtt{C} \vdash \mathit{visible}(\mathtt{e}_1;\ \mathtt{e}_2)} \text{ (V-SEQ)}$$

Figure 5.13: FOGJ Expression Visibility

variable bound by `Xo`, the primary owner parameter.

```
Inductive meth_ok : MethDecl -> class -> ty -> Prop :=
  | T_Method : forall decl m YP T0 xs e0 C XN Xo No N
                                          fs ms E E' G,
    E = (This,Ty_Var Xo)::(Xo,No)::(concat XN YP) ->
    nested_env Xo (concat XN YP) E' ->
    ok E ->
    decl = mDecl m YP T0 xs e0 ->
    In (cDecl C XN (Xo,No) N fs ms) CT ->
    (forall P, In P (range YP) ->
```

$$\frac{\begin{array}{c}\texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}; \texttt{X}_\texttt{O} \triangleleft \texttt{N}_\texttt{O}\texttt{> extends N}\{\overline{\texttt{S}}\ \overline{\texttt{f}};\ \overline{\texttt{M}}\} \\ E \supset \overline{\texttt{X}} \triangleleft \overline{\texttt{N}}, \overline{\texttt{Y}} \triangleleft \overline{\texttt{Q}} \qquad (\forall \texttt{X} \in \overline{\texttt{X}}, \texttt{X}_\texttt{O} \triangleleft \mathit{owner}_E(\texttt{X}) \in E) \\ (\forall \texttt{Y} \in \overline{\texttt{Y}}, \texttt{X}_\texttt{O} \triangleleft \mathit{owner}_E(\texttt{Y}) \in E) \qquad E; \emptyset \vdash \texttt{T}_0, \overline{\texttt{Q}}, \overline{\texttt{U}} : \texttt{ok} \qquad \mathit{visible}_E(\texttt{T}_0, \texttt{C}) \\ (\forall \texttt{Q} \in \overline{\texttt{Q}}, \mathit{visible}_E(\texttt{Q}, \texttt{C})) \qquad E; \texttt{this} : \texttt{C<}\overline{\texttt{X}}; \texttt{X}_\texttt{O}\texttt{>}, \overline{\texttt{x}} : \overline{\texttt{U}}; \texttt{C} \vdash \mathit{visible}(\texttt{e}_0) \\ E; \texttt{this} : \texttt{C<}\overline{\texttt{X}}; \texttt{X}_\texttt{O}\texttt{>}, \overline{\texttt{x}} : \overline{\texttt{U}}; \emptyset; \texttt{C} \vdash \texttt{e} : \texttt{S}_0 \\ E; \emptyset \vdash \texttt{S}_0 <: \texttt{T}_0 \qquad \mathit{override}(\texttt{m}, \texttt{N}, \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{Q}}\texttt{>}\overline{\texttt{U}} \to \texttt{T}_0)\end{array}}{\texttt{T}_0\ \texttt{m<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{Q}}\texttt{> (}\overline{\texttt{U}}\overline{\texttt{x}}\texttt{)\{return e;\} OK IN C}} \quad \text{(T-METH)}$$

$$\frac{\begin{array}{c}E \supset \overline{\texttt{X}} \triangleleft \overline{\texttt{N}} \qquad \forall \texttt{X} \in \overline{\texttt{X}}, \texttt{X}_\texttt{O} \triangleleft \mathit{owner}_E(\texttt{X}) \in E \\ E; \emptyset \vdash \texttt{N}, \texttt{N}_\texttt{O}, \overline{\texttt{N}}, \overline{\texttt{S}} : \texttt{ok} \qquad \mathit{visible}_E(\texttt{N}_\texttt{O}, \texttt{C}) \\ \forall \texttt{N} \in \overline{\texttt{N}}, \mathit{visible}_E(\texttt{N}, \texttt{C}) \qquad \forall \texttt{S} \in \overline{\texttt{S}}, \mathit{visible}_E(\texttt{S}, \texttt{C}) \qquad \forall \texttt{M} \in \overline{\texttt{M}}, \texttt{M OK IN C}\end{array}}{\texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}; \texttt{X}_\texttt{O} \triangleleft \texttt{N}_\texttt{O}\texttt{> extends N}\{\overline{\texttt{S}}\ \overline{\texttt{f}};\ \overline{\texttt{M}}\}\ \texttt{OK}} \quad \text{(T-CLASS)}$$

Figure 5.14: FOGJ Method and Class Typing Rules

```
                WF_ty (concat E E') nil P CT) ->
  WF_ty (concat E E') nil T0 CT ->
  WF_types (concat E E') nil (range xs) CT ->
  (forall T, In T (range xs) -> visible E T C) ->
  visible E T0 C ->
  (forall P, In P (range YP) -> visible E P C) ->
  G=(this,C<<toTys(toVars(dom XN));Ty_Var Xo>>)::xs->
  visible_exp E G C e0 ->
  subtyping (concat E E') G nil (P_Class C) e0 T0 ->
  override m N YP (range xs) T0 ->
  meth_ok decl C (Ty_Var Xo).
```

An oversight was made when the encoding `meth_ok` by not restricting the parameter types in `xs` to class types. Methods conforming to `meth_ok` may be declared in the following manner.

```
1 Object<World> meth <> (World x){ ... }
```

Such a method would clearly be problematic if it were ever called, however since a well-typed expression in a well-typed environment can never be of an ownership type, `meth` could never be called on a receiver. Any future encodings that make use of FOGJ should take this in account, and restrict parameter types to class types.

The encoding of T-CLASS is given below as `class_ok`, and as with `meth_ok` it is largely unchanged from that of FGJ. `class_ok decl` is gien the notation `CLASS decl OK` throughout the encoding. `CLASS decl OK` holds if `decl` is well-formed. All types within the declaration (field, method and parameter types) must be well-formed. The types must also be visible within the class declaration. The methods must also be well-formed by `meth_ok` above.

```
Inductive class_ok : ClassDecl -> Prop :=
  | T_Class : forall decl C XN Xo No N fs ms E E' E0,
      decl = cDecl C XN (Xo,No) N fs ms ->
      E = (Xo,No)::XN ->
      ok ((This,Ty_Var Xo)::E) ->
      nested_env Xo XN E' ->
      E0 = (concat E E') ->
      WF_ty_env (concat E E') nil CT ->
      (forall N, In N (range E) ->
                 WF_ty E0 nil N CT) ->
      WF_ty E0 nil N CT ->
      (forall f T, In (f,T) fs ->
      WF_ty ((This,Ty_Var Xo)::E0) nil T CT) ->
      (forall M, In M ms -> meth_ok M C (Ty_Var Xo)) ->
      visible E No C ->
      (forall N, In N (range XN) -> visible E N C) ->
      (forall T, In T (range fs) -> visible E T C) ->
      (exists D Ts', N = D <<Ts';Ty_Var Xo>>) ->
      ok_meths ms ->
```

$$\frac{\iota \notin dom(\mathcal{H}) \qquad \mathcal{H}' = \mathcal{H}, \iota \mapsto \texttt{new C<}\overline{\texttt{T}}\texttt{;O>}(\overline{\texttt{null}})}{\texttt{new C<}\overline{\texttt{T}}\texttt{;O>}()|\mathcal{H} \longrightarrow \iota|\mathcal{H}} \text{ (R-NEW)}$$

$$\frac{\mathcal{H}(\iota) = \texttt{new C<}\overline{\texttt{T}}\texttt{;O>}(...) \qquad mBody(\texttt{m<}\overline{\texttt{V}}\texttt{>}, \texttt{C<}\overline{\texttt{T}}\texttt{;O>}, \texttt{This}_\iota) = (\overline{\texttt{x}}; \texttt{e})}{\iota.\texttt{m<}\overline{\texttt{V}}\texttt{>}(\overline{\texttt{v}})|\mathcal{H} \longrightarrow \iota > [\iota/\texttt{this}, \overline{\texttt{v}}/\overline{\texttt{x}}, \texttt{This}_\iota/\texttt{This}]\texttt{e}|\mathcal{H}} \text{ (R-INVK)}$$

$$\frac{}{\iota > \texttt{v}|\mathcal{H} \longrightarrow \texttt{v}|\mathcal{H}} \text{ (R-CONTEXT)}$$

Figure 5.15: FOGJ Reduction Rules

```
(forall Ts O To fs, fields C <<Ts;O>> To fs ->
                    ok fs) ->
class_ok decl.
```

5.2.6 FOGJ Reduction

The FOGJ reduction rules are given in Figure 5.15. Field access, assignment and sequence reduction are unchanged from FGJ, and so are omitted. A new expression reduction adds a new location to the store but unlike FGJ, all fields are initialized as `null` (R-NEW). `R_New` encodes R-NEW below. When encoding `R_New`, we first need to construct the list of `null` expressions to be stored as the field values. `nulls`, an omitted predicate, holds if an input list is a list of `e_null` expressions. The list of fields for `C<<Ts;O>>` is retrieved using `fields`. `Ty_This` (the `This` variable) is passed as substitution for `Ty_This`. It does not matter what type is passed since the types will never be used, but some input is required by `fields`. Once the list of nulls has been constructed, the location is constructed in the same manner as FGJ.

```
| R_New : forall H H' i C fs ns fn Ts O,
       nulls ns -> fields (C <<Ts;O>>) Ty_This fs ->
```

```
       zip fs ns fn ->
       H' = stSnoc H (C <<Ts;O>>, fn) -> leng H = i ->
       (e_new C Ts O) / H --> (e_loc i) / H'
```

Method reduction (R-INVK) is unchanged except that all instances of `This` are substituted for $\texttt{This}_\iota$, where ι is the receiver. `R_Invk` encodes R-NEW below. A method call `e_meth (e_loc i) m Vs vs` reduces to the context expression `e_context i (subst ((this,e_loc i)::R) e)`, where `e` is the body of `m` and `R` is the substitution of the parameters `xs` of `m` for `vs`, the method call parameters. `mbody` is used to retrieve the method body, and the `Ty_This` variable is replaced by `This_ i` in the body, where `i` is the position of the receiver in the store.

```
| R_Invk : forall H i C m xs vs e R es Ts Vs O,
       lookup i H = Some (C <<Ts;O>>,es) ->
       mbody m Vs C <<Ts;O>> (This_ i) (xs,e)->
       values vs -> zip xs vs R ->
       (e_meth (e_loc i) m Vs vs) / H -->
       (e_context i (subst ((this,e_loc i)::R) e)) / H
```

A context expression ι `> v` reduces to `v` (R-CONTEXT). `R_Context` encodes R-CONTEXT below. A context expression `e_context l v` with store `H` reduces to `v` with store `H` if `v` is a value.

```
| R_Context : forall l v H,
       value v -> (e_context l v) / H --> v / H
```

5.3 FOGJ Soundness

Soundness in FOGJ is structured in much the same manner as FGJ. A large portion of the lemmas are devoted to proving type substitution preserves the various judgements. This section will present those lemmas along with the soundness proofs.

5.3.1 Type Substitution

Type Substitution Preserves Bound

The type substitution lemmas of FOGJ are very similar in statement to those of FGJ. There is one addition in Lemma 5.3.1, the preservation of ownership through type substitution. In other words if a type `T` has a owner `O`, then after type substitution, the owner of `T` will be `O`.

Lemma 5.3.1. *If* $E_2 \cup \overline{X} \triangleleft \overline{N} \vdash$ `T ok` *and* $E_1; \Delta \vdash \overline{U} :$ `ok`*, where* $E_1; \Delta \vdash \overline{U} <: [\overline{X}/\overline{U}]N$*, if* $E' = E_1 \cup ([\overline{X}/\overline{N}](E_2 - \overline{X} \triangleleft \overline{U}))$*, then* $owner'_{E'}([\overline{X}/\overline{U}]T) = [\overline{X}/\overline{U}]O$

The encoding of Lemma 5.3.1 is given below.

```
Lemma type_substitution_preserves_ownership :
  forall E T O, owner E T O ->
  forall  E1 XN E2 D Us R, E = concat XN E2 ->
  ok XN -> zip XN Us R ->
  forall E2', subtract XN E2 E2' ->
  forall E', E' = concat E1 (map [:R:] E2') ->
  WF_ty_env E D CT ->
  WF_ty_env E' D CT ->
  WF_ST D ->
  (forall U, In U Us -> WF_ty E1 D U CT) ->
  subtypes E1 D Us (map [R] (range XN)) ->
  (forall Xi Ui, In (Xi,Ui) R ->
              forall Ti, In (Xi,Ti) E2 ->
              subtype E' D Ui ([R] Ti)) ->
  owner E' ([R] T) ([R] O).
```

5.3.2 FOGJ Soundness

Theorem 5.3.1 is the statement of *Preservation* for FOGJ.

Theorem 5.3.1 (Preservation). *If* $e|\mathcal{H} \longrightarrow e'|\mathcal{H}'$ *where* $e' \neq \texttt{err}$ *and* $e; \Gamma; \Delta; P \vdash e : T$*, then* $\exists \Delta'$ *s.t.* $\Delta' \supseteq \Delta$*, and* $\exists S$*, s.t.* $E; \Gamma; \Delta'; P \vdash e' : S$ *and* $E; \Gamma \vdash S <: T$

The encoding of Theorem 5.3.1 is given below.

```
Theorem Preservation :
  (forall p p', reduction p p' ->
    (forall E G D P T e e' H H',
    (e,H) = p -> (e',H') = p' ->
    e' <> e_err ->
    WF_store D H -> WF_env G ->
    WF_ty_env E D CT ->
      typing E G D P e T ->
         (exists D', ST_Extends D' D ->
                     WF_store D' H' ->
                     subtyping E G D' P e' T))) /\
  (forall p p', ListReduction p p' ->
    (forall E G D P Ts es es' H H',
    (es,H) = p -> (es',H') = p' ->
    ~ In e_err es' ->
    WF_store D H -> WF_env G ->
    WF_ty_env E D CT ->
    subtypings E G D P es Ts ->
         (exists D', ST_Extends D' D ->
                     WF_store D' H' ->
                     subtypings E G D' P es' Ts))).
```

Progress is given in Theorem 5.3.2.

Theorem 5.3.2 (Progress). $\forall e$*, if* $E; \Gamma; \Delta; P \vdash e : T$ *then either;*

(a) e*is a value, or*

(b) $\forall \mathcal{H}$ *s.t.* $\mathcal{H}$ *is well-typed with respect to* Δ*,* $\exists e', \mathcal{H}'$ *s.t.* $e; \mathcal{H} \longrightarrow e'; \mathcal{H}'$

The encoding of Theorem 5.3.2 is given below.

```
Theorem Progress :
  (forall E G D P e T,
     typing E G D P e T -> G = nil -> E = nil ->
     (value e \/ (forall H, WF_store D H ->
     exists e', exists H', e / H --> e' / H'))) /\
  (forall E G D P e T,
     subtyping E G D P e T -> G = nil -> E = nil ->
     (value e \/ (forall H, WF_store D H ->
     exists e', exists H', e / H --> e' / H'))) /\
  (forall E G D P es Ts,
     subtypings E G D P es Ts -> G = nil -> E = nil ->
     (values es \/ (forall H, WF_store D H ->
     exists es', exists H',
                 ListReduction (es, H) (es', H')))).
```

Chapter 6

Conclusion

This thesis has presented the encoding of three formalisms, FIJ FGJ and FOGJ. The initial objective of the thesis was to encode Immutability and Ownership using Coq. This Chapter reviews the types systems of Chapters 3, 4 and 5. Section 6.2 discusses the structure of each type system and Section 6.3 discusses potential future work.

6.1 Related Work

Cast-Free Feather Weight Java

The encoding of our original FJ with Assignment type system was based upon a Coq encoding of Cast-Free Featherweight Java developed by De Fraine et al. [5]. As such, the subsequent encodings in this thesis of FIJ, FGJ and FOGJ have all inherited something of the Cast-Free FJ encoding. Predicates such as `subtype`, `typing` are encoded in these type systems are much the same as those of Cast-Free FJ in terms of structure, but there are several changes that have been made. Context reductions in Cast-Free FJ were encoded separately from the reduction predicate, while they were encoded as normal reductions in the FIJ, FGJ and FOGJ encodings. In Cast-Free FJ, types were simply classes, and so the type definition was fairly

simple. In the type systems of this thesis, types are more complex, and this has ramifications in all definitions of an encoding. One notable difference was type well-formedness. In Cast-Free FJ, a type well-formedness predicate did not exist, since all types were classes, and thus well-formed if in the class-table. In the encodings of this thesis (especially FGJ and FOGJ), type well-formedness becomes more complex, and a predicate had to be written.

Colored Fetherweight Java

Colored Featherweight Java [14] [7] (CFJ) is an extension of the Cast-Free Featherweight Java by de Fraine et al. [5]. The CFJ type system is used in evaluating multiple generated variants of a software product-line.

6.2 Encoding a Type System in Coq

While FIJ, FGJ and FOGJ are all different type systems encoding different language properties, there is much they have in common. The all share a very similar structure, and both the definitions of the type systems and the theorems that make up the soundness proofs can be organised into distinct sections that change little between type systems. This is not surprising as each encoding is built upon the same FJ encoding, and each section of an encoding such as FIJ corresponds to a distinct judgement or function in the formal description of the type system.

While there are peculiarities to each type system, such as visibility in FOGJ (see Section 5.2.5), each type system contains the following common sections:

- Syntax: The encoding of the various type system elements; classes, types, expressions, etc.
- Substitution: The substitution of expressions and types.

- Functions: Field and Method retrieval functions.
- Subtyping: The subtyping judgement.
- Well-formedness: Type and environment well-formedness.
- Typing: Expression, class and method Typing.
- Reduction: Expression Reduction.

The sections that have the most impact on the subsequent encoding and soundness proofs are Syntax and Substitution. Other aspects of each type system maintain a fairly standard structure between encodings, while the way types and type substitution is handled changes since each type system has a very different definition for types. Next is a description of the structure of the Syntax and Substitution used in each encoding.

Syntax

The Coq encoded syntax consists of a collection of definitions defining types, expressions and method and class declarations. The most important definitions are types (`ty`) and expressions (`exp`). In all three encodings types and expressions consist of **`Inductive`** definitions of type **`Type`**. The headers for `ty` and `exp` are given below.

```
Inductive ty : Type :=
Inductive exp : Type :=
```

These definitions change as types and expressions are added to each type system. Since types in FIJ, FGJ and FOGJ are very different, the way they are represented changes. In FIJ, types were simply class types parametrised by an immutability parameter. In FGJ, types were extended further to allow a variable number of generic parameters, along with type variables. FOGJ further extended types by adding both variable and non-variable

owner types. FOGJ presented a problem of how to define owner type variables; as owners, or variables. In FOGJ (Chapter 5.2.1), owner type variables are encoded as a distinct kind of type variable using `o_Var`. Thus, for a natural number `x`, we define an ownership type variable `Ty_Var (o_Var x)`. Owner type variables are therefore distinct from normal type variables (defined using `t_Var`), but can still be treated as type variables. If we were to define owner type variables as a type of owner we would redefine the `own` type in Chapter 5.2.1 as the following.

```
Inductive own : Type :=
  | O_World : own
  | O_This_ : nat -> own
  | O_Var : nat -> own.
```

This would mean that owners would be easily distinguishable from other types, but treating owner variables as type variables would be harder. We decided to use the first encoding since it made the encoding of type substitution easier, which is discussed next.

Substitution

In each type system both expression and type substitutions are used. Except for additional expressions, expression substitution is the same in all three type systems, however type substitution changes substantially with each type system. FIJ did not use type substitution, or at least used a very basic type substitution, since substitution only ever involved the substitution of one variable at a time. FGJ and FOGJ did use substitution, and while they involved different cases (corresponding to the different types), they maintained the same structure: a case for substitution applied to a type variable that replaces the type variable with its mapping in the substitution, and a case for each non-variable type that recursively applies the substitution to the type's sub-types. The basic structure is given below.

```
Function subst_ty (E:ty_env)(T:ty){struct T}:ty :=
```

```
 match T with
 | Ty_Var x => match get_Var x E with
                      | None => T
                      | Some N => N
                      end
 | <non-variable type> => <recursive substitution of E>
 ...
end
with subst_tys (E:ty_env)(Ts:tys){struct Ts}:tys:=
  <type list substitution>
end.
```

In order to use this basic structure, we need to be able to group different type variables together, and distinguish them from normal types. As discussed earlier in this Section, we decided to define owner variables along with type variables rather than owners. While this added an extra level of complication when distinguishing owners from normal types, it allowed us to keep the same structure for the type substitution function from FGJ. This meant we could use many of the metatheory theorems from FGJ for substitution and the `get_Var` function. Type substitution contributed substantially to the size of both the FGJ and the FOGJ encodings, therefore enabling code reuse reduced the size of FOGJ relative to FGJ.

Using the structure for types and type substitution discussed in this Section, it is not hard to see how one could extend types further to include other types such as immutability types. If we wanted to extend FGJ with immutability as in IGJ [16], we could follow the structure of FOGJ.

```
Inductive Var : Type :=
 | t_Var : nat -> Var
 | i_Var : nat -> Var.

Inductive imm : Type :=
```

```
  | I_readonly : imm
  | I_mutable : imm
  | I_immutable : imm.

Inductive ty : Type :=
  | Ty_Var   : Var -> ty
  | Ty_Class : class -> tys -> ty -> ty
  | Ty_Imm   : imm -> ty
```

This would allow us to quite easily apply the FOGJ substitution function and lemmas on substitution to some FIGJ type system with minor modification. Given the amount of effort it takes to formally encode a type system, reusing lemmas is premium.

6.3 Future Work

The type systems in this thesis are a good starting point for encoding various types of immutability and ownership, but there are several potential extensions that could be considered.

6.3.1 Readonly References

The addition of readonly references to FIJ would help solve the significant issue with the type system: the split type hierarchy. As discussed in Chapter 3 the use of `mutable` and `immutable` mutability parameters requires a split inheritance hierarchy. A readonly mutability type would allow for both readonly references, and the joining of the split hierarchy. In such a type system, both `mutable` and `immutable` instances of a class would extend a readonly instance. As a result, `readonly` references would require a change of the `subtype` predicate in any future encoding. In the current FIJ type hierarchy shown in Figure 3.1, if $\texttt{C<I}_1\texttt{><:C<I}_2\texttt{>}$, then $\texttt{I}_1 = \texttt{I}_2$, however with the introduction of readonly references we could weaken

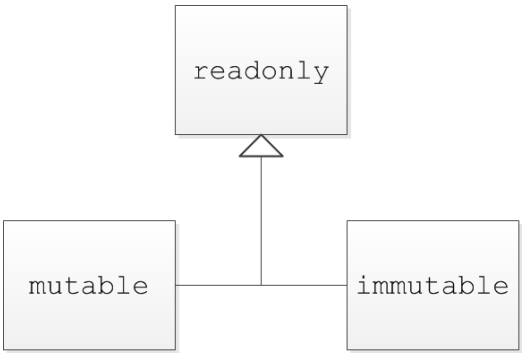

Figure 6.1: Mutability Hierarchy

that restriction to $\texttt{I}_2\texttt{<:I}_1$. A modified mutability hierarchy identical to the mutability in IGJ [16] is shown in Figure 6.3.1. Readonly references would have little effect on type substitution, type well-formedness, lookup functions, expression typing or reduction. Soundness proofs would also be affected since the immutability guarantee would have to take a different form. A likely immutability guarantee would be that any `mutable` reference points to a `mutable` object and any `immutable` reference would point to an immutable object.

Theorem 6.3.1 (Immutability Guarantee)**.** *If* $\Gamma;\Delta \vdash e : T$ *and* $e;\mathcal{H} \longrightarrow^* \iota;\mathcal{H}'$*, then* $immut(T)\texttt{<:}immut(\Delta'(\iota))$*, where* Δ' *is well-formed w.r.t.* $\mathcal{H}'$ *and* $immut(T)$ *returns the immutability of* T*.*

6.3.2 Method Guards

In order to enforce safe method calls, FIJ used the idea of `pure` and `mutating` methods. An `immutable` receiver could only have `pure` methods called on it. OIGJ [17] uses method guards to restrict the types of receiver a method may be called on.

```
void <X extends mutable>? mutableMeth (){}
void <X extends immutable>? immutableMeth (){}
void <X extends readonly>? readonlyMeth (){}
```

The three methods above give an example of method guards. `mutableMeth` may only be called on a mutable receiver, while `immutableMeth` may only be called on an immutable receiver, and may not mutate the receiver. `readonlyMeth` may be called on any receiver, but may not mutate the receiver. If readonly references were to be added, then method guards would provide the safety of pure method annotations while offering more flexibility.

The addition of method guards would obviously alter the way method calls are typed, but there could also be a change in the method retrieval function that might not be expected in a paper formalism of such a type system. Since the mutability of the receiver (`X` in the examples above) would likely be used in the parameters and body of the method, it would have to be substituted during normal type parameter substitution. This would likely mean that it would have to be substituted during method retrieval, and so would be passed as a parameter to the $mType$ and $mBody$ functions.

6.3.3 Featherweight Ownership Immutability Generic Java

Both immutability and ownership are language properties that are important to writing code that is safe from careless aliases and exposure of object representation. It follows that a combination of FIJ and FOGJ would facilitate both objectives [17].

Not only would it be useful from a language perspective, but the development of FGJ (and subsequently FOGJ) has offered solutions to difficulties in FIJ such as the representation and substitution of type parameters. Such an encoding could use the type system of [17] as a basis. Since owner parameters were introduced in FOGJ, much of the foundations of generic immutability already exists within FOGJ. Major changes would take place in subtyping, well-formedness and expression typing. There are no extra predicates needed beyond an $immut$ function returning the primary

immutability parameter for a type.

6.4 Conclusion

In this thesis, we have presented three type systems, FIJ, FGJ and FOGJ. We have encoded them using the Coq Proof Assistant, and have proven them sound. Encoding a type system using a proof assistant gives us an extra level of insurance that our type system is safe, assuming we trust that the proof assistant is correct. While encoding a type system using Coq is both time and effort intensive, we feel that these encodings form a strong basis for future extensions, and provide a basic structure for future encodings. Standardising the structure of a type system encoding is needed in order to allow for code reuse, and code reuse is crucial to reducing the overhead incurred by future encodings.

To our knowledge, the FIJ encoding is the only encoding of a type system featuring Immutability in Coq. FIJ allows for both Class and Object Immutability, and we have also provided a description of how an extension featuring readonly references could be constructed for FIJ that makes use of lessons learned while encoding FGJ and FOGJ.

FGJ extends our previous encoding of the FJ type system with assignment with Java generics. While there are other encodings of FGJ, encoding FGJ was necessary to the encoding of FOGJ and generic type substitution. FOGJ extends FGJ with Ownership, and to our knowledge is the only type system featuring Ownership that is encoded in Coq.

Bibliography

[1] BERTOT, Y., AND CASTÉRAN, P. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. springer, 2004.

[2] CARDELLI, L. Type systems. *ACM Computing Surveys 28*, 1 (1996), 263–264.

[3] CLARKE, D. G., POTTER, J. M., AND NOBLE, J. Ownership types for flexible alias protection. *ACM SIGPLAN Notices 33*, 10 (1998), 48–64.

[4] DIETL, W., DROSSOPOULOU, S., AND MÜLLER, P. Generic universe types. In *ECOOP 2007–Object-Oriented Programming*. Springer, 2007, pp. 28–53.

[5] FRAINE, B. D., ERNST, E., AND SÜDHOLT, M. Cast-free featherweight Java, 2008. `http://soft.vub.ac.be/~bdefrain/featherj/`.

[6] IGARASHI, A., PIERCE, B. C., AND WADLER, P. Featherweight java: a minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst. 23*, 3 (May 2001), 396–450.

[7] KÄSTNER, C., APEL, S., THÜM, T., AND SAAKE, G. Type checking annotation-based product lines. *ACM Trans. Softw. Eng. Methodol. 21*, 3 (July 2012), 14:1–14:39.

[8] MACKAY, J., MEHNERT, H., POTANIN, A., GROVES, L., AND CAMERON, N. Encoding featherweight java with assignment and immutability using the coq proof assistant. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs* (New York, NY, USA, 2012), FTfJP '12, ACM, pp. 11–19.

[9] NOBLE, J., VITEK, J., AND POTTER, J. Flexible alias protection. In *ECOOP'98* (1998), Springer-Verlag, pp. 158–185.

[10] ÖSTLUND, J., WRIGSTAD, T., CLARKE, D., AND ÅKERBLOM, B. Ownership, uniqueness and immutability. In *TOOLS Europe 2008* (2008).

[11] PIERCE, B., CASINGHINO, C., GABOARDI, M., GREENBERG, M., HRICU, C., SJBERG, V., AND YORGEY, B. Software foundations, 2013. `http://www.cis.upenn.edu/~bcpierce/sf/`.

[12] PIERCE, B. C. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[13] THE COQ DEVELOPMENT TEAM. The coq proof assistant, 2013. `http://coq.inria.fr/`.

[14] THÜM, T. A machine-checked proof for a product-line–aware type system. *Master's thesis, University of Magdeburg* (2010).

[15] TSCHANTZ, M., AND ERNST, M. Javari: adding reference immutability to Java. In *OOPSLA2005* (2005).

[16] ZIBIN, Y., POTANIN, A., ALI, M., ARTZI, S., KIE, UN, A., AND ERNST, M. D. Object and reference immutability using Java generics. In *ESEC/FSE2007* (New York, NY, USA, 2007), ACM, pp. 75–84.

[17] ZIBIN, Y., POTANIN, A., LI, P., ALI, M., AND ERNST, M. D. Ownership and immutability in generic Java. In *OOPSLA* (2010), pp. 598–617.