Encoding Featherweight Java with Assignment and Immutability using The Coq Proof Assistant

Julian Mackay Hannes Mehnert Alex Potanin
Lindsay Groves Nicholas Cameron

ABSTRACT

We develop a mechanized proof of Featherweight Java with Assignment and Immutability in the Coq proof assistant. This is a step towards more machine-checked proofs of a non-trivial type system. We used object immutability close to that of IGJ [8] . We describe the challenges of the mechanization and the encoding we used inside of Coq.

1. INTRODUCTION

Object immutability is a useful language property that allows for better reasoning and sharing of objects. Our immutability definition follows that of IGJ [8]: one can declare transitively immutable objects by utilising the immutability parameter, but one is still allowed to expicitly declare immutability of fields inside immutable objects. For simplicity we do not model readonly references present in IGJ [7].

An immutable object cannot be modified, and only pure (side-effect free) methods can be called on such an object. A mutable object can be modified and mutating methods can be called on that object. Our immutability is parametric and as a result we support transitively immutable objects. All fields of an immutable object are assigned in the constructor. The intersection of mutable and immutable objects is empty. Methods annotated with the `pure` keyword are only allowed to read fields and call other `pure` methods.

We provide a simple type system, based on Featherweight Java [3] with assignment and object immutability. The main contribution of this paper is a formalization of the type system and a soundness proof in the Coq Proof Assistant. Our Coq development extends the Cast-Free Featherweight Java formalization [2], which does not handle assignment. We introduced a store typing context to support assignment, which to our knowledge has not been mechanized. We highlight the encodings we used, and explain the differences between a proof on paper and formalizing it inside a theorem prover. Our Coq sources are publically available[1].

Example. We define a parametrized class `Cell`, where the mutable instantiation can get and set the interned object, whereas the immutable instantiation can only get the interned object, provided initially in the constructor. We chose to use transitive mutability in this example.

```
class Cell<M> extends Object<M> {
  Object<M> data;
  Cell<M> (Object<M> data) {
    this.data = data;
  }
  Object<M> getData pure () {
    return this.data;
  }
  void setData mutating (Object<M> data) {
    this.data = data;
  }
}
...
new Cell<immutable>(new Object<immutable>()).setData(...);
```

The call to a `Cell` object in the line above that calls a mutating method on an immutable object is not allowed by our language (encoded by FJ + AI type system described next).

2. FJ + AI TYPE SYSTEM

The syntax is given in Figure 1. An expression may be `null`, an address location, a variable, an error, a field access, a field assignment, a method call, a `new` expression or a sequence of two expressions. A value can be either `null` or a location. A type is a class name (N) parameterized by a mutability parameter. A class declaration is a class name, a class mutability parameter, a super class name and its class mutability parameter, a list of fields a constructor K and a list of methods. A constructor of a class calls the

[1] http://www.ecs.vuw.ac.nz/˜mackayjuli/FJAI.zip

$\texttt{e} ::= \texttt{null} \mid \iota \mid \texttt{x} \mid \texttt{err} \mid \iota.\texttt{f} \mid \iota.\texttt{f = e} \mid \texttt{e.m}(\overline{\texttt{e}}) \mid \texttt{new}\,\texttt{T}(\overline{\texttt{e}}) \mid \texttt{e; e}$
$\texttt{v} ::= \texttt{null} \mid \iota$
$\texttt{T} ::= \texttt{N <I>}$
$\texttt{C} ::= \texttt{class N <I> extends N <I>}\{\overline{\texttt{T}}\,\overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\}$
$\texttt{K} ::= \texttt{T}(\overline{\texttt{T}}\,\overline{\texttt{f}})\{\texttt{super}(\overline{\texttt{f}});\ \texttt{this}.\overline{\texttt{f}}\texttt{=}\overline{\texttt{f}};\}$
$\texttt{M} ::= \texttt{T m P}(\overline{\texttt{T}}\,\overline{\texttt{x}})\{\texttt{return e;}\}$
$\texttt{I} ::= \texttt{mutable} \mid \texttt{immutable} \mid \texttt{X}$
$\texttt{P} ::= \texttt{mutating} \mid \texttt{pure}$
$\texttt{Pr} ::= \overline{\texttt{C}};\ \texttt{e}$

Figure 1: FJ+AI Syntax

constructor of the super class, and then assigns values to the fields declared in the class. A method declaration has a return type, a method name, a `pure` or `mutating` annotation, a list of parameters and a body. Mutability (`I`) can be either a definite mutability parameter (`mutable` or `immutable`), or a mutability variable (`X`). Finally, a program `Pr` consists of a list of class declarations and an expression to be evaluated.

2.1 Subtyping and Functions

Subtyping is defined by three subtyping rules. S-Refl, S-Trans and S-Extend. The reflexivity and transitivity rules are standard [3] and we omit them. The interesting rule shown below is for inheritance (S-Extends) that restricts the subtyping to types with the same mutability parameter. This splits the inheritance tree into one that is `mutable`, and another that is `immutable`. `Object <immutable>` roots the `immutable` tree, `Object <mutable>` roots the `mutable` one.

$$\frac{}{\texttt{C<M> <: C<M>}}\ \text{(S-Refl)} \qquad \frac{\texttt{C<M> extends D<M>}}{\texttt{C<M> <: D<M>}}\ \text{(S-Extend)}$$

$$\frac{\texttt{C<M> <: D<M>} \qquad \texttt{D<M> <: E<M>}}{\texttt{C<M> <: E<M>}}\ \text{(S-Trans)}$$

The functions used are the standard functions for field and method lookup as found in FJ [3].

$$\frac{\texttt{class C extends D}\{\overline{\texttt{T}}\,\overline{\texttt{f}};\overline{\texttt{M}}\} \qquad \texttt{T f} \in \overline{\texttt{T}}\,\overline{\texttt{f}}}{fType(\texttt{f},\texttt{C}) = \texttt{T}}$$

$$\frac{\texttt{class C extends D}\{\overline{\texttt{T}}\,\overline{\texttt{f}};\overline{\texttt{M}}\} \qquad \texttt{T f} \notin \overline{\texttt{T}}\,\overline{\texttt{f}}}{fType(\texttt{f},\texttt{C}) = fType(\texttt{f},\texttt{D})}$$

$$\frac{}{fields(\texttt{Object}) = \emptyset} \qquad \frac{\texttt{class C extends D}\{\overline{\texttt{T}}\,\overline{\texttt{f}};\texttt{K}\,\overline{\texttt{M}}\}}{fields(\texttt{C}) = \overline{\texttt{T}}\,\overline{\texttt{f}} \cup fields(\texttt{D})}$$

$$\frac{\texttt{class C extends D}\{\overline{\texttt{T}_\texttt{f}}\,\overline{\texttt{f}};\overline{\texttt{M}}\} \qquad \texttt{T}_0\,\texttt{m}(\overline{\texttt{T}}\,\overline{\texttt{x}})\{\texttt{return e;}\} \in \overline{\texttt{M}}}{mType(\texttt{m},\texttt{C}) = \texttt{T}_0 \to \overline{\texttt{T}}}$$

$$\frac{\texttt{class C extends D}\{\overline{\texttt{T}_\texttt{f}}\,\overline{\texttt{f}};\overline{\texttt{M}}\} \qquad \forall\,\texttt{T}_0,\,\texttt{P},\,\overline{\texttt{T}},\,\overline{\texttt{x}},\,\texttt{e}\ :\notin \overline{\texttt{M}}}{mType(\texttt{m},\texttt{C}) = mType(\texttt{m},\texttt{D})}$$

$$\frac{\texttt{class C extends D}\{\overline{\texttt{T}_\texttt{f}}\,\overline{\texttt{f}};\overline{\texttt{M}}\} \qquad \texttt{T}_0\,\texttt{m}(\overline{\texttt{T}}\,\overline{\texttt{x}})\{\texttt{return e;}\} \in \overline{\texttt{M}}}{mBody(\texttt{m},\texttt{C}) = (\overline{\texttt{x}};\texttt{e})}$$

$$\frac{\texttt{class C extends D}\{\overline{\texttt{T}_\texttt{f}}\,\overline{\texttt{f}};\overline{\texttt{M}}\} \quad \forall\,\texttt{T}_0,\,\texttt{P},\,\overline{\texttt{T}},\,\overline{\texttt{x}},\,\texttt{e}\ :\ \texttt{T}_0\,\texttt{m P }(\overline{\texttt{T}}\,\overline{\texttt{x}})\{\texttt{return e;}\} \notin \overline{\texttt{M}}}{mBody(\texttt{m},\texttt{C}) = mBody(\texttt{m},\texttt{D})}$$

Figure 2: FJ+AI Field and Method Lookup Functions

2.2 Typing

An expression can be well-typed according to seven typing rules as shown in Figure 3. There are an additional two typing rules for method typing (T-Meth-Pure and T-Meth-Mut), as well as one for class typing (T-Class). An expression has type `T` in the context of two partial functions, an environment Γ and a store typing Δ. Γ maps variables to types and Δ maps locations to types.

$$\Gamma;\Delta \vdash \texttt{x} : \Gamma(\texttt{x})\ \text{(T-Var)} \qquad \Gamma;\Delta \vdash \iota : \Delta(\iota)\ \text{(T-Loc)}$$

$$\frac{\Gamma;\Delta \vdash \texttt{e} : \texttt{C <M>} \qquad fType(\texttt{f},\texttt{C <M>}) = \texttt{T}}{\Gamma;\Delta \vdash \texttt{e.f} : \texttt{T}}\ \text{(T-Field)}$$

$$\frac{fields(\texttt{C}) = \overline{\texttt{T}}\,\overline{\texttt{f}} \qquad \Gamma;\Delta \vdash \overline{\texttt{e}} : \overline{\texttt{T}'} \qquad \overline{\texttt{T}'} \texttt{<:} \overline{\texttt{T}}}{\Gamma;\Delta \vdash \texttt{new C<I>}(\overline{\texttt{e}}) : \texttt{C<I>}}\ \text{(T-New)}$$

$$\frac{\Gamma;\Delta \vdash \texttt{e} : \texttt{C <mutable>} \qquad \Gamma;\Delta \vdash \texttt{e}' : \texttt{T}' \qquad fType(\texttt{f},\texttt{C <mutable>}) = \texttt{T} \qquad \texttt{T}'\texttt{<:T}}{\Gamma;\Delta \vdash \texttt{e.f} = \texttt{e}' : \texttt{T}}\ \text{(T-Assign)}$$

$$\frac{\Gamma;\Delta \vdash \texttt{e} : \texttt{C <mutable>} \qquad mType(\texttt{m},\texttt{C <mutable>}) = \overline{\texttt{D}} \to \texttt{T} \qquad \Gamma;\Delta \vdash \overline{\texttt{e}} : \overline{\texttt{E}} \qquad \overline{\texttt{E}}\texttt{<:}\overline{\texttt{D}}}{\Gamma;\Delta \vdash \texttt{e.m}(\overline{\texttt{e}}) : \texttt{T}}\ \text{(T-Invk-Mut)}$$

$$\frac{\Gamma;\Delta \vdash \texttt{e} : \texttt{C <immutable>} \qquad mType(\texttt{m},\texttt{C <immutable>}) = \overline{\texttt{D}} \to \texttt{T} \qquad \Gamma;\Delta \vdash \overline{\texttt{e}} : \overline{\texttt{E}} \qquad \overline{\texttt{E}}\texttt{<:}\overline{\texttt{D}} \qquad \texttt{m}\,pure}{\Gamma;\Delta \vdash \texttt{e.m}(\overline{\texttt{e}}) : \texttt{T}}\ \text{(T-Invk-Imm)}$$

$$\frac{\exists\,\Delta\ s.t.\ \texttt{this} : \texttt{C<immutable>}, \overline{\texttt{C}}\,\overline{\texttt{x}};\Delta \vdash \texttt{e} : \texttt{T} \qquad \texttt{class C extends D} \qquad if\ mType(\texttt{m},\texttt{D<I>}) = \overline{\texttt{D}} \to \texttt{T}'\ then\ \overline{\texttt{C}} = \overline{\texttt{D}}\ and\ \texttt{T<:T}'}{\texttt{T m pure}(\overline{\texttt{C}}\,\overline{\texttt{x}})\{\texttt{return e;}\}\texttt{OK IN C <I>}}\ \text{(T-Meth-Pure)}$$

$$\frac{\exists\Delta\ s.t.\ \texttt{this} : \texttt{C<mutable>}, \overline{\texttt{C}}\,\overline{\texttt{x}};\Delta \vdash \texttt{e} : \texttt{T} \qquad \texttt{I} = \texttt{mutable} \vee \texttt{I} = \texttt{X} \qquad \texttt{class C extends D} \qquad if\ mType(\texttt{m},\texttt{D<I>}) = \overline{\texttt{D}} \to \texttt{T}'\ then\ \overline{\texttt{C}} = \overline{\texttt{D}}\ and\ \texttt{T<:T}'}{\texttt{T m mutating}(\overline{\texttt{C}}\,\overline{\texttt{x}})\{\texttt{return e;}\}\texttt{OK IN C <I>}}\ \text{(T-Meth-Mut)}$$

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

Figure 3: FJ+AI Typing Rules

A variable has type `T` in the context of an environment Γ if there is a mapping to `T` in Γ (T-Var). A location ι in a store $\mathcal{H}$ has type `T` in a store typing Δ for $\mathcal{H}$ if Δ maps ι to `T`(T-Loc). A field access `e.f` has type `T` if `e` has type `C<I>` and `f` has type `T` in class `C` (T-Field). An assignment `e.f = v` has type `T` if `e` has type `C<mutable>`, `f` has type `T` in class `C` and `v` has type `T` (T-Assign). A `new` expression `new C<I>`($\overline{\texttt{e}}$) has type `C<I>` if $\overline{\texttt{e}}$ are well-typed with respect to the types of the fields of class `C` (T-New).

There are two typing rules for method invocations, one for method calls on mutable objects and another for immutable objects. T-Invk-Mut allows typing on method calls to mutable receivers. A method call `e.m`($\overline{\texttt{e}}$) has type `T` if $\overline{\texttt{e}}$ has type `C<mutable>`, `m` has type $\overline{\texttt{D}} \to \texttt{T}$ in class `C` and $\overline{\texttt{e}}$ are well-typed with respect to $\overline{\texttt{D}}$. T-Invk-Imm is similar except the method `m` must be pure, since the receiver has type `C<immutable>`.

A method `m` is well-typed for a type `C<I>` in one of two cases. Firstly, if the method is `pure` (T-Meth-Pure) then the body of the method must be well-typed with an `immutable` receiver and with respect to an empty store typing. If the method overrides a method of the super class then the return type must be a subtype of the overridden method, and the parameter types must be the same. Secondly, a method `m` is well-typed for a type `C<I>` if `m` is `mutating` (T-Meth-Mut) and the body is well-typed for a mutable receiver in

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

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

$$\frac{\iota \ \mathit{fresh} \qquad \mathcal{H}' = \mathcal{H}, \iota \mapsto \texttt{new C<M>}(\overline{\texttt{v}})}{\texttt{new C<M>}(\overline{\texttt{v}}); \mathcal{H} \longrightarrow \iota; \mathcal{H}'} \quad \text{(R-New)}$$

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

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

Figure 4: FJ+AI Reduction Rules

$$\frac{\texttt{e}; \mathcal{H} \longrightarrow \texttt{e'}; \mathcal{H}' \qquad \texttt{e}' \neq \texttt{err}}{\texttt{e.f}; \mathcal{H} \longrightarrow \texttt{e}'\texttt{.f}; \mathcal{H}'} \quad \text{(RC-Field)}$$

$$\frac{\texttt{e}; \mathcal{H} \longrightarrow \texttt{err}; \mathcal{H}'}{\texttt{e.f}; \mathcal{H} \longrightarrow \texttt{err}; \mathcal{H}'} \quad \text{(RC-Field-Err)}$$

$$\frac{}{\texttt{null.f}; \mathcal{H} \longrightarrow \texttt{err}; \mathcal{H}} \quad \text{(R-Field-Null)}$$

$$\frac{\texttt{e}; \mathcal{H} \longrightarrow \texttt{e}'; \mathcal{H}' \qquad \texttt{e}' \neq \texttt{err}}{\texttt{e.m}(\overline{\texttt{e}}); \mathcal{H} \longrightarrow \texttt{e}'\texttt{.m}(\overline{\texttt{e}}); \mathcal{H}'} \quad \text{(RC-Invk-Recv)}$$

$$\frac{\texttt{e}_\texttt{i}; \mathcal{H} \longrightarrow \texttt{e}'_\texttt{i}; \mathcal{H}' \qquad \texttt{e}'_\texttt{i} \neq \texttt{err}}{\texttt{e.m}(\overline{\texttt{e}}); \mathcal{H} \longrightarrow \texttt{e.m}(..., \texttt{e}_\texttt{i-1}, \texttt{e}'_\texttt{i}, \texttt{e}_\texttt{i+1}, ...); \mathcal{H}'} \quad \text{(RC-Invk-Arg)}$$

$$\frac{\texttt{e}_\texttt{i}; \mathcal{H} \longrightarrow \texttt{err}; \mathcal{H}'}{\texttt{e.m}(\overline{\texttt{e}}); \mathcal{H} \longrightarrow \texttt{err}; \mathcal{H}'} \quad \text{(RC-Invk-Err)}$$

$$\frac{}{\texttt{null.m}(\overline{\texttt{e}}); \mathcal{H} \longrightarrow \texttt{err}; \mathcal{H}} \quad \text{(R-Invk-Null)}$$

$$\frac{\texttt{e}_\texttt{i}; \mathcal{H} \longrightarrow \texttt{e}'_\texttt{i}; \mathcal{H}' \qquad \texttt{e}'_\texttt{i} \neq \texttt{err}}{\texttt{new C<M>}(\overline{\texttt{e}}); \mathcal{H} \longrightarrow \texttt{new C<M>}(..., \texttt{e}_\texttt{i-1}, \texttt{e}'_\texttt{i}, \texttt{e}_\texttt{i+1}, ...); \mathcal{H}'} \quad \text{(RC-New)}$$

$$\frac{\texttt{e}_\texttt{i}; \mathcal{H} \longrightarrow \texttt{err}; \mathcal{H}'}{\texttt{new C<M>}(\overline{\texttt{e}}); \mathcal{H} \longrightarrow \texttt{err}; \mathcal{H}'} \quad \text{(RC-New-Err)}$$

$$\frac{\texttt{e}; \mathcal{H} \longrightarrow \texttt{e}'; \mathcal{H}' \qquad \texttt{e}' \neq \texttt{err}}{\texttt{e0.f} = \texttt{e}; \mathcal{H} \longrightarrow \texttt{e0.f} = \texttt{e}'; \mathcal{H}'} \quad \text{(RC-Assign-RHS)}$$

$$\frac{\texttt{e}; \mathcal{H} \longrightarrow \texttt{e}'; \mathcal{H}' \qquad \texttt{e}' \neq \texttt{err}}{\texttt{e.f} = \texttt{e0}; \mathcal{H} \longrightarrow \texttt{e}'\texttt{.f} = \texttt{e0}; \mathcal{H}'} \quad \text{(RC-Assign-LHS)}$$

$$\frac{\texttt{e}; \mathcal{H} \longrightarrow \texttt{err}; \mathcal{H}'}{\texttt{e0.f} = \texttt{e}; \mathcal{H} \longrightarrow \texttt{err}; \mathcal{H}'} \quad \text{(RC-Assign-Err)}$$

$$\frac{\texttt{e}; \mathcal{H} \longrightarrow \texttt{err}; \mathcal{H}'}{\texttt{e.f} = \texttt{e0}; \mathcal{H} \longrightarrow \texttt{err}; \mathcal{H}'} \quad \text{(R-Assign-Err)}$$

$$\frac{}{\texttt{null.f = e}; \mathcal{H} \longrightarrow \texttt{err}; \mathcal{H}} \quad \text{(R-Assign-Null)}$$

$$\frac{\texttt{e1}; \mathcal{H} \longrightarrow \texttt{e1}'; \mathcal{H}' \qquad \texttt{e1}' \neq \texttt{err}}{\texttt{e1; e2}; \mathcal{H} \longrightarrow \texttt{e1}'\texttt{; e2}; \mathcal{H}'} \quad \text{(RC-Seq)}$$

$$\frac{\texttt{e1}; \mathcal{H} \longrightarrow \texttt{err}; \mathcal{H}'}{\texttt{e1; e2}; \mathcal{H} \longrightarrow \texttt{err}; \mathcal{H}'} \quad \text{(RC-Seq-Err)}$$

Figure 5: FJ+AI Context Reduction Rules

the context of an empty store typing. The mutability of the type `C<I>` must be either `mutable` or a mutability variable. Again if the method overrides a method of the super class then the return type must subtype the return type of the overridden method, and the types of the parameters must be the same.

A class declaration is well-typed (T-Class) if the fields declared in the class declaration do not duplicate any fields of the super class and all declared methods are well-typed.

2.3 Reduction

Expression reduction is described by a series of reduction rules shown in Figure 4. The contrext reduxtion can be found in Figure 5 .The reduction rules R-Field, R-New, R-Assign and R-Invk show reduction for field accesses, allocation, field assignment and method invocation respectively. A field access returns the value of a field of a location in the store. `new` expression adds a new location to the store and returns the address of that location. A field assignment changes the store by replacing a value of a field of an existing location in the store and returns that value. A method invocation reduces to the body of the method substituted by the method parameters and the receiver for the `this` variable.

3. FJ + AI COQ ENCODING

The Coq encoding has two parts. Firstly a set of definitions corresponding to the syntax, reduction rules and typing rules was created, followed by proofs for type soundness of the encoded language.

3.1 Definitions

The definitions of the language can be broken down into three different types of definitions. Firstly the syntactic elements of the language such as the various expressions, classes, as well as structures such as stores, class tables and environments, form the basis of the encoding. These basic type definitions are followed by several important predicates and functions. The final part of the definiitons deal directly with the typing and reduction rules of sections 2.2 and 2.3.

3.1.1 Syntax Definitions

Classes in the encoding are defined inductively much like natural numbers are defined in Coq. A class can be created in one of two ways, as `Object` (analogous to 0 in the Coq definition of natural numbers), or by extending an existing class, `C extends D` (analogous to Successor). To avoid ambiguity between two direct subclasses of a class `D`, this constructed class has to be identified by a unique `ClassName`. The subclass relationship (described later) becomes much like the "less than" relationship of natural numbers.

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

Mutability and pure/mutating annotation for methods are defined as follows: [2]

```
Inductive mutability : Type :=
```

[2]In order to provide the method annotation `pure`, there had to be some alternate annotation `mutating`, as a default no annotation method could not be created. This is somewhat cumbersome, and is obviously not what would be used if not for the restrictions of Coq.

```
  | mutable   : mutability
  | immutable : mutability
  | variable  : mutability.
```

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

Types are encoded as a class followed by an immutability parameter. The notation `C <<M>>` is used throughout the encoding to indicate a type of class `C` followed by a mutability parameter `M`.

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

Expressions were encoded in the same way previous encodings did, using an `Inductive` type, and is given below.

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

In the definition above, the possible expressions are defined, and are taken directly from the syntax shown in Figure 1. `var` is an identifier for variables. `meth` is a method name. `field` is a field name. A sequence, given by `e_seq e1 e2`, is given the notation`e1 ;; e2` as a shortening throughout the encoding.

For ease, the following shorthand is defined. A list of arguments (`args`) is defined as a list of variable, type pairs. Similarly, a list of field, type pairs is defined as `flds`.

```
Notation args := (list (var * ty)).
Notation flds := (list (field * ty)).
```

A method declaration (`mDecl m T0 p As e`) is defined as a method name (`m`), a return type (`T0`), a method mutability parameter (`p`), a list of arguments (`As`) and a method body (an expression `e`). Much like `args` and `flds` given above, `mths` is given as a list of method declarations.

```
Inductive MethDecl : Type :=
  | mDecl : meth -> ty ->
            meth_mut -> args ->
            exp -> MethDecl.

Notation mths := (list MethDecl).
```

A class declaration (`ClassDecl`) is defined as a class with a mutability parameter, followed by a list of fields and a list of method declarations. A list of class declarations is called a class table, and is required for any program that makes use of classes more complex than Object. Throughout the encoding, a specific class table is made reference to, i.e. `CT`

```
Inductive ClassDecl : Type :=
  | cDecl : class ->
            mutability ->
            flds -> mths ->
            ClassDecl.

Notation ClassTable := (list ClassDecl).
Parameter CT : ClassTable.
```

Stores are represented as a list of pairs. Each pair consists of a type and a list of values corresponding to the fields of the type. A location `e_loc n` points to the nth element of the store. Store typings which are functions from the locations of a store to types are also represented by a list, this time of types. The nth element in a store maps to the nth element of the relevant store typing.

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

3.1.2 Subtyping

The encoded definition of the subclass relation is given below. A class `C0` is a subclass of another class `C1` in three cases, reflexivity (`S_Refl`), transitivity (`S_Trans`) and extension (`S_Extends`).

```
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, C = C0 extends D ->
                In (cDecl C mutC fs ms) CT -> subclass C D.
```

Subtyping in the encoding is given as a predicate on types. A type `T0 = C0<<mut0>>` is considered a subtype of another type `T1 = C1<<mut1>>` if `C0` is a subclass of `C1`, and `mut0 = mut1`.

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

3.1.3 Functions

Predicate `method C decl` is used to determine whether a method call with method declaration `decl` is a valid method call on an expression of class `C`. A method call on an expression of class `C` is valid in one of two cases. The first (`m_this`) is when the method declaration `decl` is defined in the list of methods (`ms`) for class `C` in the class table `CT`, and the second is when a method `m` is inherited from the superclass `D`. A method inherited from a class' super class requires that there be no method of the same name in the list of declared methods for that class.

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

The fields predicate corresponds to the function of the same name in the FJ type system [3]. `fields C fs` holds for a class C and a list of fields fs if fs are the available fields for class C. This would hold in one of two cases, firstly if C = Object, and fs = nil (the empty list), or if C ≠ Object. In the second case, the fields fs are the fields defined in the class declaration of C in the class table CT, appended with the fields of D (where C extends D).

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

3.1.4 Substitution

When a method is called, the arguments are substituted into the body of the method. A recursive function is defined to recursively substitute the arguments in the body of the method, and any sub-expressions in the body. The function subst takes an expression (the body of the method) and a relation mapping variables to expressions as inputs. For all expressions except variables, subst simply recursively applies itself to all subexpressions. In the case of variables, if the variable is in the domain of the relation E, then it is replaced by its image in E, else nothing happens. `get x E` returns the mapping of x 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_new C mut es => e_new C mut
       (List.map (subst E) es)
  | e_meth e0 m es => e_meth
       (subst E e0) m
       (List.map (subst E) es)
  | e_field e1 f => e_field
       (subst E e1) f
  | e_loc n => e
  | e_assign i f e0 => e_assign
       (subst E i) f (subst E e0)
  | e_null => e_null
  | e_err => e_err
  | e_seq e1 e2 => e_seq
       (subst E e1) (subst E e2)
end.
```

Instances of objects with variable mutability need to have their instance mutability substituted into various parts of the class declaration. The following functions handle the substitution of mutability into mutability parameters (subst_mut), types (subst_ty), pairs (subst_pair : field / type and variable / type pairs) and expressions (subst_mut_exp : new expressions in method bodies that use the mutability of the class). All these functions take two inputs: a mutability parameter to be substituted and an object to be substituted into. These functions can be found below.

```
Definition subst_mut (mut0 : mutability)
                     (M : mutability)
                     : mutability :=
  match M with
  | mutable => M
  | immutable => M
  | variable => mut0
end.

Definition subst_ty (mut0 : mutability)
                    (T : ty) : ty :=
  match T with
  C <<mutC>> => C <<subst_mut mut0 mutC>>
end.

Definition subst_pair {A : Type}
                      (mut0 : mutability)
                      (X : A * ty) : A * ty :=
  match X with
  | (a, T) => (a, (subst_ty mut0 T))
end.

Fixpoint subst_mut_exp (mut0 : mutability)
                       (e : exp) : exp :=
  match e with
  | e_new C variable es => e_new C mut0
      (List.map (subst_mut_exp mut0) es)
  | e_new C mutable es => e_new C mutable
      (List.map (subst_mut_exp mut0) es)
  | e_new C immutable es => e_new C immutable
      (List.map (subst_mut_exp mut0) es)
  | e_meth e0 m es => e_meth
      (subst_mut_exp mut0 e0) m
      (List.map (subst_mut_exp mut0) es)
  | e_field e0 f => e_field
      (subst_mut_exp mut0 e0) f
  | e_assign e0 f e1 => e_assign
      (subst_mut_exp mut0 e0) f
      (subst_mut_exp mut0 e1)
  | e_seq e1 e2 => e_seq
      (subst_mut_exp mut0 e1)
      (subst_mut_exp mut0 e2)
  | e_loc n => e
  | e_var n => e
  | e_null => e
  | e_err => e
end.
```

3.1.5 Expression Typing

The typing predicate encodes the type rules of Section 2.2. The typing predicate takes an environment (Gamma), a store typing (Delta), an expression (e) and a type (T) as inputs. `typing Gamma Delta e T` corresponds to $\texttt{Gamma}, \texttt{Delta} \vdash \texttt{e} : \texttt{T}$, or e has type T given environment Gamma and store typing Delta. The following tye rules are all part of the same predicate, and have the following header:

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

T_Var encodes typing for variables. The variable is represented by `e_var x`. Gamma is required to be OK (env_ok). This combined with requiring that (x,T) be in Gamma ensures that T is the correct type in the environment. The type T is also required to be OK (ok_type).

```
  | 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
```

T_Loc encodes typing for locations. The location is given by `e_loc i`. The ith position in the store typing is retrieved, and the returned type is the type of the location. As a requirement, `i < stLength Delta`, i.e. i is in the store typing. Again the type T must be OK.

```
| 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
```

`T_Field` encodes the type rule for field accesses. `validField C0 fi Ti` is a function that simply requires that if `fields C0 fs` holds for some `fs`, then `(fi, Ti)` must be in `fs`. Since `Ti` may use the mutability variable of `C0`, then we have to substitute `mutC0` into `Ti`(`T = subst_ty mutC0 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
```

`T_Assign` encodes typing of field assignments. Again the field `fi` must be "valid", as with `T_Field`. The receiver must be annotated as `mutable`, and the assigned expression must have a type `T` that is a sub type of `Ti` with mutable substituted into it, where `Ti` is the type of the field.

```
| 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
```

`T_Invk` encodes the type rule for method calls. To be well-typed, the method must be a valid method (`method`) for the class of the receiver (`C0`). If the receiver is not `mutable`, then the method must be annotated as pure (`mut0 <> mutable -> mutM = pure`). The method call arguments must be have types that are subtypes of the types of the method parameters with the mutability of the receiver substituted into them. The method call type must then have the mutability of the receiver substituted into it (`T = (subst_ty mut0 T0)`).

```
| 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
```

`T_New` is the encoding of `new` expressions. The mutability of the initialized object (`mutC`) must be defined (`mutability_defined mutC`). The fields of `C` have to be fetched, and the parameters must have types that subtype their types. `C <<mutC>>` must be an OK type (`ok_type C <<mutC>> CT`).

```
| 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>>
```

`T_Seq` encodes the type rule for sequences. The encoding is simple, and merely requires that each expression in the sequence be well-typed.

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

Another two predicates used along with `typing` are `subtyping` and `subtypings`. `subtyping` is a combination of the typing and subtype predicates. The subtypings prediate simply maps to the `subtyping` predicate for a list of expressions and a list 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 ->
             subtyping Gamma Delta e T ->
             subtypings Gamma Delta (e::es) (T::Ts).
```

3.1.6 Method and Class Typing

Method typing is encoded in the definition below. Instead of spliting the method typing up, method typing was encoded as asingle predicate. Different cases were captured using implications. The most important aspect of the method typing was the typing of the body. The body of all methods must be well-typed for a mutable receiver, and an immutable receiver if it is annotated as pure. `(mutC = immutable -> mutM = pure)` requires all methods for immutable types to be pure. Since `meth_ok` is only used during class typing in the encoding, this means all methods in a class with an immutability parameter of `immutable` must be pure. If the mutability parameter is defined (`mutability_defined mutC`), then the method body must be well typed without any mutability substitution, or in other words, the body may not make use of any mutability parameters.

The second part of the method typing is ensuring that if the method overrides a method from the super class, then the method conforms to the typing requirements, i.e. the parameter types must be the same, and the return type must subtype the return type of the overridden method. The pure annotation must also be the same.

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

```
            (subst_mut_exp immutable e0)
            (subst_ty immutable T0)) /\
    (mutability_defined mutC ->
    subtyping ((this, C <<mutC>>)::As) nil e0 T0) /\
    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')).

Notation "decl 'OK_IN' C" := (meth_ok decl C) (at level 0).
```

Class typing is a straightforward encoding of the T-Class rule from section 2.2. A class declaration is well-formed if all the methods and all the fields are well formed. Methods are well-formed according to `meth_ok`, and fields are well-formed if the field types are well-formed, and there are no duplicate fields.

```
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).
```

Class tables must also be well-formed, and this is captured by the inductive predicate `CT_ok`, that holds for a given class table [3].

For a class table to be well formed, one of two cases must hold. Firstly, all empty (`nil`) class tables are well formed. Secondly, for a non-empty class table, each class in the class table must extend some other class (`C = C0 extends D`), or in other words a well-formed class table may not contain a declaration for `Object`. The class must be unique in the class table, or there may not be any duplicates in the class table. Finally, the mutability parameter must conform to the mutability parameter of the super class, i.e. the type `D <<mutx>>` must be an OK type (`ok_type D <<mutx>> CTbl`).

```
Inductive CT_ok : ClassTable -> Prop :=
  | ok_nil  : CT_ok nil
  | ok_head : forall C C0 D fs ms CTbl mutx,
     CLASS (cDecl C mutx fs ms) OK ->
     C = C0 extends D ->
     (forall mut0 mut1,
       ~subtype D <<mut0>> C <<mut1>>) ->
     (forall mutx' fs' ms',
       ~In (cDecl C mutx' fs' ms') CTbl) ->
     ok_type (D <<mutx>>) CTbl ->
     CT_ok CTbl ->
     CT_ok ((cDecl C mutx fs ms)::CTbl).
```

[3] Currently there is a single `Admitted` proof in the encoding: `CT_OK` (that a particular class table `CT` is OK). `CT` is the class table used in all instances where a specific class table is required, and is the one that is in the soundness proofs. Admitting this proof merely acts as a global assumption, instead of assuming it in every proof, or using a generic class table as a parameter in every function. While this is not ideal, it simplified the functions and proofs.

3.1.7 Reduction

The reduction predicate takes two expression, store pairs `(e, H)` and `(e', H')` as inputs. The predicate holds if `(e, H)` reduces to `(e', H')`. Reduction has the following header, and all reuctions that follow are part of the same predicate.

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

`R_Field` is the encoding for field reductions. For a location `e_loc i`, the contents of the location are accessed with `store_lookup i H`. The fields of the class is then accessed using `fields C fs`. In order to ensure that the correct field is retrieved, the fields must be OK (`ok_fields fs`), i.e. there must be no duplicates. The fields (`fs`) and field values `vs` are then zipped, and the value corresponding to `f` is returned.

```
  | 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_Invk` encodes the reduction rule for method call. As in `R_Field`, the contents of `e_loc i` are retrieved. The parameters of the method call must be values (`values vs`), and the method must be valid for the class of the receiver (`method`). The parameters and their corresponding types are zipped together to create a substitution relation `R`, and substituted into the body.

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

`R_New` encodes the reduction of new expressions. Since the store is a list, indexing the positions of the store gives the first element at position 0 and the last position as $i - 1$, where `i` is trhe length of the store (`stLength H`). Thus, to append a new location to the store, will be done at position `i`. The new location is then appended to the end of the store, and `e_loc i` is returned.

```
  | 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_Assign encodes the reduction of field assignment. As in R_Field, the contents of the location e_loc i are retreived, along with the fields of the object, and the two are zipped together. The zipping is done in this case to ensure that the fields and the values are the same length. The index of the field f is identified by lookup_index n fs = Some (f, T), and the nth position of vs is then replaced by the assigned variable v (vs' = replace n v vs). The new object with the new set of values then replaces the contents of the location (H' = replace i (C <<mutC>>,vs') H), and the v is returned.

```
| 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_Seq encodes the reduction for sequences. The reduction is straightforward, only requiring that the first expression in the sequence be a value (value v), and then returning the second expression. written.

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

3.2 Soundness Proofs

3.2.1 Substitution Preserves Typing

Before type preservation is proven in section 3.2.2, type preservation during substitution must be demonstrated. For a given substitution relation R (constructed using an environment xBs and a list of expressions ds), if an expression e has type T, then e substituted with R must have the same type. This is proven using by mutual induction on the typing, subtyping and subtypings predicates. The proof is fairly straight forward.

THEOREM 1. *If* $\overline{x} : \overline{B}, \Delta, \vdash e : T$*,* $\Gamma, \Delta, \vdash \overline{d} : \overline{A}$*, where* $\overline{A} <: \overline{B}$ *then* $\Gamma, \Delta, \vdash [\overline{d}/\overline{x}]e : T' \Rightarrow T' <: T$*.*

```
Theorem Substitution_Preserves_Typing :
  forall E D ds R xBs,
  subtypings E D ds (range xBs) ->
  SubstRelZip xBs ds R ->
  (forall Gamma Delta e T,
    typing Gamma Delta e T ->
    Gamma = xBs -> Delta = D ->
    subtyping E Delta (subst R e) T) /\
  (forall Gamma Delta e T,
    subtyping Gamma Delta e T ->
    Gamma = xBs -> Delta = D ->
    subtyping E Delta (subst R e) T) /\
  (forall Gamma Delta es Ts,
    subtypings Gamma Delta es Ts ->
    Gamma = xBs -> Delta = D ->
    subtypings E Delta (List.map (subst R) es) Ts).
```

3.2.2 Preservation

Proof of type preservation is fairly standard in all cases except when dealing with method call reduction. During reduction of a method call there are two substitutions into the method body. First, parameters are substituted into the body and then the mutability of the receiver. Type preservation for substitution of parameters is proven in *Substitution Preserves Typing.* In order for a method invocation to be well typed, a method annotated as pure cannot mutate the receiver, and all method calls to immutable objects must be pure. This must also hold for subexpressions of the method body. Since some objects may be initialized as either mutable or immutable, when the mutativity is substituted into the method body, the body must still be well typed. Typing of method bodies is handled by meth_ok. This is proved in the theorem *Method Impies Typing*, given below.

THEOREM 2. *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* $T' <: T$*.*

```
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))).
```

Below is the statement for *Method Impies Typing.* Informally, the theorem states that given a type C <<mutC>> that is well-formed (ok_type C <<mutC>> CT) and that the mutability mutC is defined, the method m is a valid method for class C, and the implication that if mutC is not equal to mutable, then the method is annotated as pure, then there exists a type T that is a super type of C <<mutC>>, and the body of m substituted with mutC is well-typed with respect to some subtype of the return type of the method T0 substituted with mutC.

```
Lemma method_implies_typing :
           forall C mutC m e0 T0 mutM E,
  ok_type C <<mutC>> CT ->
  mutability_defined mutC ->
  (mutC <> mutable -> mutM = pure) ->
  method C (mDecl m T0 mutM E e0) ->
  (exists T, subtype (C <<mutC>>) T /\
  subtyping
      ((this, T) :: (List.map (subst_pair mutC) E)) nil
      (subst_mut_exp mutC e0) (subst_ty mutC T0)).
```

3.2.3 Progress

In the same way that the statements for *Preservation* are split into cases for both single expressions and lists, *Progress* is solved using a mutual inductive scheme on typing for expressions, subtyping and subtyping on lits. The proof of *Progress* is straight forward.

THEOREM 3. *If* $\Gamma, \Delta, \vdash e : T$*, 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.* $e; \mathcal{H} \longrightarrow e'; \mathcal{H}'$

```
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')))).
```

3.2.4 Immutability Guarantee

The guarantee that immutable objects do not mutate after they have been construted is given below. Informally the theorem states that the fields of immutable objects in locations in a store will not change if for any reduction of any well-typed expression. Since the field values are either locations or `null` values, requiring that the fields of immutable objects do not change does not require that any possible fields of those fields will not change, i.e. transitivity is not required. The immutablilty guarantee is straight forward for all cases except assignment and `new` expressions, since these are the only expressions that change the store. `new` expressions do not modify any existing locations, and so is resolved easily. In the case of assignment, the fact that the expression is well typed requires that the receiver be mutable, resulting in a contradiction.

THEOREM 4. *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 <immutable>,* $\mathcal{H}(\iota) = (T, \overline{v})$ *and* $\mathcal{H}'(\iota) = (T', \overline{v'})$ *then* $\overline{v} = \overline{v'}$

```
Theorem Immutability_Guarantee :
  (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'))).
```

4. RELATED WORK

We relate our work to two different research areas - one being immutability, whereas the other is mechanization of object-oriented type systems.

Immutability. Developers using off-the-shelf Java can annotate variables with the `final` keyword. A field declared `final` can only be assigned to once in the constructor, a variable declared `final` can also only be assigned once. Methods cannot be annotated at all. The `const` keyword in C++ can also be used to annotate method's arguments and receivers. However, neither annotation supports transitivity, only the annotated variable is protected, not the object it refers to. Our object immutability supports transitivity for both the immutability parameter and method immutability annotations.

Different variants of immutability have been the subject of research recently [7, 8, 5, 9]. We follow mainly the *immutable object* [8] discipline, but in our formalization the subtyping tree of mutable and immutable classes do not share a common root. Another variant are *read-only references* [7]. In this discipline each reference is either read-only or normal. A read-only reference is an immutable handle of the object, via which no modification can be done. The same object may be mutable if accessed via a normal reference, which may coexist.

Mechanization. To our knowledge, there are no mechanized formalizations of type systems with immutability, they are proven on paper solely. The basic formalization of Featherweight Java [2], on which we based our development, does not support assignment and immutability. Kim and Fu [4] extended the FJ formalization for a core Fortress type system. Their focus is multiple dispatch and multiple inheritance, they do not consider assignment. Strniška et al [6] provide a formalization in Ott of Featherweight Java. Based on that Delaware et al [1] formalized FJ with composition features, and formalized a constraint-based typing.

5. CONCLUSION AND FUTURE WORK

We presented a mechanized formalization of transitive object immutability for Featherweight Java with assignment. The Coq development consists of roughly 3000 lines of proof script. This is a moderate size. The trusted code base of the proof is Coq itself, which is a widely used proof assistant based on the Calculus of Inductive Construction (CiC).

In the future we plan to modularly extend our formalization with other immutability variants and incorporate an ownership discipline. We chose object immutability as opposed to class immutability or readonly references since it is closer to our next encoding target of ownership, due to the parameterization of class declarations.

6. REFERENCES

[1] Benjamin Delaware, William R. Cook, and Don Batory. Fitting the pieces together: a machine-checked model of safe composition. In *ESEC/FSE2009*, pages 243–252, New York, NY, USA, 2009. ACM.

[2] Bruno De Fraine, Erik Ernst, and Mario Südholt. Cast-free featherweight Java, 2008. `http://soft.vub.ac.be/~bdefrain/featherj/`.

[3] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001.

[4] Jieung Kim and Sukyoung Ryu. Coq mechanization of featherweight Fortress with multiple dispatch and multiple inheritance. In *CPP*, 2011.

[5] Johan Östlund, Tobias Wrigstad, Dave Clarke, and Beatrice Åkerblom. Ownership, uniqueness and immutability. In *TOOLS Europe 2008*, 2008.

[6] Rok Strniša, Peter Sewell, and Matthew Parkinson. The Java module system: core design and semantic definition. In *OOPSLA2007*, pages 499–514, New York, NY, USA, 2007. ACM.

[7] Matthew Tschantz and Michael Ernst. Javari: adding reference immutability to Java. In *OOPSLA2005*, 2005.

[8] Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kie, un, and Michael D. Ernst. Object and reference immutability using Java generics. In *ESEC/FSE2007*, pages 75–84, New York, NY, USA, 2007. ACM.

[9] Yoav Zibin, Alex Potanin, Paley Li, Mahmood Ali, and Michael D. Ernst. Ownership and immutability in generic Java. In *OOPSLA*, pages 598–617, 2010.