# Assignment Featherweight Java: Bringing Mutable State to Featherweight Java

Thomas Mølhave* Lars H. Petersen†

**Abstract**

*In this paper we present an extension of a very simple calculus for Java, Featherweight Java. The new dialect introduces mutable field variables, and brings the calculus closer to the way programs are usually written in Java. We call this new language AFJ, Assignment Featherweight Java. Besides the formal specification of AFJ, we make a soundness proof which is relatively simple due to the simplicity of AFJ itself. In addition to the theoretical work we present a Standard ML implementation of a type-checker for AFJ.*

## 1 Introduction

The simple Featherweight Java(FJ) calculus, as described in [Igarashi et al., 2001] is based on a subset of the syntax of Java. While not semantically a proper subset of Java, it aims to provide the same feeling. The semantics of FJ are functional and the evaluation order is nondeterministic. However due to the functional style and syntactic equivalence, FJ programs compile and run readily as ordinary Java programs. This paper adds the capability to assign new values into field variables. To do this we have to be more specific about the evaluation order to ensure that side-effects happen in a predictable order. The new dialect is called *Assignment Featherweight Java*, or simply AFJ. A strict design goal of AFJ was to keep the ability to run AFJ programs on a JVM compiled with an ordinary Java compiler. In addition we wanted to reuse much of the constructs of FJ. In order to specify the evaluation order, however, we had to do the semantics differently, and we have no semantic constructs from FJ in AFJ. We do however reuse all the typing rules, and introduce two new. In addition to the theoretical definitions and proofs of soundness we have implemented a working type-inferrer to gain experience with the practical typing aspects of AFJ. A digital version of this paper and the source code for the type-checker can be downloaded at `http://www.daimi.au.dk/~thomasm/TOOL/`. This paper is an exam project for the 2005 version of the course *Types in Object Oriented Languages* held by Mads Torgersen at the Univeristy of Aarhus.

*`thomasm@daimi.au.dk`, University of Aarhus
†`aveng@daimi.au.dk`, University of Aarhus

## 2 Previous Work

Helped by the wide-spread use of Java in commercial applications, including internet banking and other security-centric areas, the type-safety, or lack thereof, of Java has seen much research. Since Java is a huge language researches construct smaller models and subsets of Java analysing subjects of interest in these. Such models are plenty, and includes Pizza [Odersky and Wadler, 1997], Middle-weight Java [Bierman et al., 2003], Featherweight Java and Featherweight Generic Java [Igarashi et al., 2001]. Some models consider large subsets of Java making the construction of proofs very tedious and technical. Others, like FJ are small models with gives rise to small and relatively elegant proofs. Type soundness is proved in many of the models ([Drossopoulou et al., 1999], [Igarashi et al., 2001]). The entire Java system in all its generality, however, is not type-safe according to [Saraswat, 1997], which gives various examples of how to cheat the system using class-loaders and modifying internal JVM data structures. However, in local systems where one can exhibit some kind of control over the JVM and the programs run using it, type soundness is still relevant. In this aspect a model like [Igarashi et al., 2001] is, perhaps, too lightweight since most people, rightly so, find large-scale functional programming in Java too restrictive and/or cumbersome. This paper aims to make FJ a bit more in-sync with typical Java use, while still retaining a simple calculus in which soundness proofs are feasible.

## 3 Syntax

We need to expand the syntax of FJ in order to be able to assign into field variables. Our first idea was to introduce *statements* to method bodies, this results

```
L ::= class C extends C{ C̄ f̄ K M̄}
K ::= C(C̄ f̄){super(f̄); this.f̄=f̄;}
M ::= C m(C̄ x̄){ s̄ return e;}
e ::= x | e.f | e.m(ē) | new C(ē) | (C)e
s ::= x.f = e;
```

Listing 1: First try at a syntax for AFJ.

in the syntax found in Listing 1. In order to do this the we needed to add change the rule T-Method from [Igarashi et al., 2001] in order to type-check the methods, and since a statement has no type we would need to make up some sort of void type, or type-check statements similar to how methods and classes are verified.

A more elegant way of supporting assignment is to use the fact that an assignment in Java also is an expression. The following piece of code is legal Java code and, as will be shown shortly, also legal AFJ code (assuming it is embedded in a legal *Pair* class).

```
Pair f(Pair p) {
  return (this.fst = p).fst = (fst=p);
}
```

So instead of adding statements we just add a new *update* expression, the value of such an expression, $e_0.x = e_1$, is the value $e_1$. The new expression is the only one added to the syntax of FJ to form the concrete syntax of AFJ. However for technical reasons, which will be clear shortly, we also add yet another expression $(p, C)$, where $p \in \mathbb{N}$ and $C$ is the name of a class. Such an expression models a pointer to a place, $p$, in memory containing an instance of an object of type $C$. From now on the metavariable $l$ is a location and $\bar{l}$ is a comma-separated list of locations $\bar{l} = l_1, l_2, \ldots, l_{\#(\bar{l})}$, the empty list is denoted "$\bullet$". The final abstract syntax of AFJ is shown in Listing 2. Note that this is *abstract* syntax in the sense that the expression $(p, C)$ is not allowed in concrete programs. We note that the

```
L ::= class C extends C{ C̄ f̄; K M̄}
K ::= C(C̄ f̄){super(f̄); this.f̄=f̄;}
M ::= C m(C̄ x̄){ return e;}
e ::= x | e.f | e.m(ē) | new C(ē) |
      (C)e | (e.f = e) | (p,C)
```

Listing 2: Final AFJ Syntax

final syntax does not limit our expressibility in relation to the first syntax. One can easily map a list of statements to a list of assignment expressions using an auxiliary function. The following simple function with statetements

```
C f(C1 x1, ..., Cn xn) {
  s̄
  return e;
}
```

can simply be translated into

```
C g(C1 x1, ..., Cn xn, Object d1, ..., Object d#(s̄)) {
  return e;
}
C f(C1 x1, ..., Cn xn) {
  return g(x1, ..., xn, s1, ..., s#(s̄));
}
```

The first example is written in the syntax from Listing 1, the second example uses the AFJ syntax from Listing 2, the operations will be executed in the same order and give the same result.

A *class table* is kept at all times during the computation of an AFJ program, it is a mapping from class names $C$ to class declarations $L$. We now define an AFJ program to be the tuple, $(CT, e)$, of a class table and an expression. As in FJ, we enforce some sanity conditions on the class hierarchy, namely that all classes appearing in the class declaration in $CT$ are in the domain of the table as well. Additionally the are no cycles (or self-cycles) in the subtype relation induced by $CT$, making the "$<:$" relation antisymmetric.

# 4 Type Inference

With the syntax specified we are now in a position to talk about the types of various AFJ components. The type inference rules for updates and locations are given here:

$$\frac{\Gamma \vdash e_0.f_i : C \qquad \Gamma \vdash e_1 : D \qquad D <: C}{\Gamma \vdash (e_0.f_i = e_1) : C} \quad \text{T-Update}$$

$$\Gamma \vdash (p, C) : C \quad \text{T-Loc}$$

In addition to these two rules we reuse all the rules from [Igarashi et al., 2001] verbatim. For completeness we have written all these typing rules in this paper as well, they can be found in Table 1 on the following page. We also use the sub-typing relations and the auxiliary functions from FJ, these can be found in Table 2 on page 4. We note that the type-checking is as simple as in FJ, and that we maintain the ability to do modular type-checking.

To verify that our rule T-Update was correct, we wrote a small test program, shown in Listing 3 on the following page, and passed it on to the Java compiler. It showed us that the type of an assignment was indeed the type of the receiving field.

$$\Gamma \vdash x : \Gamma(x) \qquad \text{T-Var}$$

$$\frac{\Gamma \vdash e_0 : C_0 \qquad \text{mtype}(m, C_0) = \bar{D} \to C \qquad \Gamma \vdash \bar{e} : \bar{C} \qquad \bar{C} <: \bar{D}}{\Gamma \vdash e_0.m(\bar{e}) : C} \qquad \text{T-Invk}$$

$$\frac{\Gamma \vdash e_0 : D \qquad D <: C}{\Gamma \vdash (C)e_0 : C} \qquad \text{T-UCast}$$

$$\frac{\Gamma \vdash e_0 : D \qquad D \not<: C \qquad C \not<: D \qquad \textit{stupid warning}}{\Gamma \vdash (C)e_0 : C} \qquad \text{T-SCast}$$

$$\frac{\Gamma \vdash e_0 : C_0 \qquad \text{fields}(C_0) = \bar{C}\,\bar{f}}{\Gamma \vdash e_0.f_i : C_i} \qquad \text{T-Field}$$

$$\frac{\text{fields}(C) = \bar{D}\ \bar{f} \qquad \Gamma \vdash \bar{e} : \bar{C} \qquad \bar{C} <: \bar{D}}{\Gamma \vdash \text{new } C(\bar{e}) : C} \qquad \text{T-New}$$

$$\frac{\Gamma \vdash e_0 : D \qquad C <: D \qquad C \neq D}{\Gamma \vdash (C)e_0 : C} \qquad \text{T-DCast}$$

$$\frac{\bar{x} : \bar{C}, \mathit{this} : C \vdash e_0 : E_0 \qquad E_0 <: C_0 \qquad \text{class C extends D } \{\dots\} \qquad \text{if mtype}(m, D) = \bar{D} \to D_0, \text{ then } C = D \text{ and } C_0 = D_0}{C_0\ m(\bar{C}\ \bar{x})\{\text{ return } e_0;\ \} \text{ OK IN } C} \qquad \text{T-Method}$$

$$\frac{K = C(\bar{D}\ \bar{g},\ \bar{C}\ \bar{f})\{\text{super}(\bar{g});\ \mathit{this}.\bar{f} = \bar{f};\} \qquad \text{fields}(D) = \bar{D}\ \bar{g} \qquad \bar{M} \text{ OK IN } C}{\text{class } C \text{ extends } D\ \{\bar{C}\ \bar{f};\ K\ \bar{M}\}\text{OK}} \qquad \text{T-Class}$$

Table 1: The complete set of FJ typing rules. These, along with T-Update and T-Loc, make up the typing rules of AFJ.

```
class C {
  public Object bar() {
    return new Object();
  }
}
class D extends C {
  public Object foo() {
    return new Object();
  }
}
class Test {
  public C f;
  public Object run() {
    /* not possible to call foo() below */
    return (this.f=new D()).bar();
  }
}
```

Listing 3: Java Code used to verify T-Update.

# 5 Semantics

When we specify the semantics of AFJ we need to model the flow of information caused by the computation, this flow cannot be modelled in the transition systems used by [Igarashi et al., 2001] since we allow updates to fields. Instead we use the generalised labelled transition systems as explained in [Mosses, 2003].

## 5.1 The Transition System

We first define ordinary labelled transition systems. The basic idea of a labelled transition system is to allow labels on transitions, allowing different transitions between the same states to be distinguished.

**Definition 5.1.** Let *State* be a set of states, and let *Label* be a set of labels for transitions. The transition relation for a *labelled transition system* is a ternary relation: $\to \subseteq (\mathit{State} \times \mathit{Label} \times \mathit{State})$. An existence of a transition, labelled $L$, from $S$ to $S$ is indicated by writing $S - L \to S'$.

In order to better control the relationship between two relations and their labels we define the generalised transition system the same way as in [Mosses, 2003]:

**Definition 5.2.** A *generalised labelled transition system* is a labelled transition system equipped with a bi-

$$C <: C \qquad \frac{C <: D \qquad D <: E}{C <: E} \qquad \frac{\text{class C extends D } \{\ldots\}}{C <: D}$$

$$\text{fields}(Object) = \bullet$$

$$\frac{\text{class } C \text{ extends } D \ \{\ \bar{C}\ \bar{f};\ K\ \bar{M}\}}{\text{mtype}(m, C) = \bar{B} \to B} \qquad B\ m(\bar{B}\ \bar{x})\{\ \text{return } e;\ \} \in \bar{M}$$

$$\frac{\text{class } C \text{ extends } D \ \{\ \bar{C}\ \bar{f};\ K\ \bar{M}\}}{\text{mtype}(m, C) = \text{mtype}(m, D)} \qquad m \notin \bar{M}$$

$$\frac{\text{class } C \text{ extends } D \ \{\ \bar{C}\ \bar{f};\ K\ \bar{M}\}}{\text{mbody}(m, C) = \bar{x}.e} \qquad B\ m(\bar{B}\ \bar{x})\{\ \text{return } e;\ \} \in \bar{M}$$

$$\frac{\text{class } C \text{ extends } D \ \{\ \bar{C}\ \bar{f};\ K\ \bar{M}\}}{\text{mbody}(m, C) = \text{mbody}(m, D)} \qquad m \notin \bar{M}$$

Table 2: Sub-typing relations and auxiliary functions

nary label *Composable* on *Label* and a set $Unobs \subseteq Label$. The only difference from an ordinary LTS is that when considering sequences of transitions, the labels on each pair of adjacent transitions are required to be in the composable relation. Labels $U$ in *Unobs* are used for unobservable transitions, and they do not affect composability: when $L_1$ is composable with $U$ and $U$ is composable with $L_2$ then $L_1$ is composable with $L_2$. Moreover, for each $L$ there are $U_1$ and $U_2$ such that $U_1$ is composable with $L$ and $L$ is composable with $U_2$.

In order to use a generalised labelled transition system (GLTS) we need to define our labels, specify the unobservable labels and specify the relation of composable labels. We only have one kind of information in our labels, the *store*. The store $ST : (\mathbb{N}, C) \mapsto (\mathbb{N}, C)^*$ is a mapping from a location to a list of locations. It is used to lookup the list of instances pointed to by the field variables of a given object instance. For a given class name $C$, and mapping $ST$ we define $ST + (p, C) \mapsto \bar{l}$ as the function gained by picking a not already used integer $p$ and extending $ST$ with the not-previously bound mapping $(p, C) \mapsto \bar{l}$. Similarly we define $ST/(p, C) \mapsto \bar{l}$ to be the function gained by changing the mapping of the location stored at the integer $p$ to $\bar{l}$.

An AFJ label, $L$, is then defined to contain two components, $s$ and $s'$, both $s$ and $s'$ are stores. Conceptually $s$ is the store before a transition takes place and $s'$ will denote the store after such a transition. With our labels defined, we can now complete the definition of AFJ's GLTS. We define composability of labels as:

**Definition 5.3 (Composability of AFJ labels).** The Labels $L_1 = \{s = ST_1, s' = ST_1'\}$ and $L_2 = \{s = ST_2, s' = ST_2'\}$ are composable precisely when $ST_1' = ST_2$.

Two stores are equal if they have the same domain and map equal arguments to the equal values. We define unobservable labels as:

**Definition 5.4 (Unobservable AFJ labels).** A label $L = \{s = ST, s' = ST'\}$ is unobservable if $ST = ST'$.

A transition $e - L \to e'$ can be written $e \to e'$ if $L$ is unobservable.

We now only need to create the reduction rules to complete the specification of AFJ. The reduction rules are split into to categories, *computations* and *congruence rules*. All computations works almost exclusively on expressions which are locations, while the congruence rules are responsible for transforming all expressions into locations. These rules, which can be seen in Table 3 on the next page, are quite different from the ones in [Igarashi et al., 2001] and remove the nondeterministic evaluation order and replace them with the actual evaluation order of Java.

| | | | |
|---|---|---|---|
| $\dfrac{ST' = ((p,C) \mapsto \bar{l}) + ST}{new\ C(\bar{l}) - \{s = ST, s' = ST'\} \to (p,C)}$ | R-New | $\dfrac{e \to e'}{(C)e \to (C)e'}$ | RC-Cast |
| $\dfrac{\text{mbody}(m,C) = \bar{x}.e}{(p,C).m(\bar{l}) \to [\bar{l}/\bar{x}, (p,C)/this]e}$ | R-Invk | $\dfrac{e_0 \to e_0'}{e_0.m(\bar{e}) \to e_0'.m(\bar{e})}$ | RC-Invk-Recv |
| $\dfrac{ST(l) = \bar{l}}{l.f_i - \{s = ST, s' = ST\} \to l_i}$ | R-Field | $\dfrac{e_0 \to e_0'}{l.m(\bar{l}, e_0, \bar{e}) \to l.m(\bar{l}, e_0', \bar{e})}$ | RC-Invk-Arg |
| $\dfrac{C <: D}{(D)(p,C) \to (p,C)}$ | R-Cast | $\dfrac{e_0 \to e_0'}{\text{new } C(\bar{l}, e_0, \bar{e}) \to new\ C(\bar{l}, e_0', \bar{e})}$ | RC-New |
| $\dfrac{ST(l) = \bar{l} \qquad \bar{l}' = l_1, \ldots, l_{i-1}, \hat{l}, l_{i+1}, \ldots, l_{\#(\bar{l})}}{(l.f_i = \hat{l}) - \{s = ST, s' = (l \mapsto \bar{l}')/ST\} \to \hat{l}}$ | R-Update | $\dfrac{e \to e'}{e.f \to e'.f}$ | RC-Field |
| $\dfrac{e \to e'}{(l.f_i = e) \to (l.f_i = e')}$ | RC-Update-Arg | $\dfrac{e_0 \to e_0'}{(e_0.f_i = e) \to (e_0'.f_i = e)}$ | RC-Update-Rcv |

Table 3: Computations and Congruence rules for AFJ.

# 6 Soundness

We will now prove that well-typed AFJ programs only get stuck at invalid casts. The proofs follow the style from [Igarashi et al., 2001] closely, extending and revising when necessary. We start out with a series of lemmas. In all the lemmas we assume that the class table and the main expression in the formulations, have been successfully checked by the typing rules.

**Lemma 6.1.** *If* $mtype(m,D) = \bar{C} \to C_0$, *then* $mtype(m,C) = \bar{C} \to C_0$ *for all* $C <: D$

*Proof.* This is trivially proved by induction of the derivation of $C <: D$ and $\text{mtype}(m,D)$. The proof from [Igarashi et al., 2001] can be used virtually unchanged, since our two new typing rules do not deal with methods.. □

**Lemma 6.2 (Term Substitution Preserves Typing).** *If* $\Gamma, \bar{x} : \bar{B} \vdash e : D$, *and* $\Gamma \vdash \bar{d} : \bar{A}$ *where* $\bar{A} <: \bar{B}$, *then* $\Gamma \vdash [\bar{d}/\bar{x}]e : C$ *for some* $C <: D$.

*Proof.* This is proved by induction over the typing rule used. We have preserved all typing rules from [Igarashi et al., 2001] and refer the proofs of these cases to that document. We prove the lemma only for our new type rules.

**Case T-Update** When the type rule used was T-Update, $e$ was of the form $e_0.f_i = e_1$. By T-Update, we have that $\Gamma, \bar{x} : \bar{B} \vdash e_0.f_i = e_1 : D$, $\Gamma \vdash \bar{d} : \bar{A}$ and $\bar{A} <: \bar{B}$. Then, by the rule T-Update we know $\Gamma, \bar{x} : \bar{B} \vdash e_0.f_i : E_0$, $\Gamma, \bar{x} : \bar{B} \vdash e_1 : E_1$ and $E_1 <: E_0$. By the induction hypothesis, $\Gamma \vdash [\bar{d}/\bar{x}]e_0.f_i : E_0'$, $\Gamma \vdash [\bar{d}/\bar{x}]e_1 : E_1'$ where $E_0' <: E_0$ and $E_1' <: E_1$. By looking into the case for T-Field in [Igarashi et al., 2001], we notice that $E_0' = E_0$, and we then observe that $E_1' <: E_1 <: E_0 = E_0'$, so rule T-Update applies and $\Gamma \vdash [\bar{d}/\bar{x}]e : E_0$. We have now proved that $C' = E_0$ and $C = D = E_0$, completing the case.

**Case T-Loc** Straightforward, the type of a location is defined independently of the environment $\Gamma$.

This, along with the rest of the cases in [Igarashi et al., 2001] completes the proof. □

**Lemma 6.3 (Weakening).** *If* $\Gamma \vdash e : C$, *then* $\Gamma, x : D \vdash e : C$

*Proof.* As in [Igarashi et al., 2001], we omit this proof but note that it is based on straightforward induction. Note that since $e$ was assigned a type in the enviroment $\Gamma$, $e$ cannot depend on $x$ to type-check, and hence, the addition of an extra identifier to the environment does not change the resulting type of the expression. □

**Lemma 6.4.** *If* $mtype(m, C_0) = \bar{D} \to D$, *and* $mbody(m, C_0) = \bar{x}.e$, *then for some* $D_0$ *with* $C_0 <: D_0$, *there exists* $C <: D$ *such that* $\bar{x} : \bar{D}, this : D_0 \vdash e : C$.

*Proof.* This is proved by induction on the derivation of $\text{mbody}(m, C_0)$. If $m \in CT(C_0)$, we can use the rule T-METHOD since m is OK IN $C_0$, this immediately gives the desired result. Otherwise, $m \notin CT(C_0)$ and we go one step up the inheritance chain using the rules for mbody and mtype, since $\text{mbody}(m, C_0)$ and $\text{mtype}(m, C_0)$ are defined, the inheritance chain is finite and the induction hypothesis can be used. □

We now need to prove that the labels are used properly in the rules. We do that by proving that the list of locations pointed to by a location have one entry for each field variable of the class of which the location represents an instance, and that the types of the fields are consistent with what is stored in them.

**Lemma 6.5.** *If $l = (p, C)$, $ST(l) = (\bar{p}, \bar{D})$ and $\mathit{fields}(C) = \bar{C}\ \bar{f}$ then $\bar{D} <: \bar{C}$*

*Proof.* This is proved by enumerating the cases where the store is altered. There are only two rules that alter the store, R-NEW and R-UPDATE. The last transition to update the rule is the last transition with a label $L \notin \mathit{UnObs}$. This is because, by definition of the generalized labelled transition system, all labels in the deriviation chain are composable. Now, the unobservable labels does not change the store - in fact they guarantee it to be unaltered, so the last update must have been from a transition with an observable label and R-NEW and R-UPDATE are the only rules with such labels.

If the *last* rule to update $l$ was R-NEW then $l$ is the result of $\mathit{new}\ C((\bar{p}, \bar{D}))$, rule T-NEW then gives us $\bar{D} <: \bar{C}$.

Otherwise the last rule to update $l$ was R-UPDATE on some $f_i$. By T-UPDATE, $D_i <: C_i$, and by applying this theorem to $l$ before the update, the rest of $\bar{l}$ also satisfy $\bar{D} <: \bar{C}$. □

The following lemma, which is trivial to prove, is *not* stated explicitly in [Igarashi et al., 2001], but is needed to disregard identifiers as a valid case when considering different expression at runtime.

**Lemma 6.6 (Identifiers do not Survive).** *If $e$ is well-typed no identifiers are ever encountered during a computation.*

*Proof.* Since $e$ was well-typed, all identifiers, $x$ have been looked up in the environment using T-VAR. Thus all identifies appears as formal parameters of a function and are replaced by R-INVK during a computation. □

We now move on to the first of the theorems. The subject reduction theorem states that the type of an expression get more specific, in terms of the inheritance hierarchy, as the expression is evaluated by the semantic rules.

**Theorem 6.7 (Subject Reduction).** *If $\Gamma \vdash e : C$ and $e \to e'$, then $\Gamma \vdash e' : C'$ for some $C' <: C$*

*Proof.* This is proved by induction on a derivation of $e \to e'$, casing over the rule used. We only cover some of the rules for congruence, they are simple to verify. Assume that $\Gamma \vdash e : C$.

**Case R-NEW** $e = \text{NEW}\ C(\bar{l})$ $\quad e' = (p, C)$

By T-NEW $\Gamma \vdash e : C$. And by T-LOC we get that $\Gamma \vdash e' : C$, completing the case.

**Case R-INVK** $e = (p, C_0).m(\bar{l})$
$\text{mbody}(m, C_0) = \bar{x}.e_0$
$e' = [\bar{l}/\bar{x}, (p, C_0)/\mathit{this}]e_0$

By the rules T-INVK, T-LOC, and T-NEW we have

$$\Gamma \vdash (p, C_0) : C_0 \quad \text{mtype}(m, C_0) = \bar{D} \to C$$
$$\Gamma \vdash \bar{l} : \bar{C} \qquad\qquad \bar{C} <: \bar{D}$$

for some $\bar{C}$ and $\bar{D}$. Invoking Lemma 6.4 there are some $D_0$ and $B$ where $B <: C$ and $C_0 <: D_0$ giving $\bar{x} : \bar{D}, \mathit{this} : D_0 \vdash e_0 : B$. By the weakening lemma (6.3) we get that $\Gamma, \bar{x} : \bar{D}, \mathit{this} : D_0 \vdash e_0 : B$. And since term substitution preserves typing (6.2) there are some $E <: B$ so $\Gamma \vdash [\bar{l}/\bar{x}, (p, C_0)/\mathit{this}]e_0 : E$, and since $E <: B <: C$ we know, by the transitivity of $<:$, that $E <: C$. Setting $C' = C$ completes the case.

**Case R-FIELD** $e = l.f_i$
$ST(l) = \bar{l}$
$e' = l_i$

By the rule T-FIELD, $\Gamma \vdash l : C_0$ and $\text{fields}(C_0) = \bar{C}\ \bar{f}$. By Lemma 6.5 $\Gamma \vdash \bar{l} : \bar{D}$ where $\bar{D} <: \bar{C}$, setting $C = C_i$ and $C' = D_i$ completes the case.

**Case R-CAST** $e = (D)(p, C_0)$
$C_0 <: D$
$e' = (p, C_0)$

The assumption $C_0 <: D$ implies that the only way $e$ can be assigned the type $C$ is by the rule T-UCAST. By that rule, and T-LOC $C = D$, $\Gamma \vdash e' = C_0$, $C' = C_0$, and using $C_0 <: D = C$ we get that $C' <: C$ completing the case.

**Case RC-CAST** $e = (D)(e_0)$
$e_0 \to e_0'$
$e' = (D)(e_0')$

Depending on the rule used to check the type of $e$ there are three cases:

**Subcase T-UCAST** We know that $\Gamma \vdash e_0 : C_0$ and $C_0 <: D$. By the induction hypothesis we know that there exists a type $C_0'$, so $\Gamma \vdash e_0' : C_0'$ and $C_0' <: C_0$. By transitivity we know that $C_0' <: D$, then T-UCAST applies to $e'$ resulting in $D = C' = C$.

**Subcase T-DCast** We know that $\Gamma \vdash e_0 : C_0$, $D <: C_0$ and $D \neq C_0$. By the induction hypothesis we know that there exists a type $C_0'$, so $\Gamma \vdash e_0' : C_0'$ and $C_0' <: C_0$. If $C_0' <: D$ we can apply T-UCast to $e'$, yielding $D = C = C'$. If $D <: C_0'$ and $D \neq C_0'$ we can apply T-DCast to $e'$, yielding $D = C = C'$. If neither of the above cases apply then $C_0' \not<: D$ and $D \not<: C_0'$, and we can use T-SCast with a *stupid warning*.

**Sub-case T-SCast** We know that $\Gamma \vdash e_0 : C_0$, $D \not<: C_0$ and $C_0 \not<: D$. By the induction hypothesis we know that there exists a type $C_0'$, so $\Gamma \vdash e_0' : C_0'$ and $C_0' <: C_0$. We would like to use T-SCast on $e'$, in order to do that we need to prove that $D \not<: C_0'$ and $C_0' \not<: D$. Assume for the contrary that $D <: C_0'$, but since $C_0' <: C_0$, that would imply that $D <: C_0$, but this is a contradiction to our assumption so $D \not<: C_0'$. We then assume for the contrary that $C_0' <: D$, but since only have single inheritance and $C_0' <: C_0$, that would imply that either $D <: C_0$ or $C_0 <: D$ holds, contrary to our assumption.

The above proves that $D \not<: C_0'$ and $C_0' \not<: D$, so by T-SCast $D = C = C'$, with a *stupid warning*.

**Case R-Update** $e = l.f_i = \hat{l} \qquad e' = \hat{l}$

By the rule T-Update we get that $\Gamma \vdash l.f_i : C$, $\Gamma \vdash \hat{l} : C'$ and $C' <: C$ which immediately concludes the case.

**Case RC-Field** $e = e_0.f \qquad e' = e_0'.f$

Rule T-Field gives that $\Gamma \vdash e_0 : C_0$ and $\text{fields}(C_0) = \bar{C}\ \bar{f}$. By induction we know that $\Gamma \vdash e_0' : C_0'$ and $C_0' <: C_0$ for some $C_0'$. $\text{fields}(C_0)$ must be included in $\text{fields}(C_0')$ since $C_0' <: C_0$, so $f$ lies in both of the field lists with the same type, so $e$ and $e'$ has the same type by T-Field.

□

Having proved the subject reduction theorem, we now move on to state that we never call non-existing functions or access non-declared field-variables in well-typed AFJ expressions.

**Theorem 6.8 (Progress).** *Suppose $e$ is a well-typed expression, then*

1. *If $e$ includes $(p, C_0).f$ as a subexpression then $\mathit{fields}(C_0) = \bar{C}\ \bar{f}$ and $f \in \bar{f}$ for some $\bar{C}$ and $\bar{f}$.*
2. *If $e$ includes $(p, C_0).m(\bar{d})$ as a subexpression, then $\mathit{mbody}(m, C_0) = \bar{x}.e_0$ and $\#(\bar{x}) = \#(\bar{d})$ for some $\bar{x}$ and $e_0$.*
3. *If $e$ includes $(p, C_0).f = e_0$ as a subexpression then $\mathit{fields}(C_0) = \bar{C}\ \bar{f}$ and $f \in \bar{f}$ for some $\bar{C}$ and $\bar{f}$.*

*Proof.* We make a proof for each case.

1. If $e$ has $(p, C_0).f$ as a subexpression. By well-typedness $e$ has been validated by T-Field, so $\mathit{fields}(C_0)$ is defined and $f \in \bar{f}$.
2. If $e$ includes $(p, C_0).m(\bar{d})$ as a subexpression. By T-Invk $\text{mtype}(m, C_0)$ is defined, its easy to see that from the definitions of mbody and mtype, that if one is defined, so is the other. T-Invk also ensure that $\#(\bar{x}) = \#(\bar{d})$ since $\bar{C} <: \bar{D}$ is defined only when the lists has the same length.
3. If $e$ includes $(p, C_0).f = e_0$ as a subexpression the conclusion follows immediately from the first condition of rule T-Update.

□

We are now ready to prove, using mainly the subject reduction and progress theorems that when an expression is stuck, it is either because we are done and have computed a location or because we are stuck at an invalid cast. Note that if the expression $e$ is a normal form, none of the semantic rules can be used to evaluate $e$ further.

**Theorem 6.9 (AFJ Soundness).** *If $\emptyset \vdash e : C$ and $e \rightarrow^* e'$ with $e'$ a normal form, then $e'$ is either a location $(p, D)$ and $D <: C$, or an expression containing $(D)(p, C')$ where $C' \not<: D$.*

*Proof.* This proved by induction, considering the different kind of expression $e'$ could be.

Assume that $e'$ is an update, $e' = (e_0.f_i = e_1)$, since $\emptyset \vdash e' : C$ the rule T-Update implies that $\emptyset \vdash e_0.f_i : C$ and $\emptyset \vdash e_1 : D$ for some $D$. Then, by induction, when $e_1 \rightarrow^* e_1'$, $e_1'$ is either a location or contains an invalid cast. If $e_1'$ contains an invalid cast, we are done. Otherwise assume that $e_1' = l$ for some location, $l$. Then we turn to $e_0.f_i$, since $e_0.f_i$ had the type $C$ in the empty environment, by rule T-Field we know that $\emptyset \vdash e_0 : E_0$ for some $E_0$. Then when $e_0 \rightarrow^* e_0'$ we know by induction that $e_0'$ is either an invalid cast or an location. Assume for contradiction that $e_0'$ is a location, then by Theorem 6.8, we can apply R-Update and $e'$ could not have been on normal form; thus $e_0'$ contains an invalid cast. By subject reduction $\emptyset \vdash e' : D$, where $D <: C$.

When $e'$ is a field access or a method invocation the reasoning is similar to the above case and is therefore omitted. If $e' = \text{new } C_0(\bar{e})$ then the reasoning is reused from the above case as well, but since R-New has no premises, other than the one on labels, another step can always be taken unless $\bar{e}$ contains an invalid cast.

If $e'$ is cast, $e' = (D)e_0$, we know by induction and the reasoning above that $e_0$, when reduced to a normal form, is either a location or an invalid cast, if the latter is the case, we are done. Thus we assume that $e_0$ reduces to a location, $l$ where $\emptyset \vdash l : C'$. Now, if $C' <: D$ we could use R-CAST, contradicting that $e'$ was a normal form, so $C' \not<: D$, which is what we wanted.

Last, if $e'$ is simply a location, subject reduction concludes that $\emptyset \vdash e' : D$ where $D <: C$.

By Lemma 6.6 $e'$ cannot be an identifier, completing the proof. □

We have now shown that we only get a normal form if we are done and the output is a location, otherwise we are stuck at an invalid cast. We would now like to state that these invalid casts cannot appear in programs with no upcasts or stupid-casts, that property, along with Theorem 6.9 will enable us to state that all well-typed programs with no upcasts or stupid-casts will end up computing a location.

**Definition 6.10 (Cast-Safe).** We say that an well-typed program $(e, CT)$ is *cast-safe* if the rules T-DCAST and T-SCAST were not used in the type-inference of $e$ and CT.

We are now able to prove that the definition of cast-safe is sane with respect to reductions.

**Lemma 6.11 (Cast-Safety is Preserved by Reduction).** *If $e$ is cast-safe and $e \to e'$ then $e'$ is cast-safe.*

*Proof.* The proof is a trivial, using Theorem 6.7. For example, if the rule T-UCAST was used on $e = (D)e_0$ with $\Gamma \vdash e_0 : C$ with $C <: D$ and $e_0 \to e_0'$ then $\Gamma \vdash e_0' : C'$ where $C' <: C$ by Theorem 6.7. Using transitivity of $<:$ we conclude that $C' <: D$ and thus $e'$ is still cast-safe. □

We are now able to conclude that case-safe programs always compute a location.

**Corollary 6.12 (Cast-Safe Expressions Compute Values).** *If $e$ is cast-safe and $e \to^* e'$ with $e'$ a normal form, then $e'$ is a location $l$.*

*Proof.* This follows from Lemma 6.11 and Theorem 6.9. Theorem 6.9 states that $e'$ is either a location or an invalid cast, and by Lemma 6.11 $e'$ is cast-safe and can therefore not contain any invalid casts. □

# 7 Type-checking

In order to test the typing aspects of AFJ in practice we have implemented a type-inferrer in Standard ML of New Jersey (SML/NJ). The Standard ML code has been tested on SML/NJ, Version 110.0.6.

The type-inference works on the data types found in Listing 4. Note that the expression datatype does not include locations, this is due to the fact that locations only exists at runtime in the abstract syntax tree and they are never present in user-written code.

```
datatype classn = CLASSN of string
datatype methodn = METHODN of string
datatype fieldn = FIELDN of string
datatype identifier = IDENTIFIER of string
datatype e = LOOKUP of identifier
           | FIELD of e * fieldn
           | METHOD of e * methodn * (e list)
           | NEW of classn * (e list)
           | CAST of classn * e
           | UPDATE of e * fieldn * e
datatype methodd = METHODDEF of
  classn * methodn * (classn * identifier) list * e
datatype class = CLASS of
  classn * classn * (classn * fieldn) list * methodd list
datatype program = PROGRAM of class list * e
```

Listing 4: The datatypes making up an AFJ program.

The implementation of the inference rules has been relatively straight forward using the type inference rules as a starting point. The auxiliary functions were more tedious to write, but conceptually simple. As seen in the following listing it is very easy to relate the implementation of the type-checker with the theoretical specifications of this article.

```
fun typecheck(g,cl,NEW(c,el)) =
    let
      val bd = map (fn (a,_) => a) (fields(cl, cl, c))
      val bc = map (fn e => typecheck(g,cl,e)) el
    in
      if subtypel(cl,bc,bd) then
        c
      else
        raise NotASubtype
    end
```

The listing shows the ML code used to infer the type of an expression of the form "new $C(e_1)$". First the type of the field variables are extracted, then the types of the arguments to the constructer are inferred, and last it is checked that the arguments are subtypes of the field variables.

The signature, seen in Listing 5, of the type-checker shows the functionality of our system. We did not write a lexer or parser to supplement the type-checker, so we

```
signature TYPECHECKER = sig
  val main: program -> classn
  val prettyPrint: program -> unit
end
```

Listing 5: Signature of the type-checker

had to construct the AFJ programs used for testing, directly in the abstract syntax. This was very tedious and error-prone. To better understand the abstract syntax created, we implemented code to print the abstract syntax tree (AST) in a human-readable manner, this printing is legal Java code. The printed code can be passed on to a Java compiler and runtime system for more investigation. The following listing shows the Java program generated from a simple AFJ program.

```
public class AFJ2Java {
    /* Automatically generated Java Code */
    public static class C extends Object {
        public C() { super(); }
    }
    public static class D extends Object {
        public C sdf;
        public D(C sdf) {
            super();
            this.sdf=sdf;
        }
        public C set(C x) {
            return (this).sdf=x;
        }
    }
    /* Execution interface */
    public static void main(String[] args) {
        Object o=(D)new D(new C()).set(new C());
    }
}
```

The AFJ source program used to generate the above Java code was

```
class C extends Object {
    C() { super(); }
}
class D extends Object {
    C sdf;
    D(C sdf) {
        super();
        this.sdf=sdf;
    }
    C set(C x) {
        return (this).sdf=x;
    }
}
```

and the expression to evaluate using the above classes was, (D)**new** D(**new** C()).set(**new** C());

The type-checker correctly inferred that the type of the last expression was $D$. It also produced a stupid warning, since the return type of the `set` function is $C$ and $C$ is not in a subtype relation with $D$. Note the different behaviour of AFJ and Java in this respect. Our standard SUN Java compiler refuses to compile the above program solely due to the stupid cast, wheres AFJ only specifies that a warning should be issued.

This means that we have effectively written a simple (abstract) AFJ to Java compiler, and since the code was type-checked by us, there should be no need to enable the type-checking in the Java compiler. Pedantically, one should of course prove that our type-checker is a correct implementation of the type rules. But then it should also be proved that the CPU and SML/NJ runtime system is correct, the list goes on, and it is quite infeasible to do all that in a lifetime.

Along with the source code several test cases are provided, of interest is the example where an illegal downcast passes unnoticed through our type-checker, just like the Java type-checker. The test cases also include an example with a stupid cast, and many other AFJ programs of varying interest.

# 8 Final Remarks

We constructed AFJ as an extension of FJ with mutable state, and proved that well-typed programs only get prematurely stuck if they contain invalid casts. Implementing the type-checker helped us understand some of the design decisions made in FJ and also gave a better view on how the rules work together.

# References

[Bierman et al., 2003] Bierman, G., Parkinson, M., and Pitts, A. (2003). MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, Cambridge University Computer Laboratory.

[Drossopoulou et al., 1999] Drossopoulou, S., Eisenbach, S., and Khurshid, S. (1999). Is the Java type system sound? *Theory and Practice of Object Systems*, 5(1):3–24.

[Igarashi et al., 2001] Igarashi, A., Pierce, B. C., and Wadler, P. (2001). Featherweight Java: A minimal core calculus for Java and GJ. *ACM/Transactions on Programming Languages and Systems*, 23(3):396 – 450.

[Mosses, 2003] Mosses, P. D. (2003). Fundemental conceps and formal semantics of programming languages.

[Odersky and Wadler, 1997] Odersky, M. and Wadler, P. (1997). Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 146–159. ACM Press, New York (NY), USA.

[Saraswat, 1997] Saraswat, V. (1997). Java is not type-safe.