IFJ: a Minimal Imperative Variant of FJ *

Lorenzo Bettini† Ferruccio Damiani‡ Ina Schaefer

1 Dipartimento di Informatica, Università di Torino, C.so Svizzera, 185 - 10149 Torino, Italy
2 Chalmers University of Technology, 421 96 Gothenburg, Sweden


Abstract
In this paper we present IFJ (IMPERATIVE FJ), minimal imperative variant of FJ (FEATHERWEIGHT JAVA). We have designed IFJ with the aim of providing an imperative minimal core calculus for JAVA.

Categories and Subject Descriptors D.1.5 [Programming Techniques]: Object-oriented Programming; D.3.3 [Programming Languages]: Language Constructs and Features; F.3.3 [Studies of Program Constructs]: Type Structure

General Terms Design, Languages, Theory

Keywords Java, Type System

1. Introduction
The type soundness of IFJ can be formulated, similar as for FJ in [1], as a subject reduction theorem for a small-step semantics. Since FJ has no imperative constructs, it is possible to formulate its small-step semantics without introducing runtime expressions [1]. Instead, to deal with the imperative construct (field assignment) of IFJ, the small-step semantics of IFJ formalizes the notion of address, value, object, heap and runtime expression.

2. Imperative Featherweight Java
2.1 IFJ Syntax
The abstract syntax of the IFJ constructs is given in Figure 1. Following [1], we use the overline notation for possibly empty sequences. For instance, we write “¯e” as short for a possibly empty sequence of expressions “e1,...,en” and “MD” as short for a possibly empty sequence of method definitions “MD1...Mn” (without commas). The empty sequence is denoted by пусто. The length of a sequence пусто is denoted by #(пусто). We abbreviate operations on sequences of pairs in similar way, e.g., we write “C f” as short for “C; f1,...,fn” and “C ¯f” as short for “C; f1,...,fn”. Sequences of named elements (field, method or parameter names, field, method or class definitions,…) are assumed to contain no duplicate names (that is, the names of the elements of the sequence must be distinct). The set of variables includes the special variable this (implicitly bound in any method declaration), which cannot be used as the name of a method’s formal parameter.

A class definition class C extends D { FD; MD } consists of its name C, its superclass D (which must always be specified, even if it is Object), a list of field definitions FD and a list of method definitions MD. The fields declared in C are added to the ones declared by D and its superclasses and are assumed to have distinct names (i.e., there is no field shadowing). All fields and methods are public. Each class is assumed to have an implicit constructor that initializes all instance variables to null.

A class table CT is a mapping from class names to class definitions. The subtyping relation <: on classes (types) is the reflexive and transitive closure of the immediate extends relation (the immediate subclass relation, given by the extends clauses in CT). The class Object has no members and its definition does not appear in CT. We assume that a class table CT satisfies the following sanity conditions: (i) CT(C) = class C... for every C ∈ dom(CT) (ii) for every class name C (except Object) appearing anywhere in CT, we have C ∈ dom(CT); (iii) there are no cycles in the transitive closure of the immediate extends relation.

An IFJ program is a class table CT. A class definition CD can be understood as a mapping from the keyword extends to a class name and from field/method names to field/method definitions. We use the metavariable a to range over field/method names, and the metavariable AD to range over field/method definitions. The lookup of the definition of a field/method a in class C is denoted by aDef(C)(a). For every class C in dom(CT), the function aDef(C) is defined as follows:

aDef(C)(a) = { CT(C)(a) if a ∈ dom(CT(C)) aDef(D)(a) if a ∉ dom(CT(C)) and CT(C)(extends) = D

Figure 1. IFJ: syntax of classes

CD ::= class C extends C { FD; MD } classes
FD ::= C f fields
MD ::= C m (C x)(return e;) methods
e ::= x | e.f | e.m(пусто) | new C() | (C)e expressions
e.f = e null
Given a field definition \( FD = C \, f \) and a method definition \( MD = C \, m(\bar{x} \mid \cdots) \), we write \( \text{signature}(FD) \) to denote the type \( C \) of the field \( f \) and \( \text{signature}(MD) \) to denote the type \( \bar{C} \rightarrow \bar{C} \) of the method \( m \).

### 2.2 IFJ Typing

A class signature \( CS \) is a class definition deprived of the bodies of its methods. The abstract syntax is as follows:

\[
\text{CS} ::= \text{class } C \text{ extends } C \{ FD; RH \} \quad \text{class signatures}
\]

\[
\text{MH} ::= C \, m(\bar{x}) \quad \text{method headers}
\]

A class signature table \( CST \) is a mapping from class names to class signatures. We write \( \text{signature}(CT) \) to denote the class signature table consisting of the signatures of the classes in the class table \( CT \).

The lookup of the type of a field/method \( a \) in the signature of the class \( C \) is denoted by \( \text{aType}(C)(a) \). For every class \( C \) in \( \text{dom}(CST) \), the function \( \text{aType}(C) \) is defined as follows:

\[
\text{aType}(C)(a) = \begin{cases} 
\text{CST}(C)(a) & \text{if } a \in \text{dom}(\text{CST}(C)) \\
\text{aType}(\text{D})(a) & \text{if } a \notin \text{dom}(\text{CST}(C)) \\
\text{and CST}(C)(\text{extends}) = D 
\end{cases}
\]

The subtyping relation \( <: \) can be read off from the class signature table such that it is possible to check whether there are no cycles in the transitive closure of the \( \text{extends} \) relation. Moreover, by inspecting a class signature table, it is possible to check, for every class \( C \) in \( \text{dom}(CST) \), whether the names of the fields defined in \( C \) are distinct from the names of the fields inherited from its superclasses, and whether the type of each method defined in \( C \) is equal to the type of any method with the same name defined in any of the superclasses of \( C \). Therefore, in the following we can safely assume that a class signature table satisfies the following sanity conditions:

(i) \( \text{CS}(C) = \text{class } \cdots \) for every \( C \in \text{dom}(\text{CS}) \);

(ii) for every class name \( C \) (except \text{Object}) appearing anywhere in \( \text{CS} \), we have \( C \in \text{dom}(\text{CS}) \);

(iii) the subtyping relation \( <: \) is acyclic;

(iv) \( C_1 <: C_2 \) implies that, for all method names \( m \), if \( \text{aType}(C_2)(m) \) is defined then \( \text{aType}(C_1)(m) = \text{aType}(C_2)(m) \); and

(v) \( C_1 <: C_2 \) and \( C_1 \neq C_2 \) imply that, for all field names \( f \), if \( f \in \text{dom}(\text{CST}(C_1)) \) then \( f \notin \text{dom}(\text{CST}(C_2)) \).

In order to type the \( \text{null} \) value, we introduce the special type \( \perp \), that cannot occur in IFJ programs and is a subtype of any other type. We will use the metavariable \( T \) to denote either a class name or \( \perp \).

The IFJ typing rules are given in Figure 2. A type environment \( \Gamma \) is a mapping from variables (including \text{this}) to class names, written \( \bar{x} : C \) ; the empty environment will be denoted by \( \bullet \). The rules for variable \( \langle \text{T-VAR} \rangle \), field selection \( \langle \text{T-FIELD} \rangle \), method invocation \( \langle \text{T-INVK} \rangle \), object creation \( \langle \text{T-NEW} \rangle \), upcast \( \langle \text{T-UCAST} \rangle \), downcast \( \langle \text{T-DCAST} \rangle \), method definition \( \langle \text{T-METHOD} \rangle \) and class definition \( \langle \text{T-CLASS} \rangle \) are analogous to the corresponding rules for FJ given in [1]. However, the presentation is slightly different since our rules refer to the class signature table of the program rather than to the class table. In particular, the rule for typing the definition of a method \( m \) in a class \( C \), \( \langle \text{T-METHOD} \rangle \), relies on the fact that, according to the sanity condition \( iv \) of the class signature table, any definition of a \( a \) the method with name \( m \) in a superclass of \( C \) must have the same type. We also have a rule for \( \text{null} \) and a rule for field assignment (not contained in FJ) and a rule for typing the whole program (left implicit in FJ). Note that expression like \( C e \) where the type of \( e \) is not a subtype of \( C \) (called \textit{stupid casts} in [1]) or \( \text{null} \, f \) and \( \text{null} \, m(\cdots) \) (that we will call \textit{stupid selections}) are ill typed.

### Expression Typing

\[
\begin{align*}
\Gamma \vdash x : \Gamma(x) & \quad \langle \text{T-VAR} \rangle \\
\Gamma \vdash \text{null} \cdot \perp & \quad \langle \text{T-NULL} \rangle \\
\Gamma \vdash e : C & \quad \langle \text{T-TYP} \rangle \\
\Gamma \vdash \text{aType}(C)(f) = A & \quad \langle \text{T-FIELD} \rangle \\
\Gamma \vdash e : f : A & \quad \langle \text{T-ASSIG} \rangle \\
\Gamma \vdash e : C & \quad \langle \text{T-NEW} \rangle \\
\Gamma \vdash e \cdot m(\bar{x}) : B & \quad \langle \text{T-INVK} \rangle \\
\Gamma \vdash e : T & \quad \langle \text{T-UCAST} \rangle \\
\Gamma \vdash e : C & \quad \langle \text{T-DCAST} \rangle \\
\end{align*}
\]

### Method Definition Typing

\[
\begin{align*}
\text{this} : C ; \bar{x} : A & \vdash e : T \quad \langle \text{T-ASSIG} \rangle \\
\end{align*}
\]

### Program Typing

\[
\begin{align*}
\forall C \in \text{dom}(CT) & , \quad \vdash CT(C) \quad \langle \text{T-CLASS} \rangle \\
\end{align*}
\]

**Figure 2.** IFJ: Typing rules for expressions, methods and program \( CT \) w.r.t. the class signature table \( CST = \text{signature}(CT) \)

Following [1], we say that a well-typed IFJ program is \textit{cast safe} to mean that it can be typed without using the rule for downcast. Every well-typed IFJ is literally a well-typed JAVA program.

### 2.3 IFJ Reduction

In order to properly model imperative features we introduce the concepts of \textit{address} and \textit{heap}. Addresses, ranged over by the metavariable \( \tau \), are the elements of the denumerable set \( \text{I} \). Values, ranged over by the metavariable \( v \), are either addresses or \text{null}. Objects are denoted by \( \langle C, \bar{x} = v \rangle \), where \( C \) is the class of the object, \( \bar{x} \) are the names of the fields and \( v \) are the values of the fields. A heap \( H \) is a mapping from addresses to objects. The empty heap will be denoted by \( \emptyset \). Runtime expressions are obtained from expressions by replacing all the variables (including \text{this}) by addresses. We will use \( e \) to denote runtime expressions.

The states of a computation are represented by means of configurations. A configuration is a pair consisting of a heap and a runtime expression, written \( H, e \). The reduction relation has the form \( H, e \rightarrow H', e' \), to read “the configuration \( H, e \) reduces to the configuration \( H', e' \) in one step”. The initial con-
In order to be able to formulate the type soundness of IFJ as a subject reduction theorem for the small-step semantics, we need to formulate a type system for run-time expressions. Expressions containing either a stupid cast (a notion introduced in [1]), i.e., a cast where the subject and the target are unrelated, or a stupid selection, i.e., a field selection null.f or a method invocation null.m(···), are not well typed according to the IFJ (source level) type system. However, a run-time expression without stupid casts and stupid selections may reduce to a run-time expression containing either a stupid cast or a stupid selection. The type system for run-time expressions contains a rule for typing stupid casts, and a rule for assigning any type T to the value null (so that stupid selection can be typed).

Typing rules for runtime expressions are shown in Figure 5; these rules use the environment Σ, which is a finite (possibly empty) mapping from addresses to class names, and they are of the shape Σ ⊢ t : T. In Figure 5 we also present the notion of well-formed heap and of well-formed configuration. The notion of well-formed heap ensures that the environment Σ maps all the addresses in the heap into the type of the corresponding object and that for every object stored in the heap, the fields of the object contain appropriate values.
Type soundness can be proved by using the standard technique of subject reduction and progress theorems.

**Lemma 2.1.** If $\texttt{aType}(C_0)(m) = D \rightarrow D$ and $\texttt{mbody}(C_0, m) = \langle \bar{z}, e \rangle$ then for some $D_0$ and some $T : <D$ we have $C_0 : <D_0$ and this : $D_0 : \bar{z} : D \rightarrow e : T$.

**Proof.** By straightforward induction on the derivation of $\texttt{mbody}(C_0, m)$, that is, on $\texttt{aDef}(C_0)(m)$.

**Lemma 2.2.** (Substitution). If
1. $\Sigma \vdash t(m, \bar{v}) : D$ where $\Sigma(t) = C_0$ for some $\Sigma, C_0$ and $D$,
2. $\texttt{aType}(C_0)(m) = \bar{\bar{A}} \rightarrow D$, and
3. $\texttt{mbody}(C_0, m) = \langle \bar{z}, e \rangle$,

then for some $C' : <D$ we have $\Sigma \vdash t(\bar{v}, \bar{x}, \bar{\bar{e}}) : e : C'$.

**Proof.** By hypothesis 1. and 2. and by Lemma 2.1. for some $C$ and some $T : <D$ we have $C_0 : <C$ and this : $C : \bar{\bar{A}} : e : T$.

The proof then proceeds by structural induction on the derivation of $\texttt{this} : C : \bar{\bar{A}} : e : T$. We present only a few interesting cases (the cases for casts are the same as in FJ, in particular, for (\texttt{T-CAST}) we can use (\texttt{RT-CAST})). Note that, by rule (\texttt{RT-INV}), $\Sigma \vdash \bar{v}, \bar{\bar{C}} : C$ for some $\bar{\bar{C}}$ such that $C : <\bar{\bar{A}}$ (in particular, $C_i : A_i$ when $v_i$ is null by rule (\texttt{RT-NUL})).

**Case (T-VAR)** In this case $e = x_i$ for some $x_i \in \bar{x}$; $\bar{\bar{v}} \vdash \bar{\bar{C}}$, $\bar{\bar{v}} : \bar{x}, \bar{\bar{e}} : i : \bar{\bar{C}}$. For some $C_i$ such that $C_i : <A_i$; letting $C_i = \bar{C}$ finishes the case.

**Case (T-FIELD)** In this case $e = e'.f$. By (\texttt{T-FIELD}) we have $\texttt{this} : C : \bar{\bar{A}} : e' : C'$ and $\texttt{aType}(C')(e') = \bar{\bar{A}}$. By the induction hypothesis, $\Sigma \vdash e' : C'$ for some $C' : <C'$.

Thus the thesis follows from $\texttt{aType}(C')(e) = \texttt{aType}(C')(e') = \bar{\bar{A}}$.

**Case (T-INV)** In this case $e = e'.\bar{m}(\bar{e})$. Similar to the previous case, using the induction hypothesis on $e'$ and $\bar{e}$, and using the fact that $\texttt{aType}(C')(e) = \texttt{aType}(C')(e')$ if $C' : <C'$.

**Case (T-ASSG)** In this case $e$ is of the shape $e_0.f = e$. By (\texttt{T-ASSIG}) we have $\texttt{this} : C : \bar{\bar{A}} : e_0 : \bar{\bar{A}}$, this : $C : \bar{\bar{A}} : e_0 : T_1$ for some $T_1 : <A$. The thesis follows from the induction hypothesis and the transitivity of $\vdash$.

**Lemma 2.3.** (Weakening). If $\Sigma \vdash e : T$ then $\Sigma, t : C \vdash e : T$.

**Proof.** Straightforward induction on the derivation of $\Sigma \vdash e : T$.

**Theorem 2.4.** (Subject reduction). If $\Sigma \vdash \mathcal{H}, \Sigma \vdash e : T$ and $\mathcal{H}, e \rightarrow \mathcal{H}', \mathcal{H}'$ then there exists $\mathcal{H}_1 \supseteq \mathcal{H}$ such that $\Sigma \vdash \mathcal{H}_1, \Sigma \vdash e' : T$ for some $T' : <T$.

**Proof.** The proof is by induction on a derivation of $\mathcal{H}, e \rightarrow \mathcal{H}', \mathcal{H}'$, with a case analysis on the reduction rule used. We show only the most interesting cases for computation rules; for congruence rules simply use the induction hypothesis (using Lemma 2.3).

**Case (R-FIELD)** The last applied rule is $\mathcal{H}(t).t.f : \mathcal{H}, \mathcal{V}$, where $\mathcal{H}(t) = \langle C, f = \bar{v} \rangle$. By hypothesis $\Sigma \vdash t.f : C_i$ and by (\texttt{WF-HEAP}) we have $\Sigma \vdash \bar{v} : T_i$ for some $T_i : <C_i$. Thus we have the thesis.

**Case (R-INV)** The last applied rule is

$$\mathcal{H}(t) = \langle C, f = \bar{v} \rangle \quad \texttt{mbody}(m, C) = \langle \bar{z}, e_0 \rangle$$

$$\mathcal{H}, t.m(\bar{v}) \rightarrow \mathcal{H}, \langle \bar{v}, \bar{\bar{e}} \rangle : i : e_0$$

Since by hypothesis $\Sigma \vdash t.m(\bar{v}) : B$ the thesis follows by applying Lemma 2.2.

**Case (R-NEW)** Let $\mathcal{H}' = \Sigma \cup \{ i \vdash C \}$. By hypothesis $\Sigma \vdash \mathcal{H}$, and by applying (\texttt{WF-HEAP}) we also have $\mathcal{H}' \vdash \mathcal{H} \cup \{ t \vdash (C, i = \texttt{null}) \}$. $\mathcal{H}' \vdash t : \mathcal{C}$ follows from (\texttt{RT-VAR}).

**Case (R-ASSG)** By rule (\texttt{RT-ASSIG}) we have that $\Sigma \vdash v : T'$ and $T' : <T$ for some $T'$. By hypothesis $\Sigma \vdash \mathcal{H}$, and by applying (\texttt{WF-HEAP}) we also have $\Sigma \vdash \mathcal{H}[\mathcal{H}(t) \rightarrow (C, \ldots, f_i = \bar{v}, \ldots)]$.

**Lemma 2.5.** Let $\mathcal{H}, e$ be a well-typed configuration.

1. If $e = t.f$ then $\mathcal{H}(t) = \langle \bar{v}, \cdots \rangle$ with aType$(C)(f) = A$ for some class name $A$.
2. If $e = t.m(\bar{v})$ then $\mathcal{H}(t) = \langle \bar{v}, \cdots \rangle$ with aType$(C)(m) = A \rightarrow B$ and $\mathcal{H}[\bar{v}] = \bar{t}$.($\bar{t}$).

**Proof.** Straightforward.

**Theorem 2.6.** (Progress). Let $\mathcal{H}, e$ be a well-typed normal form. Then

1. either $e$ is a value, or
2. for some evaluation context $E$ we can express $e$ as
   (a) $\text{either } E[\bar{t}(A)\bar{t}]$ such that $\mathcal{H}(t) = \langle \bar{v}, \cdots \rangle$ with $B \not\vdash A$, or
   (b) $E[\text{null}, f]$ for some $f$, or
   (c) $E[\text{null}, m(\bar{v})]$ for some $m$ and $\bar{v}$.

**Proof.** Straightforward induction on typing derivations.

**Lemma 2.7.** If $\bullet \vdash e : T$ then $\bullet \vdash e : T$.

**Proof.** Straightforward induction on typing derivations.

**Theorem 2.8.** (Type Soundness). If $\vdash C : T_1.OK$, $C = \text{class } \text{Main} \{ \text{return } e; \}$. $\vdash e : T$ and

θ, $e -> \mathcal{H}, e'$ with $\mathcal{H}, e'$ a normal form. Then $e'$ is

1. either null,
2. or an address $t$ such that $\mathcal{H}(t) = \langle C, \cdots \rangle$ with $C : <T$,
3. or an expression containing $(A)t$ such that $\mathcal{H}(t) = \langle B, \cdots \rangle$ with
   $B \not\vdash A$,
4. or an expression containing either $\text{null}, f$ or $\text{null}, m(\bar{v})$ for
   some $f$, $m$ and $\bar{v}$.

**Proof.** Follows from Lemma 2.7, Theorem 2.4 and Theorem 2.6.

**References**