Type Safe Dynamic Object Delegation in Class-based Languages

Lorenzo Bettini Viviana Bono
Dipartimento di Informatica, Università di Torino
Corso Svizzera 185, 10149 Torino - Italy

ABSTRACT
Class inheritance and method overriding, as provided by standard class-based languages, are not flexible enough to represent the dynamic behavior of objects; with this respect, object composition and delegation are often advocated as a more flexible alternative to class inheritance since they act at run-time, thus permitting the behavior of objects to be specialized dynamically. In this paper we present Incomplete Featherweight Java (IFJ), an extension of Featherweight Java with incomplete objects, i.e., objects that require some missing methods which can be provided at run-time by composition with another (complete) object. The mechanism for method invocation is based on delegation and it is disciplined by static typing, therefore the language enjoys type safety (which implies no “message-not-understood” run-time errors) and avoids possible accidental overrides due to method clashes.

Categories and Subject Descriptors: D.3.3 [Programming Languages]: Language Constructs and Features

General Terms: Languages, Theory

Keywords: Language extensions, Featherweight Java, Object Composition, Delegation.

1. INTRODUCTION
Standard mechanisms provided by class-based object oriented languages, such as inheritance and dynamic binding, usually do not suffice for representing the dynamic behavior of objects. To overcome these limitations, design patterns [15] were introduced as “programming recipes” to be used in specific scenarios. Most of the design patterns in [15] rely on object composition as an alternative to class inheritance, since it is defined at run-time and it enables dynamic object code reuse by assembling existing components. Patterns exploit the programming style consisting in writing small software components (units of reuse), that can be composed at run-time in several ways to achieve software reuse. However, design patterns require manual programming and their correct implementation cannot be checked statically, since they do not represent a linguistic construct with a type system and a formal semantics.

Differently from class-based languages, object-based languages use object composition and delegation, a more flexible mechanism, to reuse code (see, e.g., the languages [24, 16, 9], and the calculi [14, 1]). Every object has a list of parent objects: when an object cannot answer a message it forwards it to its parents until there is an instance that can process the message. However, a drawback of delegation is that run-time type errors (“message-not-understood”) can arise when no delegates are able to process the forwarded message [25] (Kniesel [18] presents an overview of problems when combining delegation with static type discipline).

In this paper we present a possible solution for implementing object composition and delegation in Java-like languages. In particular, we formalize these linguistic constructs in Incomplete Featherweight Java (IFJ), an extension of Featherweight Java [17, 21] that combines the static type discipline of class-based languages with the flexibility of object-based ones. The programmer, besides standard classes, can define incomplete classes whose instances are incomplete objects that can be composed in an object-based fashion. Hence, in our calculus it is possible: (i) to instantiate standard classes, obtaining fully-fledged objects ready to be used; (ii) to instantiate incomplete classes, obtaining incomplete objects that can be composed (by object composition) with complete objects, thus yielding new complete objects at run-time.

Incomplete classes, besides standard method definitions, can declare some methods as “incomplete” (either abstract or redefining); these missing methods must be provided during object composition by complete objects. Thus, incomplete objects represent the run-time version of inheritance and dynamic binding mechanisms. This is a sort of dynamic inheritance since it implies both substitutivity (that is, a composed object can be used where a standard object is expected) and dynamic code reuse (since composition permits supplying at run-time the missing methods with those of other objects). Therefore, we can model some features related to dynamic object evolution: while incomplete classes separate the object invariant behavior from the variant one at compile time, at run-time object composition customizes the unpredictable behavior based on dynamic conditions (for instance, the object state) in a type safe way. In particular, some behavior that was not foreseen when the class hierarchy was implemented might be supplied dynamically by making use of already existing objects, thus generating an unanticipated reuse of code and a sharing of components.

In the following, we comment on our key design choices. Firstly, we keep the nominal subtyping mechanism that is typical of mainstream languages like Java and C++. This feature defines an extension that is conservative with respect to the core Java, since it...
does not affect those parts of the programs that do not use incomplete objects. Furthermore, incomplete classes can rely on standard class inheritance to reuse code of parent classes (although this kind of inheritance does not imply subtyping in our setting). Thus incomplete objects provide two forms of code reuse: vertical (i.e., the code reuse achieved via standard class inheritance) and horizontal (i.e., the one achieved via object composition). Finally, in order to enhance run-time flexibility in composing objects we implicitly use structural subtyping during composition: an incomplete object can be composed with any object providing all the required methods (the signatures must match) independently of the classes of these objects. Therefore, the language extension we propose is not a simple automatic implementation of the object composition that one might implement manually. In fact, any object providing the required methods can be used in object composition, no matter what its class is. In case of a manual implementation, instead, the object should be stored in a class field, thus forcing it to belong to a specific class hierarchy (see Section 2.2).

The first version of IFJ was proposed in [5]: the main novelty in this paper is that we use the delegation mechanism for method invocation on objects that are obtained by composition, while in [5] we only implemented consultation. We note that, in the literature (e.g., [15]), the term delegation, originally introduced by Lieberman [20], is given different interpretations and it is often confused with the term consultation. In both cases an object A has a reference to an object B. However, when A forwards to B the execution of a message m, two different bindings of the implicit parameter \( \text{this} \) can be adopted for the execution of the body of \( m \): with delegation, \( \text{this} \) is bound to the sender (A), thus, if in the body of the method \( m \) (defined in B) there is a call to a method \( n \), then also this call will be executed binding \( \text{this} \) to A; with consultation, during the execution of the body the implicit parameter is always bound to the receiver B. Providing delegation, instead of consultation, enhances the flexibility of object composition in IFJ and makes dynamic method redefinition effective (in [5] we did not provide method redefinition for incomplete objects); moreover, it requires some interesting technical treatments to achieve a correct implementation of type safe delegation in Java-like languages. For instance, we need to avoid possible name clashes for methods with the same name but with different signatures (possibly due to the subtyping, [14]) and possible accidental method overrides (when a method in the incomplete object, which is not redefining, has the same name and signature of a method of the complete object). In order to achieve this, we employ a static annotation procedure (based on static types) that will be used in the operational semantics to bind \( \text{this} \) correctly in method bodies.

Summarizing, our main intention is to have a language with a tradeoff between the dynamic flexibility that is typical of object-based languages and the static discipline of class-based languages. Objects are then still instances of (possibly incomplete) classes and they are still disciplined by the nominal subtyping, but they are also prototypes that can be used, via the object composition, to create new objects at run-time, while ensuring statically that the composition is type safe. Finally, object composition is the run-time version of class inheritance and delegation in composed objects corresponds to dynamic binding for method invocation in standard derived classes.

The paper is structured as follows: Section 2 presents our language, its typing system and semantics and some programming examples. Section 3 sketches the main steps to show that our language enjoys the type safety property which guarantees that in a well-typed program no “message-not-understood” can occur at run-time (this will also exclude possible run-time accesses to methods of an incomplete object). Section 4 concludes the paper and relates our language to some literature.

2. INCOMPLETE FEATHERWEIGHT JAVA

In this section we present syntax, typing and operational semantics of our proposal, the core language IFJ (Incomplete Featherweight Java), which is an extension of FJ (Featherweight Java) with incomplete objects, dynamic object composition and delegation. FJ [17, 21] is a lightweight version of Java, which focuses on a few basic features: mutually recursive class definitions, inheritance, object creation, method invocation, method recursion through \( \text{this} \), subtyping and field access. Thus, the minimal syntax, typing and semantics make the type safety proof simple and compact, in such a way that FJ is a handy tool for studying the consequences of extensions and variations with respect to Java (“FJ’s main application is modeling extensions of Java”, [21], page 248). Although we assume the reader is familiar with FJ, we will briefly comment on the FJ part and then we will focus on the novel aspects introduced by IFJ.

The abstract syntax of the IFJ constructs is given in Figure 1 and it is just the same as FJ extended with incomplete classes, abstract methods, redefining methods and object composition (and some run-time expressions that are not written by the programmer, but are produced by the semantics, that we will discuss later, Section 2.4). The metavariables \( B, C, D \) and \( E \) range over class names (both concrete and incomplete); \( M \) ranges over (standard) method definitions, \( N \) ranges over (abstract) method signatures and \( R \) ranges over redefining method definitions; \( f \) and \( g \) range over attribute names; \( x \) ranges over method parameter names; \( e \) and \( d \) range over expressions and \( v \) and \( u \) range over values. As in FJ, we will use the overline notation for possibly empty sequences (e.g., \( \overline{x} \) is a shorthand for a possibly empty sequence “\( e_1 \ldots e_n \)”). We abbreviate pairs of sequences in a similar way, e.g., \( \overline{C} \overline{x} \) stands for \( C_1 f_1, \ldots, C_n f_n \). The assignment \( \text{this} = \overline{T} \) stands for \( \text{this} = f_1; \ldots; \text{this} = f_n; \). The empty sequence is denoted by \( \varepsilon \).

Following FJ, we assume that the set of variables includes the special variable \( \text{this} \) (implicitly bound in any method declaration), which cannot be used as the name of a method’s formal parameter (this restriction is imposed by the typing rules). In IFJ we also introduce the special variable, \( \text{next} \): in a redefining method body, with \( \text{next} \), one can access the “redefined” object. For instance, in a redefining method \( m \) we can access the redefined version with \( \text{next} . m \). Thus, \( \text{next} \) is the dynamic (and horizontal) version of

---

1FJ also includes up and down casts; however, since these features are completely orthogonal to our extension, they are omitted in IFJ.

2An alternative choice could have been to allow a redefining method to access only the redefined version of the method, and not the whole “next” object; however, our choice is in line with the mainstream programming languages, although it is considered poor style [23].
super (intended as in the full Java language, not only as the call to the super-constuctor). Just like this, next is a variable that will be implicitly bound in redef methods. Note that since we treat this and next in method bodies as ordinary variables, no special syntax for it is required.

A class declaration class C extends D { ⟨ C T; K; R⟩ } consists of its name C, its superclass D (which must always be specified, even if it is Object), a list of field names CT with their types, the constructor K, and a list of method definitions R. The fields of C are added to the ones declared by D and its superclasses and are assumed to have distinct names. The constructor declaration shows how to initialize all these fields with the received values. A method definition R specifies the name, the signature and the body of a method; a body is a single return statement since FJ is a functional core of Java. In the following, we will write m ∉ R to mean that the method definition of the name m is not included in R. The same convention will be used for K and for method signatures R.

An incomplete class declaration class C abstracts D { ⟨ C T; K; R N R⟩ } inherits from a standard (or incomplete) class and, apart from adding new fields and adding/overriding methods, it can declare some methods as “incomplete”; there are two kinds of incomplete methods:

• “abstract” methods: the incomplete class declares only the signature of these “expected” methods; the body of these methods must be provided during object composition;

• “redefining” methods: although the body of these methods is provided by the incomplete class, they are still incomplete since the next special variable will be bound during object composition. We call these methods “redefining” because they will be the active part in the redefinition when an incomplete object (of an incomplete class) will be composed with a complete object. We then call the corresponding overridden methods of the complete object “redefined”.

Standard classes cannot inherit from incomplete classes (this is checked by typing, Section 2.3). The main idea of our language is that an incomplete class can be instantiated, leading to incomplete objects. Method invocation and field selection cannot be performed on incomplete objects.

An incomplete object expression e1 can be composed at runtime with a complete object expression e2; this operation, denoted by e1 ←→ e2, is called object composition. The key idea is that e1 can be composed with a complete object e2 that provides all the requested methods, independently from the class of e2 (of course, the method signatures must match). Then, in e1 ←→ e2, e1 must be an incomplete object and e2 must be a complete object expression (these requirements are checked by the type system); indeed, e2 can be, in turn, the result of another object composition. The object expression e1 ← e2 represents a brand new (complete) object that consists of the sub-object expressions e1 and e2; in particular, the objects of these sub-expressions are not modified during the composition. This also highlights the roles of incomplete and complete objects as re-usable building blocks for new objects at runtime, while retaining their identity and state.

We note that in this basic version of the language we do not allow object composition operations leading to incomplete objects, i.e., incomplete objects can only be fully completed. However, for instance, object compositions of the shape (e1 ←→ e2) ←→ e3, where e3 is incomplete in the methods provided by e3, can be obtained as e1 ←→ (e2 ←→ e3) in FJ. Furthermore, we prohibit the object composition between two complete objects; the semantics and the type system can be extended in order to deal with such an operation in a type safe way, but we prefer to keep the core calculus and its formal theory simple in this presentation.

Finally, values, denoted by v and u, are fully evaluated object creation terms. The object representation of IFJ is different from FJ in that fully evaluated objects can be also compositions of many objects. Thus, objects are represented as lists of terms new C(v) (i.e., expressions that are passed to the constructor are values too). For instance, new C(v) :: new D(t) :: v represents the composition of the incomplete object of class C with a standard complete object of class D (ε denotes the empty list). During method invocation, this list is scanned starting from the leftmost object in search for the called method (of course, in a well-typed program this search will terminate successfully). However, in order to implement delegation in a type safe way, we need to keep the position in the list where we found the method called. Technically, this is implemented by using a pair of object lists: the first one will be the object list scanned during method invocation, and the second one will be the entire composed object, i.e., the complete list. The basic idea is to use the complete list when binding this for redefined methods, and use the current position of the scanned list when binding this for methods that are not redefined (this solves the problem of accidental name clashes) and for binding next in redefining methods. This run-time representation of objects will be further explained when presenting the operational semantics of the calculus in Section 2.4.

As in FJ, a class table CT is a mapping from class names to class declarations. Then a program is a pair (CT, ε) of a class table (containing all the class definitions of the program) and an expression ε (the program’s main entry point). The class Object has no members and its declaration does not appear in CT. We assume that CT satisfies some usual 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 extends relation. Thus, in the following, instead of writing CT[C] = class . . . we will simply write class C . . .

### 2.1 Subtyping

In the type system we will need to distinguish between the type of an incomplete object and the type of a composed object (i.e., an incomplete object that has been composed with a complete object). If C is the class name of an incomplete object, then (C) denotes the type of an incomplete object of class C that has been composed. To treat complete and incomplete objects uniformly, we will use T to refer both to C and (C). However, types of the shape (C) are only used by the type system for keeping track of objects that are created via object composition. Indeed, the programmer cannot write (C) explicitly, i.e., T cannot be used in arrow types nor for declaring method parameters; this is consistent with Java-like languages'
philosophy where the class names are the only types that can be mentioned in the program (apart from basic types).

The subtype relation \( \ll \) (defined for any class table \( CT \)) on class types is induced by the standard subclass relation extended in order to relate incomplete objects (Figure 2). First of all, we consider an incomplete class \( C \) abstracts \( D \) \( \{ \ldots \} \); if \( D \) is a standard class, since \( C \) can make some methods of \( D \) incomplete, then it is obvious that an incomplete object of class \( C \) cannot be used in place of an object of class \( D \). Thus, \( \text{abstracts} \) implements subclassing without subtyping. Instead, when the incomplete object is composed with a complete object (providing all the methods requested by \( C \)), then its type is \( \ll \) (\( C \)), and it can be used in place of an object of class \( D \) (see the fourth rule). Since, as said above, we do not permit object composition on a complete object, then a complete object can never be used in place of an incomplete one. We could introduce subtyping between incomplete objects: this would require checking that the subtype does not have more incomplete methods than the supertype (contra-variance on requirements). To keep the presentation simple, however, in this first version we are not considering subtyping on incomplete objects. Instead, subtyping holds on their completed versions (last rule).

2.2 Programming Examples

In this section, we show how incomplete objects, and object composition, can be used to implement some recurrent programming scenarios. For simplicity, we will use here the full Java syntax (consider all methods as public) and we will denote object composition operation with \( \ll \).

We consider a scenario where it is useful to add some functionality to existing objects. Let us consider the development of an application that uses widgets such as graphical buttons, menus, and keyboard shortcuts. These widgets are usually associated to an event listener (e.g., a callback function), that is invoked when the user sends an event to that specific widget (e.g., one clicks the button with the mouse or chooses a menu item).

The design pattern command [15] is useful for implementing these scenarios, since it permits parametrization of widgets over the event handlers, and the same event handler can be reused for similar widgets (e.g., the handler for the event “save file” can be associated with a button, a menu item, or a keyboard shortcut). Thus, they delegate to this object the actual implementation of the action semantics, while the action widget itself abstracts from it. This decouples the action visual representation from the action controller implementation.

We can implement directly this scenario with incomplete objects, as shown in Listing 1: the class \text{Action} and \text{SaveActionDelegate} are standard Java classes (note that they’re not related). The former is a generic implementation of an action, and the latter implements the code for saving a file. We then have three incomplete classes implementing a button, a menu item, and a keyboard accelerator; note that these classes inherit from \text{Action}, make the method \text{run} incomplete, override the method \text{display} and redefine the method \text{enable}. Note that \text{display} is overridden in the classical inheritance sense, while \text{enable} is intended to be redefined at run-time, during object composition, i.e., it is a redefining method.

We also assume a class \text{Frame} representing an application frame where we can set keyboard accelerators, menu items, and toolbar buttons. An instance of class \text{Button} is an incomplete object (it requires the method \text{run} and \text{enable}) and, as such, we cannot pass it to \text{addToToolbar}, since \text{Button} \( \ll \) \text{Action} (subclassing without subtyping).

However, once we composed such an instance (through object composition operation, \( \ll \)) with an instance of \text{SaveActionDelegate}, then we have a completed object (of type \text{Button}) that can be passed to \text{addToToolbar} (since \text{Button} \( \ll \) \text{Action}). Note that we compose \text{Button} with an instance of \text{SaveActionDelegate} which provides the requested methods \text{run} and \text{enable}, although \text{SaveActionDelegate} is not related to \text{Action}. Furthermore, we can use the same instance of \text{SaveActionDelegate} for the other incomplete objects.

We now concentrate on the dynamic redefinition of \text{enable}; this method is used to enable/disable the graphical widget (e.g., buttons and menu items can be shaded when disabled) and also actions (when a document is saved, the action can be disabled until the document is modified). When \text{run} is executed in \text{SaveActionDelegate} the method also invokes \text{enable}. In a composed object, since we implement delegation, it is guaranteed that the redefining version of the method will be called (note, however, that the redefining versions will also call \text{enable} on next).

We now investigate some possible manual implementations in Java of this scenario, showing that our proposal is not simply syntactic sugar. We could write \text{Button} class with a field, say \text{deleg}, on which we call the method \text{run}. This approach requires \text{deleg} to be declared with a class or interface that provides such a method, say \text{Runnable}. However, this solution would not be as flexible as our incomplete objects, since one can then assign to \text{deleg} only objects belonging to the \text{Runnable} hierarchy.

On the other hand, if we wanted to keep the flexibility, we should declare \text{deleg} of type \text{Object}, and then call the method \text{run} by using Java Reflection APIs, (e.g., \text{getMethod}); however, this solution is not type safe, since exceptions can be thrown at run-time due to missing methods.

There are situations when one needs to add functionalities to an object dynamically; the design pattern decorator [15] is typically used to deal with these scenarios: at run-time an object (called \text{component}) is embedded in another object (decorator) that associates to the component additional features (by relying also on the implementation of the component). Since a decorator is a component itself, this can be used to create a chain of decorators.

Typically, stream libraries are implemented using this pattern: a stream class provides the basic functionalities for reading and writing bytes; then there are several specializations of streams (e.g., streams for compression, for buffering, etc.) that are composed in a chain of streams. The actual composition is done at run-time.

Although this pattern is useful in practice, it still requires manual programming. With method redefinition we could easily implement a stream library, as sketched in Listing 2: the specific stream specializations rely on the methods provided during object composition (using \text{next}) and redefine them. In order to show how delegation is implemented in our language, we introduced also the method \text{readBuffer} both in \text{CompressStream} and in \text{BufferedStream}. These two methods, in spite of having the same name, are completely unrelated (we also used different signatures). The operational semantics (Section 2.4) guarantees that the right implementation will be invoked, depending on the context in which this method is invoked; for instance, if a method in \text{BufferedStream} invokes \text{readBuffer}, then the version defined in \text{BufferedStream} will be selected (thus run-time type errors are avoided).

This example also shows the two different designs in modeling our stream framework: \text{FileStream} is not modeled as an incomplete class since it can be implemented with all the functionalities; on the contrary \text{CompressStream} and \text{BufferedStream} rely on another stream and thus they are incomplete classes (and their methods are redefining). It is then clear that \text{CompressStream}...
Moreover, note that \( C \) in incomplete methods of \( D \) is made empty; this reflects the fact that now all the methods of \( D \) that are not made incomplete by \( C \) returns a signature where the first element \( \{\} \) b) \{ \} \)

We define auxiliary functions (see Figure 3) to lookup fields and methods from \( CT \); these functions are used in the typing rules and in the operational semantics.

A signature set, denoted by \( S \), is a set of method signatures of the shape \( \mathbb{C} \rightarrow \mathbb{C} \). The signature of a class \( C \), denoted by \( \text{sign}(C) \), is a pair of signature sets \( \langle S_1, S_2 \rangle \), where the first set is the signature set of the complete methods and the second set is the signature set of the incomplete methods (both abstract and redefining). Of course, for standard classes, the second set will be empty.

The lookup function \( \text{fields}(C) \) returns the sequence of the field names, together with the corresponding types, for all the fields declared in \( C \) and in its subclasses. The \( \text{mtype}(m, C) \) lookup function (where \( m \) is the method name we are looking for, and \( C \) is the class where we are performing the lookup) differs from the one of \( FJ \) in that it relies on the new lookup function \( \text{sign} \); the lookup function \( \text{sign}(C) \) returns the signature of the class \( C \) by inspecting the signatures of its methods. In particular, since the superclass \( D \) of an incomplete class \( C \) can be in turn an incomplete class, the methods that are complete are those defined in \( C \) and those defined in \( D \) that are not made incomplete by \( C \) (i.e., \( \text{sign}(\mathbb{C}) \cup \langle S_1 - (\text{sign}(\mathbb{F}) \cup \text{sign}(\mathbb{R})) \rangle \); conversely, the incomplete methods are the incomplete methods of \( C \) and those of \( D \) that are not defined in \( C \) (i.e., \( \text{sign}(\mathbb{F}) \cup \text{sign}(\mathbb{R}) \cup \langle S_2 - \text{sign}(\mathbb{F}) \rangle \)). Moreover, for a composed object of type \( \langle C \rangle \), \( \text{sign} \) returns a signature where the first element is the union of the signature sets of its class and the second element is made empty; this reflects the fact that now all the methods of the object are concrete. Since we introduced this lookup function, the definition of \( \text{mtype} \) is straightforward (w.r.t. the one of \( FJ \) [17]). Moreover, note that \( \text{mtype} \) is defined for both \( C \) and \( \langle C \rangle \) and also for a method signature \( S \); the last case is for handling the \( \text{next} \) variable, whose type is a signature (see later in this section). Since \( \text{mtype} \) is the only lookup function defined on a signature set, it is not po-
possible to perform field selection on \texttt{next} (therefore, this will not be checked in the typing rules), since we do not express any requirements on fields. The lookup function for method bodies, \texttt{mbody}, is basically the same of FJ extended to deal with incomplete classes (note that it returns an empty element \texttt{*} for abstract methods).

A type judgment of the form $\Gamma \vdash e : T$ states that \texttt{e} has type \texttt{T} in the type environment \texttt{Γ}. A type environment is a finite mapping from variables (including this) to types, written \texttt{ξ : T}. Again, we use the sequence notation for abbreviating $\Gamma \vdash e_1 : T_1, \ldots, \Gamma \vdash e_n : T_n \Rightarrow \Gamma \vdash \textbf{e} : \textbf{T}$. In order to treat the \texttt{next} special variable uniformly, we extend the set of types with signature sets (i.e., \texttt{T} ranges over class names, completed types and signature sets).

Typing rules (Figure 4) are adapted from those of FJ in order to handle incomplete objects and object composition. In particular, field selection and method selection are allowed only on objects of concrete types, where a concrete type is either a standard class \texttt{C} or \texttt{(C)}. The key rule (T-Comp) for dealing with object composition is introduced. It checks that the left expression is actually an incomplete object ($S_2 \neq \emptyset$), and that the right one is a complete object that provides all the methods needed by the incomplete object. Note that the final type is the concrete type based on the original class of the incomplete object (we could have chosen the final type to be a structural combination of the types of the objects taking part in the composition, but our design choice is more suited to a nominal setting). This rule also shows that the typing of \texttt{←} is structural, which is a key feature of the system, since it enhances the flexibility of object composition.

Also typing rules for methods and classes of FJ are adapted to deal with incomplete classes (we use the \texttt{override} predicate of \texttt{[21]} to check that the signature of a method is preserved by method overriding). Note that, in order to type a \texttt{redef} method, we also need to assume a type for \texttt{next} when typing its body; it is safe to assume the signature set $S_2$, i.e., the signature set of incomplete methods. This is consistent with the way \texttt{next} is bound in the operational semantic rule for redefined method invocation (see Section 2.4). As noted before, thanks to the way lookup functions are defined (Figure 3), the only operation that is possible on next is method invocation.

Moreover, (T-CLASS) checks that a concrete class extends another concrete class and (T-ACLASS) checks that also the signatures of incomplete methods satisfy the \texttt{override} predicate. Typing rules for run-time expressions are in Figure 5; note that the type of

$$\text{Concrete predicate} \quad \text{sign(C)} = \langle S, \emptyset \rangle \quad \text{Concrete(C)}$$

**Expression typing**

$\Gamma \vdash e : T$

$\text{fields(T)} = \{ \text{f} \}$

$\text{concrete(T)}$

$\Gamma \vdash e : T$

$\text{mtype(m,T)} = B \rightarrow \emptyset$

$\text{concrete(T)}$

$\Gamma \vdash e : m : B$

$\text{new C(m,T)} : C$

$\Gamma \vdash e : C$

$\text{sign(C)} = \langle S_1, S_2 \rangle$

$\text{S_2} \neq \emptyset$

$\Gamma \vdash e : T$

$\text{sign(T)} = \langle S_1, S_2 \rangle$

$\text{mbody(m,T)} = \text{mtype(m,S)}$
a composed object is the one of the head of the list, consistently with the typing rule for object composition.

### 2.4 Operational semantics

The operational semantics, shown in Figure 6, is defined by the reduction relation $e \rightarrow e'$, read “$e$ reduces to $e'$ in one step”. The standard reflexive and transitive closure of $\rightarrow$ defines the reduction relation in many steps. We adopt a deterministic call-by-value semantics, analogous to the call-by-value strategy of FJ [21]. The congruence rules formalize how operators (method invocation, object creation, object composition, and field selection) are reduced only when all their subexpressions are reduced to values (call-by-value).

As already discussed, we need to annotate method invocation expressions with the (arrow) type of the method used during the static type checking; thus, in case of methods with the same name but different signatures within a composed object, we will not risk invoking the wrong version (generating a run-time type error).

Thus, the operational semantics is defined on annotated programs, i.e., IFJ programs where all expressions (including class method bodies) are annotated using the annotation function $\mathcal{A}$. Since this function relies on the static type system, it is parametrized over a type environment $\Gamma$.

**Definition 2.1 (Annotation Function).** The annotation of a new $e$, denoted by $\mathcal{A}[e]$, is defined on the syntax of $e$, by case analysis:

- $\mathcal{A}[(x)] = x$
- $\mathcal{A}[(e.f)] = \mathcal{A}[e].f$
- $\mathcal{A}[(m(\mathcal{A}[e])] = \mathcal{A}[e].m(\mathcal{A}[e])$

if $\Gamma \vdash c : C$ and $\text{mtype}(m,c) = B \rightarrow B$

- $\mathcal{A}[(\text{new } C(e))] = \text{new } C(\mathcal{A}[e])$

Given a method definition $B \rightarrow (\text{return } e)$, in a class $C$, the annotation of the method body $e$, is defined as $\mathcal{A}[e] : B \rightarrow \text{this}

Given a method redefinition $\text{redef } B \rightarrow (\text{return } e)$, in a class $C$, the annotation of the method body $e$, is defined as $\mathcal{A}[e] : B \rightarrow \text{this}_C \rightarrow C$.

In a real implementation, such annotation would be performed directly during the compilation, i.e., during the type checking. However, in the formal presentation, separating the two phases (type checking and annotation) makes the theory simpler.

In the following we will use $e$ (and $\mathcal{A}[e]$) also for annotated expressions where not ambiguous.

In order to represent run-time objects we use lists of standard FJ objects, of the shape $\text{new } C(t)$; moreover, in order to treat composed objects and standard objects uniformly, we represent a standard object with a list of only one element, $\text{new } C(t) : e$. The main idea of the semantics of method invocation is to search for the method definition in the (class of the) head of the list using the $\text{mbody}$ lookup function. If this is found, by rule (R-INVK), then the method body is executed; otherwise, by rule (R-DINVK), the search continues on the following element of the list (of course, in a well-typed program, this search will succeed eventually). However, in order to implement delegation, we need also to keep the original complete composed object so that we can bind this correctly; this is the reason why we represent a run-time object as a pair or two object lists: the first one is used for searching for the method, while the second one is the entire composed object. In the following we show how to perform the binding of this in the method body correctly. The expression $[\text{this} : \text{new } C(t)] e$ denotes the expression obtained from $e$ by replacing $x_1$ with $x_1$. . . . $x_n$ with $u$, and this with $\text{new } C(t) : 1$. 1.1 using the substitution of Definition 2.3. For redefining methods we also replace $\text{next}$, using the standard replacement (rule (R-RINVK)). The following formal definitions will be explained later.

**Definition 2.2 (findredef).**

- $\text{findredef}(m,e,B \rightarrow B) = \emptyset$
- $\text{findredef}(m,\text{new } C(t) : 1,B \rightarrow B) = \{\text{new } C(t) : 1, B \rightarrow B = \text{mtype}(m,c) \quad \text{findredef}(m,1,B \rightarrow B) \quad \text{otherwise}\}$
We conclude this section by considering possible alternative semantics to the one provided. The key point of our approach is that, when objects are composed, the resulting object consists of a list of sub-objects; in particular these sub-objects are not modified. Thus, the state and the identity of the objects within an object composition never change and the object composition produces a brand new object. Each object composition creates indeed a brand new object: for instance, each object composition would get a new object identifier in an imperative model. This shows that objects are not only instances of classes (possibly incomplete classes), but they are also prototypes that can be used, via the object composition, to create new objects at run-time (as in a prototype-based language like Self [24]), while ensuring statically that the composition is type safe. We then can use incomplete and complete objects as our re-usable building blocks to assemble at run-time, on the fly, brand new objects.

In an imperative setting, this mechanism will assure that there will not be problems when an object is pointed to by references in different parts of the program. On the contrary, if we modified the objects directly during the object composition, what would happen to the existing references? Surely they would all refer to the modified objects, but we would create situations where a reference of an incomplete class type would point to a complete object. This would break our discipline of not composing already complete objects with other complete objects; but, most of all, it would undermine the type safety, in fact there is no subtyping between an incomplete class and a complete class. However, if one is willing to relax this discipline, and introduce such a subtyping, composition between two complete objects could still be performed in a type safe way, thanks to the technique of annotation and the this substitution presented here (this was not possible in the first proposal of incomplete objects of [5]).

3. PROPERTIES

The language IFJ enjoys the type safety property, thus no "message-not-understood" errors can occur at run-time. In this section, we sketch the proof of this property by showing the main steps that are related to object composition. Namely, in the formal proofs, we explicitly deal with crucial points involving typing and semantics rules for incomplete objects, while we omit the parts that are similar or unchanged with respect to the corresponding proofs in FJ.

First of all, we give the typing rules for annotated run-time expressions; these rules are basically the same as the ones presented in Figure 4; the only rule that needs to be adapted is the one for method invocation, which is straightforwardly extended as follows:

\[
\begin{aligned}
\Gamma \vdash e : T & & \Gamma \vdash e_1 : T & & \Gamma \vdash e_2 : T & & \text{concrete}(T) \\
\end{aligned}
\]

By the definition of annotation (Definition 2.1), it is also trivial to prove the following property.

PROPERTY 3.1 (ANNOTATION PRESERVES TYPING).
If \( \Gamma \vdash e : T \) for some \( \Gamma \) and \( T \), then \( \Gamma \vdash \overline{e} : T \).

Thus, in the following, to make the presentation of the properties simpler, we will not write the annotations explicitly.

The following key lemmas state some crucial properties about the signatures of classes \( C_1, \ldots, C_n \) when considering values of the shape:

\[
v = \langle \text{new } C_i(\overline{v}_i) : \ldots : \text{new } C_0(\overline{v}_0) : \varepsilon, 1 \rangle
\]
Indeed, these properties are proved for the expression

\[ e = \text{new } C_1(\overline{v}_1) \leftarrow \ldots \leftarrow \text{new } C_n(\overline{v}_n) \]

using typing and semantics rules. Then, the properties are inherited by \( \overline{v} \) since \( \overline{v} \) can only be obtained by applying rule (R-COMP) to \( e \) (Figure 6); it is clear that the application of (R-COMP) does not affect signatures of \( C_1, \ldots, C_n \).

**Lemma 3.2.** Let \( 1 = \text{new } C(\overline{v}) :: \Gamma' \) such that \( \Gamma' \neq \varepsilon \) and \( \Gamma \vdash 1 : T \) where \( \text{concrete}(T) \). Then:

1. \( \Gamma \vdash \Gamma' : T' \) for some \( T' \) such that \( \text{concrete}(T') \)
2. If \( \text{sign}(C) = (S_1, S_2) \), for any \( m \) such that \( m : B \rightarrow B \) \( \in S_2 \) then \( \text{mtype}(m, T') = B \rightarrow B \)

**Proof.** 1. By (T-LIST) and (T-COMP).

2. By (T-COMP).

\[ \square \]

**Lemma 3.3.** If \( \text{mbody}(m, T) = \text{mbody}(m, T') \) implies that there exists some \( C_i, 1 \leq i \leq n \), such that \( m : B \rightarrow B \in S_i \).

**Theorem 3.5.** (Type Preservation). If \( \Gamma \vdash e : T \) and \( e \rightarrow e' \) then \( \Gamma \vdash e' : T' \) for some \( T' \rightarrow T \).

**Proof.** By induction on a derivation of \( e \rightarrow e' \). The only interesting cases are:

- (R-COMP): By (T-COMP) and (T-LIST), the type is preserved after reduction.
- (R-INVK): By (T-INVK) and Lemma 3.3-2, using Substitution Lemma 3.4.
- (R-DINVK): By Lemma 3.2.

\[ \square \]

**Theorem 3.6.** (Progress). Let \( e \) be a closed expression. If \( \vdash e : T \) for some \( T \), then either \( e \) is a value or \( e \rightarrow e' \) for some \( e' \).

**Proof.** By induction on \( \vdash e : T \); the crucial case is:

- (T-INVK): the method invocation can be reduced by (R-INVK), (R-RINVK) or by (R-DINVK); Lemma 3.3-1 is the key argument to guarantee that the search for the method body will eventually succeed (i.e., (R-INVK) or (R-RINVK) will be applied after some applications of (R-DINVK)).

\[ \square \]

Theorems 3.5 and 3.6 show how type safety of FJ is preserved when adding incomplete objects, redefining methods and delegation with our approach, i.e., any well-typed FJ program cannot get stuck.

## 4. CONCLUSIONS AND RELATED WORK

In this paper we presented a linguistic construct to deal with incomplete objects, object composition and delegation, combining static type discipline class-based features with the flexibility of object-based ones, in a type safe way. The implementation of incomplete objects in Java is currently under development. The functionalities provided by the dynamic method redefinition, together with the delegation mechanism, further unleash the flexibility of incomplete objects (compared to the first proposal of [5]) and pose some interesting technical challenges that we solved in a pragmatic way (easily implementable in a Java-like language). For instance, object composition and delegation introduce the “width subtyping versus method addition” problem that is well known in the object-based setting (see, e.g., [14]). We solve this issue by representing objects as lists of subobjects in such a way that we can explicitly deal with the “scope” of a method invocation; we believe this solution is much more implementation-oriented than the dictionaries of [22] and simpler than the one of [4].

We conclude the paper by discussing some related work in the literature. Concerning the theory of incomplete objects, our main inspiration comes from [4]; however, while that calculus builds on top of the lambda calculus, here we aim at proving how object composition can fit within the basic principles of Java-like languages.

An explicit form of incomplete objects was introduced in [7], where an extension of Lambda Calculus of Objects of [13] is presented. In this work, “labelled” types are used to collect information on the mutual dependencies among methods, enabling a safe subtyping in width. Labels are also used to implement the notion of completion, which permits adding methods in an arbitrary order allowing the typing of methods that refer to methods not yet present in the object, thus supporting a form of incomplete objects. The context is again a lambda calculus, while in this work we are interested in incorporating object composition into Java-like languages.

Incomplete object mechanisms were originally inspired by mixin-based inheritance [8]; mixins are classes parameterized over the superclass and new subclasses can be generated by applying a mixin to a class (that provides all the requirements of the mixin). However, object composition in our language takes place at run-time, while mixin inheritance, although more flexible than standard class-based inheritance, is still a compile-time mechanism.

In [18], delegation is presented in the model of the language Darwin; however, this model requires some basic notions to be modified, such as method overriding. Our language, instead, proposes a conservative extension of a Java-like language (so that existing code needs not to be changed). Furthermore, in [18] the type of the parent object must be a declared class and this limits the flexibility of dynamic composition, while in our approach there is no implicit parent and missing methods can be provided by any complete object, independently from its class.
Incomplete objects can be seen as wrappers for the objects used in object composition. However, they differ from decorator-based solutions such as the language extension presented in [6]: incomplete objects provide a more general-purpose language construct and the wrappers of [6] could be actually implemented through incomplete objects.

Objective-C [19] provides categories, a run-time mechanism for modifying existing objects: the programmer can place groups of related methods into a category and can add the methods within a category to a class at run-time. Thus, categories allow the programmer to add methods to an existing class without the need to recompile that class or even have access to its source code. The main difference with our incomplete object mechanism is that categories act at the class level, while our linguistic feature acts at the object level.

Traits [11] are composable units containing only methods, and they were proposed as an add-on to traditional class-based inheritance in order to enhance decoupling and high cohesion of code in classes, therefore with the aim of allowing a higher degree of code reuse. Incomplete objects can be seen as a tool for rapid prototyping, that is, for adding methods on the fly to already existing objects. Traits and incomplete objects share an important feature, composition, which permits composing sets of methods “at the right level”, for instance not too high in a hierarchy for traits, and “when needed” for incomplete objects. The main difference is that traits are a compile-time feature, while incomplete objects are composed at run-time. An issue to pursue as a further research may be the use of incomplete objects as an exploratory tool to design traits: experiments made at run-time without modifying a class hierarchy might give indications on where to put a method in a new version of the hierarchy.

There are some relations between aspects [10] and our incomplete objects. Both are used to combine features taken from different sources. In the aspect case, the main idea is to factorize into aspects some cross-cutting functions (such as logging services or concurrency primitives) that are needed globally by a library, instead of duplicating and scattering them into the business code. In our case, we consider objects as building blocks that can be used to combine features on the fly, in order to obtain and experiment with multi-function objects whenever it is desired. In a sense, the role of incomplete objects is orthogonal to the one of aspects, because the former play a local role, while the latter a more global one.

In [2], a general model (Method Driven Model) for languages supporting object composition is proposed: this is based on the design of classes in an aspect-oriented style. The authors do not formalize their model within a calculus, but it is possible to see that the main feature of a language based on this model would be to compose dynamically the overall behavior of an object from the multiple “aspects” that abstract the variant behavior, as discussed in [3]. The main difference between their proposal and ours is that for the run-time behavior is codified in aspects, whereas we internalize it in Java by exploiting partial classes and object composition.

The language *gheta* [12] supports a mechanism called “object metamorphosis”, which is a mechanism to specialize dynamically an existing object, by applying to it a class as a constraint in such a way the object becomes an instance of that class. The main difference between the *gheta* specializing objects and our incomplete objects is that the former maintain the object identity, while the latter are used to create dynamically new objects which are not instances of any classes present in the program. Both proposals are proved type-safe, but a more direct comparison is not straightforward, as the type system of *gheta* exploits concepts such as virtual classes which are not present in a Java-like setting like ours. It is important to remark that one of our main design decision was that our extension must integrate seamlessly in a Java-like language as a conservative extension.

**Acknowledgments.** We warmly thank Betti Venneri for her work on the first version of IFJ. Her contribution made its way in this version, too. We also thank the anonymous referees for useful advices on how to improve the paper.

5. REFERENCES


[11] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. ACM Transactions on Programming Languages and Systems, 28(2):331–388, 2006.


[16] A. Goldberg and D. Robson. Smalltalk 80: The Language. Addison-Wesley, 1989.

[17] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GI. ACM Transactions on Programming Languages and Systems, 23(3):396–450, 2001.


[23] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. ACM Transactions on Programming Languages and Systems, 28(2):331–388, 2006.


[27] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[28] A. Goldberg and D. Robson. Smalltalk 80: The Language. Addison-Wesley, 1989.

[29] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GI. ACM Transactions on Programming Languages and Systems, 23(3):396–450, 2001.


