Object Reuse and Behavior Adaptation in Java-like Languages [*]

Lorenzo Bettini
Dip. di Informatica, Università di Torino,
Corso Svizzera 185, 10149 Torino, Italy
bettini@di.unito.it

Betti Venneri
Dip. di Sistemi e Informatica, Università di Firenze
Viale Morgagni 65, 50134 Firenze, Italy
venneri@dsi.unifi.it

Abstract

Inheritance, which is a basic mechanism in mainstream object-oriented languages, introduces a strong coupling which limits modularity and code reuse. Furthermore, static class hierarchies cannot easily deal with unpredictable dynamic adaptations of the object behavior. In order to overcome these limitations, we propose new linguistic constructs for composing objects in a Java-like language. Objects are intended as featherweight components which can be used in multiple compositions, and object types specify not only the implemented functionalities, but also the required methods, which will be provided by other components during composition. Thus the language supports flexible object reuse and adaptation of the object behavior at run time. The static type discipline guarantees that method calls on well-typed object compositions are safe.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.1.5 [*Programming Techniques*]: Object-oriented Programming

General Terms Languages, Theory

Keywords Featherweight Java, Object Composition, Delegation, Type systems

1. Introduction

Class-based object-oriented languages provide inheritance and method overriding, with late binding, to account for code reuse and dynamic specialization, respectively. However, these mechanisms do not suffice for modeling situations when objects have to modify their behavior according to conditions known only at run time. Indeed, inheritance introduces a strong coupling between classes, which limits modularity and safe reuse (we refer to [14, 29], and to the references therein, for an insightful review of the limitations of inheritance).

To overcome these limitations, a number of *design patterns* were proposed in [17] that rely on *object composition*. The alternative programming style, proposed by these patterns, consists in writing small software components (units of reuse), that can be composed in several ways at run time. This leads, for instance, to the use of *wrappers* (see, e.g., [7, 10]), that are interposed between the objects and their clients in order to implement new behavior. Besides the fact that design patterns require manual programming, which is prone to errors, these composition-based solutions are limited by the underlying model of object composition and method forwarding which is intrinsic to class-based languages. Namely, when an object forwards a message to its component, to achieve a new functionality, the *self* variable results to be bound to the component, then the successive method calls are applied to the component, not to the whole object. This is known as the *self problem* and the method forwarding above is called *consultation* (we refer to [22] for an insightful discussion on the self problem and the limits of consultation). With consultation we lose the *transparent redirection* [27]; when we manually implement object composition and method forwarding we cannot achieve a real dynamic object inheritance and dynamic method redefinition. For instance, when implementing the *decorator* pattern [17], we will surely experience the anomaly of the *self problem* [24], i.e., *broken delegation* [18].

As opposite to consultation, *delegation*, originally introduced by Lieberman [24], seems to be the suitable method call mechanism to reuse code and achieve dynamic method redefinition (see, e.g., [11, 16, 30]). By delegation, when A forwards to B the execution of a message, `this` is bound to the sender A, in such a way that any successive method call will be executed on A.

In this paper we address the above issues from a foundational point of view by proposing a core language, called CompObJ (*Composing Objects in Java-like languages*) which integrates new constructs for composing objects into a statically-typed class-based setting, using delegation for method calls: the proposed language aims at flexible code reuse and dynamic adaptation of the object behavior, while retaining static type checking and type safety.

The basic idea is to adopt a generalized notion of object, which is inspired by the component-based programming [32]. Objects are intended as featherweight components which can be used in different compositions to modify their behavior based on dynamic conditions and independently from inheritance hierarchies. Namely, an object type is intended as the specification of what the object provides and what it requires from other components. The composition of that object with other objects will provide the requested code so specifying the object behavior.

In CompObJ class definitions, besides standard method definitions, can declare also the required methods, namely the *expected* and the *redefining* methods. Expected methods are similar to traditional abstract methods: their bodies are missing and must be provided during object composition. Analogously, redefining methods are similar to method overriding in class-based inheritance: they redefine a method of another object inside a composition. Then, we call these methods "redefining", and we call the corresponding overridden versions "redefined". As in Java an overridden method

[*] This work has been partially funded by MIUR (PRIN 2008 *DISCO*) and by EU Collaborative project n. 257414 *Autonomic Service-Component Ensembles* (ASCENS).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PPPJ '11, August 24–26, 2011, Kongens Lyngby, Denmark.
Copyright © 2011 ACM 978-1-4503-0935-6. . . $10.00

can access the previous implementation by `super`, in a redefining method we can access the redefined version with the special variable `next` (`next` plays the role of a "horizontal" version of `super`). Summarizing, required methods formalize the variable part of the object behavior, which will be completely or partially provided during object composition. Classes with required methods are then *incomplete*, and their instances are called *incomplete objects*; standard classes (objects) of Java-like languages are represented by classes with no requested method.

The incomplete methods, expected and redefined, must be provided during object composition by other objects. Thus, 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. We see this as 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). In particular, some behavior that was not foreseen when the class hierarchy was implemented may be supplied dynamically by composition with existing objects, thus generating an unanticipated reuse of code and a sharing of components. Then, we can use incomplete and complete objects as our re-usable building blocks to assemble at run time, on the fly, brand new objects. 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).

One of our key design choices is to integrate object composition within the nominal subtyping mechanism of Java-like languages. However, in order to enhance run-time flexibility in composing objects, we implicitly use structural type checks during composition: an object can be composed with any object with some of the requested methods (provided the signatures match), no matter what its class is. Therefore, the language we propose is not a manual implementation of the object composition. In the case of a manual implementation, the object should be stored in a class field, thus forcing it to belong to a specific class hierarchy. Alternatively, one could use type `Object`, and then call methods using Java Reflection APIs or down-casts: this solution does not avoid possible run-time exceptions due to missing methods. Finally, implementing delegation manually would even be harder: it would require to modify all the methods in order to pass explicitly "another" `this`, i.e., the one bound to the original sender, in order to avoid the *self problem* [24] and *broken delegation* [18].

CompObJ relies on the approach that has been presented in [6]. However, there is a significant difference regarding the model of classes and objects. In [6] we had two separate hierarchies, standard classes (i.e., basically Java classes defined with the `extends` clause) and incomplete classes (with the `abstracts` clause for the superclass); for instance, standard classes could not inherit from incomplete classes. In CompObJ we unify the two concepts, in order to enhance the flexibility of classes and, in particular, of the objects. As a consequence, with CompObJ we overcome the restrictions that were in [6] concerning the shape of object compositions, the nature of objects involved in a composition, and in particular the subtyping among objects. In [6] we only allowed an incomplete object to be composed with a complete object and the latter had to provide all the methods required by the former; thus, object compositions could produce complete objects only. Instead, to add practicality to our approach, in CompObJ object composition can also produce incomplete objects and can compose two incomplete objects.

This new model poses some crucial issues to the type system. First of all, in our language, subclassing does not imply subtyping (indeed we use a different keyword for class inheritance, `inherits`): an incomplete class `C` which inherits from a complete class `D` (thus introducing some required methods) cannot be considered as a subtype of `D` since `C`'s instances are incomplete objects and cannot be seen as `D`'s instances (which are instead complete objects). Our types for objects formalize not only their original class, but also the requirements that are still to be fulfilled (via object composition). Thus, subtyping considers a type to be subtype of another one if the former has less requirements than the latter. However, to keep the nominal setting of Java-like languages, our subtype relation still relies on a nominal subclass relation while adding some structural checks: subclassing is a necessary condition for subtyping, but not a sufficient condition. Note that subtyping allows two objects to be compared independently from the way they are generated: each of them can be a standard object (an instance of a complete class), an instance of an incomplete class or a composed object (either complete or incomplete). This is a crucial difference between CompObJ and the type system of [6] and enhances the flexibility of object compositions where a component can be replaced by a "less incomplete" (possibly complete) object during the evaluation, by preserving the well-typedness.

Dealing with delegation, in particular in the presence of subtyping and object composition, requires an interesting technical treatment to achieve a type-safe implementation. For instance, we need to avoid possible name clashes for methods with the same name but with different signatures (possibly due to the subtyping [16]) 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 in another component). In order to deal with such problems, we employ a static annotation procedure (based on static types) which is used in the operational semantics (Section 4) to bind the self object `this` correctly in the method bodies.

The paper is organized as follows. Section 2 defines the core language CompObJ for object composition with delegation. Sections 3 and 4 present the type system and the operational semantics, respectively. Section 5 illustrates the application of our proposal to some programming scenarios. Section 6 discusses some related works and Section 7 concludes.

2. CompObJ

In this section we present the core language CompObJ (*Composing Objects in Java-like languages*). It is based on FJ (*Featherweight Java*), the minimal core calculus introduced in [21] for modeling Java typing; we only omit type casts which are irrelevant for our issues. The main features of CompObJ with respect to FJ can be summarized in the following points:

1. a class can declare not only the provided methods but also the required ones, then class instantiation can produce *incomplete objects*;
2. we introduce a new operator for *composing* objects dynamically, so manufacturing new objects where the actual implementation of (some of the) required methods is incrementally provided or specialized by the added components;
3. method call on composed objects is performed by *delegation*;
4. subclassing does not directly imply subtyping, even though subtyping still relies on a nominal type relation.

The abstract syntax of the CompObJ is defined in Figure 1 (grayed expressions are not written by the programmer, but are produced by the semantics, Section 4). We use the overline notation for possibly empty sequences (e.g., "$\overline{\mathtt{e}}$" is a shorthand for a sequence "$\mathtt{e}_1, \ldots, \mathtt{e}_n$"), and the empty sequence is denoted by $\bullet$.

In a class declaration $\mathtt{class\ C\ inherits\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\ \ \overline{\mathtt{N}}\ \ \overline{\mathtt{R}}\}$, `C` inherits from the superclass `D` (which must always be specified, even if it is `Object`) all fields and methods. Then `C` introduces a list of new fields $\overline{\mathtt{C}}\ \overline{\mathtt{f}}$ with their types (the fields of `C` are added to the

$\mathtt{A}$	::=	$\mathtt{class\ C\ inherits\ C}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\ \overline{\mathtt{N}}\ \overline{\mathtt{R}}\}$	classes
$\mathtt{M}$	::=	$\mathtt{C\ m}\ (\overline{\mathtt{C}}\ \overline{\mathtt{x}})\{\mathtt{return\ e;}\}$	methods
$\mathtt{N}$	::=	$\mathtt{C\ m}\ (\overline{\mathtt{C}}\ \overline{\mathtt{x}});$	expected methods
$\mathtt{R}$	::=	$\mathtt{redef\ C\ m}\ (\overline{\mathtt{C}}\ \overline{\mathtt{x}})\{\mathtt{return\ e;}\}$	redefining methods
$\mathtt{e}$	::=	$\mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m}(\overline{\mathtt{e}}) \mid \mathtt{new\ C}(\overline{\mathtt{e}}) \mid \mathtt{e} \longleftarrow\!\!\!+ \mathtt{e}$	expressions
$\mathtt{v}$	::=	$\langle \mathtt{l}_1, \mathtt{l}_2 \rangle$ where $\mathtt{l}_1 \subseteq \mathtt{l}_2$	values
$\mathtt{l}$	::=	$\mathtt{new\ C}(\overline{\mathtt{v}}) :: \varepsilon \mid \mathtt{new\ C}(\overline{\mathtt{v}}) :: \mathtt{l}$	run-time object list

Figure 1. CompObJ syntax; run-time syntax appears shaded. $\subseteq$ denotes list inclusion.

ones inherited by the superclasses and are assumed to have distinct names). Constructors are considered implicit with a fixed syntax: $\mathtt{C}(\overline{\mathtt{C}}\ \overline{\mathtt{f}})\{\mathtt{super}(\overline{\mathtt{f}});\ \mathtt{this}.\overline{\mathtt{f}} = \overline{\mathtt{f}};\}$. The lists $\overline{\mathtt{M}}$, $\overline{\mathtt{N}}$ and $\overline{\mathtt{R}}$ contain the methods which are declared by $\mathtt{C}$. A method definition $\mathtt{M}$ specifies the name, the signature and the body of the method, where the body is a single `return` statement. Instead $\overline{\mathtt{N}}$ and $\overline{\mathtt{R}}$ concern the methods which are declared as "required" (or "incomplete"). Namely:

- "expected" methods: the class declares only the signature of these methods, while their bodies are expected to 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 special variable `next` will be bound during object composition. We call these methods "redefining" because they will be the active part in the redefinition when objects are composed. We then call the corresponding overridden methods "redefined". For instance, in a redefining method `m` we can access the redefined version with `next.m()`.

We assume that method names in $\overline{\mathtt{M}}$, $\overline{\mathtt{N}}$ and $\overline{\mathtt{R}}$ are all distinct. However, the subclass can make "incomplete" a method which was complete in the superclass and can give a complete implementation for a method which was incomplete in the superclass.

As in FJ, we assume that 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. Analogously, the special variable `next` is implicitly bound in `redef` methods: we can see `next` as the dynamic (and horizontal) version of `super` (if `super` was added to FJ, it could safely coexist with `next`). Since we treat `this` and `next` in method bodies as ordinary variables, no special syntax for them is required.

If a class contains incomplete methods then it is considered itself as an *incomplete class*. Indeed, an incomplete class can be instantiated, leading to an *incomplete object*: method invocation cannot be performed on incomplete objects (the static typing will guarantee this property), while field selection can be safely performed on them. A standard FJ (Java) class is represented by a class which is completely defined, i.e., the sets $\overline{\mathtt{N}}$ and $\overline{\mathtt{R}}$ are empty, so instances of complete classes are (complete) standard objects.

An object expression $\mathtt{e}_1$ can be composed at run time with another object expression $\mathtt{e}_2$ (*object composition*). We point out that components of an object composition can be, in turn, the results of other object compositions. Furthermore, the result of an object composition can be still an incomplete object. When $\mathtt{e}_1$ is composed with $\mathtt{e}_2$, the latter can provide some of the methods required by the former, independently from the class (type) of $\mathtt{e}_2$ (of course, the method signatures must match in well-typed expressions). Then the expression $\mathtt{e}_1 \longleftarrow\!\!\!+ \mathtt{e}_2$ represents a brand new object that consists of the sub-object expressions $\mathtt{e}_1$ and $\mathtt{e}_2$; in particular, the objects of these sub-expressions are not modified during the composition. Thus objects retain their identity and state in all compositions they are part of, while their behavior can be, partially or completely, adapted in different compositions according to dynamic conditions.

$$\frac{\mathtt{class\ C\ inherits\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\overline{\mathtt{M}}\,\overline{\mathtt{N}}\,\overline{\mathtt{R}}\}}{\mathit{defined}(\mathtt{C}) = \mathit{sign}(\overline{\mathtt{M}}) \cup (\mathit{defined}(\mathtt{D}) - (\mathit{sign}(\overline{\mathtt{N}}) \cup \mathit{sign}(\overline{\mathtt{R}})))}$$

$$\frac{\mathtt{class\ C\ inherits\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\overline{\mathtt{M}}\,\overline{\mathtt{N}}\,\overline{\mathtt{R}}\}}{\mathit{required}(\mathtt{C}) = \mathit{sign}(\overline{\mathtt{N}}) \cup \mathit{sign}(\overline{\mathtt{R}}) \cup (\mathit{required}(\mathtt{D}) - \mathit{sign}(\overline{\mathtt{M}}))}$$

$$\mathit{interface}(\mathtt{C}) = \mathit{defined}(\mathtt{C}) \cup \mathit{required}(\mathtt{C})$$

Figure 2. *defined*, *required* and *interface* definitions.

Regarding class definitions, we observe a significant difference of the present approach with respect to [6]. In [6] we had two separate hierarchies, standard classes (i.e., basically Java classes defined with the `extends` clause) and incomplete classes (with the `abstracts` clause for the superclass); for instance, standard classes could not inherit from incomplete classes. In CompObJ we unify the two concepts, in order to enhance the flexibility of classes and, in particular, of the objects. As a consequence, we allow the object composition to produce both complete and incomplete objects.

For the run-time representation of objects, we use lists of terms $\mathtt{new\ C}(\overline{\mathtt{v}})$ (i.e., expressions that are passed to the constructor are values, too). For instance, $\mathtt{new\ C}(\overline{\mathtt{v}}) :: \mathtt{new\ D}(\overline{\mathtt{u}}) :: \varepsilon$ (ε denotes the empty list). The role of this representation will be clear in Section 4. Intuitively, during method invocation, this list is scanned starting from the leftmost object in search for the called method.

As in FJ, a class table *CT* is a mapping from class names to class declarations. Then a program is a pair $(\mathit{CT}, \mathtt{e})$ of a class table (containing all the class definitions of the program) and an expression `e` (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 the usual sanity conditions [21]. In the following, instead of writing $\mathit{CT}(\mathtt{C}) = \mathtt{class\ C}\ \ldots$ we will simply write $\mathtt{class\ C}\ \ldots$. Moreover, to simplify the notation, we will always assume a fixed class table *CT* (as in FJ).

2.1 Types

In order to define the typing rules and the lookup functions, we introduce some preliminary definitions. A *signature set*, denoted by `S`, is a set of method signatures of the shape $\mathtt{m} : \overline{\mathtt{C}} \to \mathtt{C}$. If

$$\overline{\mathtt{M}} = \overline{\mathtt{C\ m}\ (\overline{\mathtt{C}}\ \overline{\mathtt{x}})\{\mathtt{return\ e;}\}}$$

represents the sequence of method definitions

$$\mathtt{C}_1\ \mathtt{m}_1\ (\overline{\mathtt{C}}_1\ \overline{\mathtt{x}})\{\mathtt{return\ e}_1;\}\ \ldots\ \mathtt{C}_n\ \mathtt{m}_n\ (\overline{\mathtt{C}}_n\ \overline{\mathtt{x}})\{\mathtt{return\ e}_n;\}$$

then $\mathit{sign}(\overline{\mathtt{M}})$ will denote the set of signatures $\overline{\mathtt{m} : \overline{\mathtt{C}} \to \mathtt{C}}$. The same convention is used for redefining methods and for expected method definitions (and their corresponding signatures). The auxiliary functions (Figure 2) $\mathit{defined}(\mathtt{C})$ and $\mathit{required}(\mathtt{C})$ return the signatures of the methods which are defined and required in class `C`, respectively, by inspecting the signatures of its methods (and recursively its superclass). In particular, the methods that are complete are those defined in `C` and those defined in `D` that are not made incomplete by `C` (i.e., $\mathit{sign}(\overline{\mathtt{M}}) \cup (\mathit{defined}(\mathtt{D}) - (\mathit{sign}(\overline{\mathtt{N}}) \cup \mathit{sign}(\overline{\mathtt{R}})))$); conversely, the incomplete methods are the incomplete methods of `C` and those of `D` that are not defined in `C` (i.e., $\mathit{sign}(\overline{\mathtt{N}}) \cup \mathit{sign}(\overline{\mathtt{R}}) \cup (\mathit{required}(\mathtt{D}) - \mathit{sign}(\overline{\mathtt{M}}))$). The "interface" of a class is the set of all the signatures of the class. We observe that if `C inherits D` then $\mathit{interface}(\mathtt{D}) \subseteq \mathit{interface}(\mathtt{C})$.

The syntax of types, denoted by `T`, is the following:

$\mathtt{T}$	::=	$\mathtt{C}$	class type
	$\mid$	$\langle \mathtt{C}/\mathtt{S} \rangle$ where $\mathtt{S} \subseteq \mathit{interface}(\mathtt{C})$	object type
	$\mid$	$\mathtt{S}$	signature type

$$\mathtt{C} \lhd \mathtt{C} \qquad \frac{\mathtt{C}_1 \lhd \mathtt{C}_2 \qquad \mathtt{C}_2 \lhd \mathtt{C}_3}{\mathtt{C}_1 \lhd \mathtt{C}_3} \qquad \frac{\mathtt{class}\ \mathtt{C}\ \mathtt{inherits}\ \mathtt{D}\ \{\ldots\}}{\mathtt{C} \lhd \mathtt{D}}$$

$$\mathtt{T} <: \mathtt{T} \qquad \frac{\mathtt{T}_1 <: \mathtt{T}_2 \qquad \mathtt{T}_2 <: \mathtt{T}_3}{\mathtt{T}_1 <: \mathtt{T}_3}$$

$$\frac{\mathtt{C} \lhd \mathtt{D} \qquad \mathit{otype}(\mathtt{T}_1) = \langle \mathtt{C}/\mathtt{S}_1 \rangle \qquad \mathit{otype}(\mathtt{T}_2) = \langle \mathtt{D}/\mathtt{S}_2 \rangle \qquad \mathtt{S}_1 \subseteq \mathtt{S}_2}{\mathtt{T}_1 <: \mathtt{T}_2}$$

Figure 3. Subtyping rules.

The programmer can only write class names as types; this is consistent with Java-like languages' philosophy. On the other hand, *object types* $\langle \mathtt{C}/\mathtt{S} \rangle$ are used only by the type system. An object type consists of a class name `C` and of a signature set `S` which represents the methods that are still required by the object (either expected or redefining). Note that an object type is considered well-formed only if the signatures of required methods are included in the interface of `C`. Thus, an object is considered complete if its type is of the shape $\langle \mathtt{C}/\emptyset \rangle$. The signature type, represented by a signature set `S`, is used by the type system only for handling the `next` variable (see Section 3).

Then, given a type `T`, we define the auxiliary function *otype* which returns an object type:

$$\mathit{otype}(\mathtt{T}) = \begin{cases} \langle \mathtt{C}/\mathit{required}(\mathtt{C}) \rangle & \text{if } \mathtt{T} = \mathtt{C} \\ \mathtt{T} & \text{if } \mathtt{T} \text{ is an object type} \\ \bot & \text{otherwise} \end{cases}$$

Informally, the object type $\langle \mathtt{C}/\mathtt{S} \rangle$ represents the type of an object which is either an instance of `C` (then $\mathtt{S} = \mathit{required}(\mathtt{C})$) or an object composition starting with an instance of `C`. In the latter case `S` is the set of method signatures which are still required by the composed object (provided that $\mathtt{S} \subseteq \mathit{interface}(\mathtt{C})$). The role of object types will be clear in the next sections.

2.2 Subtyping

The subclass relation $\lhd$ (defined for any class table *CT*) on class types is induced by the `inherits` specification. However, in our language, the inheritance relation `inherits` does not directly imply subtyping, denoted by $<:$.

The subtype relation (Figure 3) is defined for each form of types `T` except for signature types (since in the premises of the last rule we use *otype*, which is undefined for signature types, we implicitly forbid subtyping on signature types). Subtyping rules formalize the situation when we can safely use an object of type $\langle \mathtt{C}/\mathtt{S}_1 \rangle$ when an object of type $\langle \mathtt{D}/\mathtt{S}_2 \rangle$ is expected. Namely, we control that the former has less requirements than the latter, and that `C` is a subclass of `D`. This means that our subtype relation still relies on a nominal subclass relation while adding some structural checks. The conjunction of the two conditions imply that a subtype provides at least all the defined methods and require less methods than the supertype. Note that subtyping allows two objects to be compared independently from the way they are generated: each of them can be a standard object (an instance of a complete class), an instance of an incomplete class or a composed object (either complete or incomplete). This is a major difference between CompObJ and the type system of [6] and enhances the flexibility of object compositions where a component can be replaced by a "less incomplete" (possibly complete) object during the evaluation.

2.3 Lookup functions

We define lookup functions (see Figure 4) to lookup fields and methods from *CT*; these functions are used in the typing rules and in the operational semantics.

The lookup function *fields*(`C`) returns the sequence of the field names, together with the corresponding types, for all the fields declared in `C` and in its superclasses. The $\mathit{mtype}(\mathtt{m},\mathtt{T})$ lookup function (where `m` is the method name we are looking for, and `T` is the type on which we are performing the lookup) returns the signature of `m` by inspecting a signature set: when `T` is a class type `C` the signature set is the interface of `C` (including the defined and required methods). The case for a signature type `S` is used for handling the `next` variable, whose type is a signature set (see Section 3). Since *mtype* is the only lookup function defined on a signature set, it is not possible to perform field selection on `next`. The lookup function for method bodies, *mbody*, is similar to FJ but it is extended to deal with incomplete classes (note that it returns an empty element $\bullet$ for expected methods).

$$\mathit{fields}(\mathtt{Object}) = \bullet \qquad \mathit{fields}(\langle \mathtt{C}/\emptyset \rangle) = \mathit{fields}(\mathtt{C})$$

$$\frac{\mathtt{class}\ \mathtt{C}\ \mathtt{inherits}\ \mathtt{D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\ \overline{\mathtt{N}}\ \overline{\mathtt{R}}\} \qquad \mathit{fields}(\mathtt{D}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}}}{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}}, \overline{\mathtt{C}}\ \overline{\mathtt{f}}}$$

$$\mathit{mtype}(\mathtt{m},\mathtt{C}) = \mathit{mtype}(\mathtt{m},\mathit{interface}(\mathtt{C}))$$

$$\mathit{mtype}(\mathtt{m},\langle \mathtt{C}/\ldots \rangle) = \mathit{mtype}(\mathtt{m},\mathtt{C})$$

$$\frac{\mathtt{m} : \overline{\mathtt{B}} \to \mathtt{B} \in \mathtt{S}}{\mathit{mtype}(\mathtt{m},\mathtt{S}) = \overline{\mathtt{B}} \to \mathtt{B}}$$

$$\frac{\begin{array}{c}\mathtt{class}\ \mathtt{C}\ \mathtt{inherits}\ \mathtt{D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\ \overline{\mathtt{N}}\ \overline{\mathtt{R}}\} \\ \mathtt{B}\ \mathtt{m}\ (\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\mathtt{return}\ \mathtt{e};\} \in \overline{\mathtt{M}} \text{ or } \mathtt{redef}\ \mathtt{B}\ \mathtt{m}\ (\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\mathtt{return}\ \mathtt{e};\} \in \overline{\mathtt{R}}\end{array}}{\mathit{mbody}(\mathtt{m},\mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e})}$$

$$\frac{\mathtt{class}\ \mathtt{C}\ \mathtt{inherits}\ \mathtt{D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\ \overline{\mathtt{N}}\ \overline{\mathtt{R}}\} \qquad \mathtt{m} \notin \overline{\mathtt{M}} \cup \overline{\mathtt{N}} \cup \overline{\mathtt{R}}}{\mathit{mbody}(\mathtt{m},\mathtt{C}) = \mathit{mbody}(\mathtt{m},\mathtt{D})}$$

$$\frac{\mathtt{class}\ \mathtt{C}\ \mathtt{inherits}\ \mathtt{D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\ \overline{\mathtt{N}}\ \overline{\mathtt{R}}\} \qquad \mathtt{B}\ \mathtt{m}\ (\overline{\mathtt{B}}\ \overline{\mathtt{x}}); \in \overline{\mathtt{N}}}{\mathit{mbody}(\mathtt{m},\mathtt{C}) = \bullet}$$

Figure 4. Lookup functions.

3. Typing

A type judgment of the form $\Gamma \vdash \mathtt{e} : \mathtt{T}$ states that "`e` has type `T` in the type environment Γ". A type environment is a finite mapping from variables (including `this` and `next`) to types, written $\overline{\mathtt{x}} : \overline{\mathtt{T}}$. Again, we use the sequence notation for abbreviating $\Gamma \vdash \mathtt{e}_1 : \mathtt{T}_1, \ldots, \Gamma \vdash \mathtt{e}_n : \mathtt{T}_n$ to $\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}}$.

Typing rules are shown in Figure 5. Method selection is allowed only on objects with a concrete type, where a type is *concrete* if it has no method requirements or it is a signature set `S` (used for the case of `next` as explained below). The key rule is (T-COMP) which deals with object composition. In this new language (w.r.t. [6]) the left object is not requested to be incomplete (it might be a complete object as well), neither the right object to be complete nor to provide some of the methods requested by the left object. Moreover, the right object can introduce new requirements, provided the signatures of the required methods are in the interface of the class of the left object; so the composition can also produce an object which is not yet complete. Note that the final type is the type based on the original class of the left 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. The (T-COMP) rule also shows that the typing of $\leftarrow\!\!+$ employs internally structural type checks, and we require no relation between the classes of the two objects; this is the key feature of our approach, which is aimed to enhance the flexibility of object composition.

The typing rules for methods and classes of FJ are adapted to our context (we still use the *override* predicate of [21] to check

Predicates

$$\frac{\mathit{required}(\texttt{C}) = \emptyset}{\mathit{concrete}(\texttt{C})} \qquad \mathit{concrete}(\langle \texttt{C}/\emptyset \rangle) \qquad \mathit{concrete}(\texttt{S})$$

$$\frac{\mathit{mtype}(\texttt{m},\texttt{D}) = \overline{\texttt{C}} \to \texttt{C} \text{ implies } \overline{\texttt{C}} = \overline{\texttt{B}} \text{ and } \texttt{C} = \texttt{B}}{\mathit{override}(\texttt{m},\texttt{D},\overline{\texttt{B}} \to \texttt{B})}$$

Expression typing

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

$$\frac{\Gamma \vdash \texttt{e} : \texttt{T} \qquad \mathit{fields}(\texttt{T}) = \overline{\texttt{C}}\,\overline{\texttt{f}}}{\Gamma \vdash \texttt{e.f}_i : \texttt{C}_i} \qquad \text{(T-FIELD)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{e} : \texttt{T} \qquad \Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{T}} \\ \mathit{mtype}(\texttt{m},\texttt{T}) = \overline{\texttt{B}} \to \texttt{B} \qquad \overline{\texttt{T}} <: \overline{\texttt{B}} \qquad \mathit{concrete}(\texttt{T})\end{array}}{\Gamma \vdash \texttt{e.m}(\overline{\texttt{e}}) : \texttt{B}} \qquad \text{(T-INVK)}$$

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

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{e}_1 : \texttt{T}_1 \qquad \mathit{otype}(\texttt{T}_1) = \langle \texttt{C}/\texttt{S}_1 \rangle \\ \Gamma \vdash \texttt{e}_2 : \texttt{T}_2 \qquad \mathit{otype}(\texttt{T}_2) = \langle \texttt{D}/\texttt{S}_2 \rangle \qquad \texttt{S}_2 \subseteq \mathit{interface}(\texttt{C})\end{array}}{\Gamma \vdash \texttt{e}_1 \mathrel{\leftarrow\!\!+} \texttt{e}_2 : \langle \texttt{C}/(\texttt{S}_1 - \mathit{defined}(\texttt{D})) \cup \texttt{S}_2 \rangle} \qquad \text{(T-COMP)}$$

Method and Class typing

$$\frac{\begin{array}{c}\overline{\texttt{x}} : \overline{\texttt{B}}, \texttt{this} : \langle \texttt{C}/\emptyset \rangle \vdash \texttt{e} : \texttt{T} \qquad \texttt{T} <: \texttt{B} \\ \texttt{class C inherits D } \{\overline{\texttt{C}}\,\overline{\texttt{f}};\, \overline{\texttt{M}}\,\overline{\texttt{N}}\,\overline{\texttt{R}}\} \qquad \mathit{override}(\texttt{m},\texttt{D},\overline{\texttt{B}} \to \texttt{B})\end{array}}{\texttt{B m } (\overline{\texttt{B}}\,\overline{\texttt{x}})\{\texttt{return e;}\}\ \texttt{OK IN C}} \qquad \text{(T-METHOD)}$$

$$\frac{\texttt{class C inherits D } \{\overline{\texttt{C}}\,\overline{\texttt{f}};\, \overline{\texttt{M}}\,\overline{\texttt{N}}\,\overline{\texttt{R}}\} \qquad \mathit{override}(\texttt{m},\texttt{D},\overline{\texttt{B}} \to \texttt{B})}{\texttt{B m } (\overline{\texttt{B}}\,\overline{\texttt{x}});\ \texttt{OK IN C}} \qquad \text{(T-AMETHOD)}$$

$$\frac{\begin{array}{c}\texttt{S} = \mathit{required}(\texttt{C}) \qquad \overline{\texttt{x}} : \overline{\texttt{B}}, \texttt{this} : \langle \texttt{C}/\emptyset \rangle, \texttt{next} : \texttt{S} \vdash \texttt{e} : \texttt{E} \qquad \texttt{E} <: \texttt{B} \\ \texttt{class C inherits D } \{\overline{\texttt{C}}\,\overline{\texttt{f}};\, \overline{\texttt{M}}\,\overline{\texttt{N}}\,\overline{\texttt{R}}\} \qquad \mathit{override}(\texttt{m},\texttt{D},\overline{\texttt{B}} \to \texttt{B})\end{array}}{\texttt{redef B m } (\overline{\texttt{B}}\,\overline{\texttt{x}})\{\texttt{return e;}\}\ \texttt{OK IN C}} \qquad \text{(T-RMETHOD)}$$

$$\frac{\overline{\texttt{M}}\ \texttt{OK IN C} \qquad \overline{\texttt{N}}\ \texttt{OK IN C} \qquad \overline{\texttt{R}}\ \texttt{OK IN C}}{\texttt{class C inherits D } \{\overline{\texttt{C}}\,\overline{\texttt{f}};\, \overline{\texttt{M}}\,\overline{\texttt{N}}\,\overline{\texttt{R}}\}\ \texttt{OK}} \qquad \text{(T-CLASS)}$$

Figure 5. Typing rules.

that the signature of a method is preserved by method overriding). When typing a method and a redefining method in a (possibly incomplete) class `C` (with rules (T-METHOD) and (T-RMETHOD), respectively), we cannot simply assume `C` for the type of `this`, since we would not be able to type any method invocation on `this` in the methods of `C` (in fact, the rule (T-INVK) would fail since *concrete*(`C`) will not hold if the class has some required methods). Although we prohibit to invoke methods on incomplete objects, it is still safe to accept method invocations on `this` inside an incomplete class, since, at run time, `this` will be replaced by a complete object; thus, we will assume $\langle \texttt{C}/\emptyset \rangle$ for the type of `this`.

In order to type a `redef` method, we also need to assume a type for `next` when typing its body; it is safe to assume it has the signature set $\texttt{S} = \mathit{required}(\texttt{C})$, i.e., the signature set of the required methods. This is consistent with the way `next` is bound in the operational semantic rule for redefined method invocation (see Section 4). As noted before, thanks to the way lookup functions are defined (Figure 4), the only operation that is possible on `next` is method invocation. Furthermore, the type of `next`, being a signature set, is always considered a concrete type, when typing method invocations.

Rule (T-CLASS) basically checks all the methods are `OK`. Note that, although the syntax of types includes also signature sets, we will use such types only to type `next` inside method bodies, and, in a well-typed expression, `next` can appear only as the receiver of a message: thus, expressions of the form `next.f` or `e.m(next)` cannot be type checked. This is due to the fact that the subtyping is not defined between a signature set and a class name or an object type, and signature sets cannot be used by the programmer to write types. In contrast, *mtype*, which is used in (T-INVK), is defined also for signature sets.

4. Operational semantics

The operational semantics, shown in Figure 6, is defined by the reduction relation $\texttt{e} \longrightarrow \texttt{e}'$, read "`e` reduces to $\texttt{e}'$ in one step". The standard reflexive and transitive closure of $\longrightarrow$, denoted by $\longrightarrow^\star$, defines the reduction relation in many steps. We adopt a deterministic call-by-value semantics. 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).

The operational semantics is defined on *annotated programs*, i.e., CompObJ programs where all expressions (including class method bodies) are annotated using the annotation function $\mathscr{A}$. Since this function relies on the static types, it is parametrized over a type environment Γ. In the following we will use `e` and $\overline{\texttt{e}}$ also for annotated expressions where not ambiguous.

Definition 4.1 (Annotation Function)**.** *The annotation of* `e` *with respect to* Γ*, denoted by* $\mathscr{A}[\![\texttt{e}]\!]_\Gamma$*, is defined on the syntax of* `e`*, by case analysis:*

- $\mathscr{A}[\![\texttt{x}]\!]_\Gamma = \texttt{x}$;
- $\mathscr{A}[\![\texttt{e.f}]\!]_\Gamma = \mathscr{A}[\![\texttt{e}]\!]_\Gamma\texttt{.f}$;
- $\mathscr{A}[\![\texttt{e.m}(\overline{\texttt{e}})]\!]_\Gamma = \mathscr{A}[\![\texttt{e}]\!]_\Gamma\texttt{.m}(\mathscr{A}[\![\overline{\texttt{e}}]\!]_\Gamma)^{\overline{\texttt{B}}\to\texttt{B}}$ *if* $\Gamma \vdash \texttt{e} : \texttt{T}$ *and* $\mathit{mtype}(\texttt{m},\texttt{T}) = \overline{\texttt{B}} \to \texttt{B}$;
- $\mathscr{A}[\![\texttt{new C}(\overline{\texttt{e}})]\!]_\Gamma = \texttt{new C}(\mathscr{A}[\![\overline{\texttt{e}}]\!]_\Gamma)$*;*
- $\mathscr{A}[\![\texttt{e}_1 \mathrel{\leftarrow\!\!+} \texttt{e}_2]\!]_\Gamma = \mathscr{A}[\![\texttt{e}_1]\!]_\Gamma \mathrel{\leftarrow\!\!+} \mathscr{A}[\![\texttt{e}_2]\!]_\Gamma$.

Given a method definition $\texttt{B m } (\overline{\texttt{B}}\,\overline{\texttt{x}})\{\texttt{return e;}\}$*, in a class* `C`*, the annotation of the method body* `e` *is defined as* $\mathscr{A}[\![\texttt{e}]\!]_{\overline{\texttt{x}}:\overline{\texttt{B}},\texttt{this}:\langle\texttt{C}/\emptyset\rangle}$.

Given a method redefinition $\texttt{redef B m } (\overline{\texttt{B}}\,\overline{\texttt{x}})\{\texttt{return e;}\}$*, in a class* `C`*, the annotation of the method body* `e` *is defined as* $\mathscr{A}[\![\texttt{e}]\!]_{\overline{\texttt{x}}:\overline{\texttt{B}},\texttt{this}:\langle\texttt{C}/\emptyset\rangle,\texttt{next}:\texttt{S}}$ *where* $\texttt{S} = \mathit{required}(\texttt{C})$.

This annotation will help the semantics in selecting the right method definition according to the type of the method used during the static type checking; in case of methods with the same name but different signatures within a composed object, the annotation will avoid invoking the wrong version (generating a run-time type error).

To represent a run-time object we use a pair of lists of values of the shape $\texttt{new C}(\overline{\texttt{v}})$. Indeed, we use two lists because we need to keep the original whole composed object so that we can bind `this` correctly: the first list is used for searching for the method, while the second list is the entire composed object. The rule for object instantiation (R-NEW) generates lists of the shape $\texttt{new C}(\overline{\texttt{v}}) :: \varepsilon$. Rule (R-COMP) composes two run-time objects by appending the lists; note that this requires that the objects involved in the

Redefining set

$$\mathit{redef}(\mathtt{C}) = \overline{\mathtt{R}} \quad \text{if} \quad \mathtt{class\ C\ inherits\ D\ }\{\,\overline{\mathtt{C}}\,\overline{\mathtt{f}};\ \overline{\mathtt{M}}\ \ \overline{\mathtt{N}}\ \ \overline{\mathtt{R}}\,\}$$

Reduction

$$\mathtt{new\ C}(\overline{\mathtt{v}}) \longrightarrow \langle \mathtt{new\ C}(\overline{\mathtt{v}}) :: \varepsilon, \mathtt{new\ C}(\overline{\mathtt{v}}) :: \varepsilon \rangle \qquad \text{(R-NEW)}$$

$$\begin{array}{c}\langle \mathtt{new\ C}(\overline{\mathtt{v}}) :: \mathtt{l}, \mathtt{new\ C}(\overline{\mathtt{v}}) :: \mathtt{l} \rangle \leftarrow\!\!\!+ \langle \mathtt{l}', \mathtt{l}' \rangle \longrightarrow \\ \langle \mathtt{new\ C}(\overline{\mathtt{v}}) :: \mathtt{l} :: \mathtt{l}', \mathtt{new\ C}(\overline{\mathtt{v}}) :: \mathtt{l} :: \mathtt{l}' \rangle\end{array} \qquad \text{(R-COMP)}$$

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{C}}\,\overline{\mathtt{f}}}{\langle \mathtt{new\ C}(\overline{\mathtt{v}}) :: \mathtt{l}, \mathtt{new\ C}(\overline{\mathtt{v}}) :: \mathtt{l} \rangle.\mathtt{f}_i \longrightarrow \mathtt{v}_i} \qquad \text{(R-FIELD)}$$

$$\frac{\mathit{mbody}(\mathtt{m},\mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e}_0) \qquad \mathtt{m} \notin \mathit{redef}(\mathtt{C})}{\langle \mathtt{new\ C}(\overline{\mathtt{v}}) :: \mathtt{l}, \mathtt{l}' \rangle.\mathtt{m}(\overline{\mathtt{u}})^{\overline{\mathtt{B}} \to \mathtt{B}} \longrightarrow [\overline{\mathtt{x}} \leftarrow \overline{\mathtt{u}}, \mathtt{this} \Leftarrow \langle \mathtt{new\ C}(\overline{\mathtt{v}}) :: \mathtt{l}, \mathtt{l}' \rangle]\mathtt{e}_0} \qquad \text{(R-INVK)}$$

$$\frac{\mathit{mbody}(\mathtt{m},\mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e}_0) \qquad \mathtt{m} \in \mathit{redef}(\mathtt{C})}{\begin{array}{c}\langle \mathtt{new\ C}(\overline{\mathtt{v}}) :: \mathtt{l}, \mathtt{l}' \rangle.\mathtt{m}(\overline{\mathtt{u}})^{\overline{\mathtt{B}} \to \mathtt{B}} \longrightarrow \\ [\overline{\mathtt{x}} \leftarrow \overline{\mathtt{u}}, \mathtt{this} \Leftarrow \langle \mathtt{new\ C}(\overline{\mathtt{v}}) :: \mathtt{l}, \mathtt{l}' \rangle, \mathtt{next} \leftarrow \langle \mathtt{l}, \mathtt{l}' \rangle]\mathtt{e}_0\end{array}} \qquad \text{(R-RINVK)}$$

$$\frac{\mathit{mbody}(\mathtt{m},\mathtt{C}) = \bullet}{\langle \mathtt{new\ C}(\overline{\mathtt{v}}) :: \mathtt{l}, \mathtt{l}' \rangle.\mathtt{m}(\overline{\mathtt{u}})^{\overline{\mathtt{B}} \to \mathtt{B}} \longrightarrow \langle \mathtt{l}, \mathtt{l}' \rangle.\mathtt{m}(\overline{\mathtt{u}})^{\overline{\mathtt{B}} \to \mathtt{B}}} \qquad \text{(R-DINVK)}$$

Congruence rules

$$\frac{\mathtt{e} \longrightarrow \mathtt{e}'}{\mathtt{e.f} \longrightarrow \mathtt{e}'.\mathtt{f}} \qquad \frac{\mathtt{e} \longrightarrow \mathtt{e}'}{\mathtt{e.m}(\overline{\mathtt{e}})^{\overline{\mathtt{B}} \to \mathtt{B}} \longrightarrow \mathtt{e}'.\mathtt{m}(\overline{\mathtt{e}})^{\overline{\mathtt{B}} \to \mathtt{B}}}$$

$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i'}{\mathtt{v}_0.\mathtt{m}(\overline{\mathtt{v}}, \mathtt{e}_i, \overline{\mathtt{e}})^{\overline{\mathtt{B}} \to \mathtt{B}} \longrightarrow \mathtt{v}_0.\mathtt{m}(\overline{\mathtt{v}}, \mathtt{e}_i', \overline{\mathtt{e}})^{\overline{\mathtt{B}} \to \mathtt{B}}} \qquad \frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i'}{\mathtt{new\ C}(\overline{\mathtt{v}}, \mathtt{e}_i, \overline{\mathtt{e}}) \longrightarrow \mathtt{new\ C}(\overline{\mathtt{v}}, \mathtt{e}_i', \overline{\mathtt{e}})}$$

$$\frac{\mathtt{e}_2 \longrightarrow \mathtt{e}_2'}{\mathtt{e}_1 \leftarrow\!\!\!+ \mathtt{e}_2 \longrightarrow \mathtt{e}_1 \leftarrow\!\!\!+ \mathtt{e}_2'} \qquad \frac{\mathtt{e}_1 \longrightarrow \mathtt{e}_1'}{\mathtt{e}_1 \leftarrow\!\!\!+ \mathtt{v} \longrightarrow \mathtt{e}_1' \leftarrow\!\!\!+ \mathtt{v}}$$

Figure 6. Semantics of CompObJ

composition consist of two identical lists (this is also required by rule (R-FIELD)); as it will be clear in the following, objects of the shape $\langle \mathtt{l}, \mathtt{l}' \rangle$ where $\mathtt{l} \neq \mathtt{l}'$ appear only as message receivers and they are produced only during method invocations; thus, the actual values, returned by methods, passed to methods, used in object compositions, etc., can only be of the shape $\langle \mathtt{l}, \mathtt{l} \rangle$.

The main idea of method invocation is to search for the method definition in the (class of the) head of the first list using the *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). The following two definitions show how to perform the binding of `this` in the method body. The expression $[\overline{\mathtt{x}} \leftarrow \overline{\mathtt{u}}, \mathtt{this} \Leftarrow \langle \mathtt{new\ C}(\overline{\mathtt{v}}) :: \mathtt{l}, \mathtt{l}' \rangle]\mathtt{e}$ denotes the expression obtained from $\mathtt{e}$ by replacing $\mathtt{x}_1$ with $\mathtt{u}_1, \ldots, \mathtt{x}_n$ with $\mathtt{u}_n$ and `this` with $\langle \mathtt{new\ C}(\overline{\mathtt{v}}) :: \mathtt{l}, \mathtt{l}' \rangle$ using the substitution of Definition 4.3. For redefining methods we also replace `next`, using the standard replacement (rule (R-RINVK)).

Definition 4.2 (*findredef*). *Given a method name, an object list and a method type we define:*

1. $\mathit{findredef}(\mathtt{m}, \varepsilon, \overline{\mathtt{B}} \to \mathtt{B}) = \emptyset$;
2. $\mathit{findredef}(\mathtt{m}, \mathtt{new\ C}(\overline{\mathtt{v}}) :: \mathtt{l}, \overline{\mathtt{B}} \to \mathtt{B}) =$
$$\begin{cases} \mathtt{new\ C}(\overline{\mathtt{v}}) :: \mathtt{l} & \mathit{if}\ \mathtt{m} \in \mathtt{redef}(\mathtt{C}) \wedge \overline{\mathtt{B}} \to \mathtt{B} = \mathit{mtype}(\mathtt{m}, \mathtt{C}) \\ \mathit{findredef}(\mathtt{m}, \mathtt{l}, \overline{\mathtt{B}} \to \mathtt{B}) & \mathit{otherwise.} \end{cases}$$

Definition 4.3 ($\mathtt{this} \Leftarrow \langle \mathtt{l}, \mathtt{l}' \rangle$)**.** *We define the substitution* $\mathtt{this} \Leftarrow \langle \mathtt{l}, \mathtt{l}' \rangle$ *on expressions as follows:*

1. $[\mathtt{this} \Leftarrow \langle \mathtt{l}, \mathtt{l}' \rangle]\mathtt{this} = \langle \mathtt{l}, \mathtt{l} \rangle$;
2. $[\mathtt{this} \Leftarrow \langle \mathtt{l}, \mathtt{l}' \rangle]\mathtt{x} = \mathtt{x}$ *where* $\mathtt{x} \neq \mathtt{this}$;
3. $[\mathtt{this} \Leftarrow \langle \mathtt{l}, \mathtt{l}' \rangle](\mathtt{this.m}(\overline{\mathtt{e}})^{\overline{\mathtt{B}} \to \mathtt{B}}) =$
let $\mathtt{l}_1 = \mathit{findredef}(\mathtt{m}, \mathtt{l}', \overline{\mathtt{B}} \to \mathtt{B})$ **in**
$$\begin{cases} \langle \mathtt{l}_1, \mathtt{l}' \rangle.\mathtt{m}([\mathtt{this} \Leftarrow \langle \mathtt{l}, \mathtt{l}' \rangle]\overline{\mathtt{e}})^{\overline{\mathtt{B}} \to \mathtt{B}} & \mathit{if}\ \mathtt{l}_1 \neq \emptyset \\ \langle \mathtt{l}, \mathtt{l}' \rangle.\mathtt{m}([\mathtt{this} \Leftarrow \langle \mathtt{l}, \mathtt{l}' \rangle]\overline{\mathtt{e}})^{\overline{\mathtt{B}} \to \mathtt{B}} & \mathit{otherwise}; \end{cases}$$
4. $[\mathtt{this} \Leftarrow \langle \mathtt{l}, \mathtt{l}' \rangle](\mathtt{e.m}(\overline{\mathtt{e}})^{\overline{\mathtt{B}} \to \mathtt{B}}) =$
$([\mathtt{this} \Leftarrow \langle \mathtt{l}, \mathtt{l}' \rangle]\mathtt{e}).\mathtt{m}([\mathtt{this} \Leftarrow \langle \mathtt{l}, \mathtt{l}' \rangle]\overline{\mathtt{e}})^{\overline{\mathtt{B}} \to \mathtt{B}}$ *where* $\mathtt{e} \neq \mathtt{this}$;
5. $[\mathtt{this} \Leftarrow \langle \mathtt{l}, \mathtt{l}' \rangle](\mathtt{this.f}) = \langle \mathtt{l}, \mathtt{l} \rangle.\mathtt{f}$;
6. $[\mathtt{this} \Leftarrow \langle \mathtt{l}, \mathtt{l}' \rangle](\mathtt{e.f}) = ([\mathtt{this} \Leftarrow \langle \mathtt{l}, \mathtt{l}' \rangle]\mathtt{e}).\mathtt{f}$ *where* $\mathtt{e} \neq \mathtt{this}$;
7. $[\mathtt{this} \Leftarrow \langle \mathtt{l}, \mathtt{l}' \rangle](\mathtt{new\ C}(\overline{\mathtt{e}})) = \mathtt{new\ C}([\mathtt{this} \Leftarrow \langle \mathtt{l}, \mathtt{l}' \rangle]\overline{\mathtt{e}})$;
8. $[\mathtt{this} \Leftarrow \langle \mathtt{l}, \mathtt{l}' \rangle](\mathtt{e}_1 \leftarrow\!\!\!+ \mathtt{e}_2) =$
$([\mathtt{this} \Leftarrow \langle \mathtt{l}, \mathtt{l}' \rangle]\mathtt{e}_1) \leftarrow\!\!\!+ ([\mathtt{this} \Leftarrow \langle \mathtt{l}, \mathtt{l}' \rangle]\mathtt{e}_2)$.

Field selections in a method body expect to deal with an object of the class where the field is defined (or of a subclass). Thus, we must substitute `this` with the first list that is where the invoked method is defined (case 5). Indeed, the first list implements the scope of `this` inside a method body. Another crucial case is when `this` occurs as a right-hand side expression, e.g., when passing this as a method argument; in this case the natural meaning is to refer to the current object "up to now" in the list, and the second list is useless in this context, thus we perform the substitution $[\mathtt{this} \Leftarrow \langle \mathtt{l}, \mathtt{l}' \rangle]\mathtt{this} = \langle \mathtt{l}, \mathtt{l} \rangle$ (case 1). This is also consistent with reduction rule (R-COMP), where we require that the objects consist of two identical lists (indeed, we can use `this` in an object composition, e.g., $\mathtt{e} \leftarrow\!\!\!+ \mathtt{this}$).

Concerning method invocation, we must take into consideration possible ambiguities due to method name clashes. Suppose we have an incomplete class *A* that requires a method *m* and defines a method *n*. An instance of *A* can be composed with an object that provides *m*, say an object of class *B* that also defines a method *n*, but with a different signature (see the stream example in Section 5). When we invoke *m* on the composed object, we actually execute the definition of *m* in *B*; if this method then invokes *n*, the definition of *n* in *B* must be executed (executing the version in *A* would not be sound). This is consistent with the typing that has checked the invocation of *n* in *B.m* using the signature of *B.n*. On the contrary, if the definition of *n* has the same signature as in *A*, then we must execute the version of *A* (according to the semantics of delegation) only if in *A n* is redefining. This method selection strategy is implemented by point 3 of Definition 4.3 by relying on the function *findredef* (Definition 4.2) which uses the static annotation: given a method name, an object list and a method signature, $\mathit{findredef}(\mathtt{m}, \mathtt{new\ C}(\overline{\mathtt{v}}) :: \mathtt{l}, \overline{\mathtt{B}} \to \mathtt{B})$ searches for the object in the list that redefines `m` (in particular, it checks whether in the class of the head of the list `m` is redefining, otherwise it performs a recursive lookup in the tail of the list). If the search succeeds, then we replace `this` with the sublist returned by *findredef* (this corresponds to the delegation mechanism of replacing `this` with the original sender); otherwise, we replace `this` with the head of the scanned list, since that method was not intended to be redefined.

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. Each object composition creates indeed a brand new object: for instance, each object composition would get a new object identifier in an imperative model. Thus, this design choice would scale well also to an imperative setting, since this mechanism will assure that there will not be problems when an object is pointed to by references in different parts of the program.

For proving that CompObJ is *type safe*, we first prove that well-typedness is preserved under reduction and then that well-typed expressions cannot get stuck due to a message-not-understood error or message-ambiguous error. Here we only state the two main theorems (the proofs are sketched in a document available online at `http://www.dsi.unifi.it/~bettini/compobj.pdf`)

Theorem 4.4 (Type Preservation)**.** *If* $\Gamma \vdash \mathtt{e} : \mathtt{T}$ *and* $\mathtt{e} \longrightarrow \mathtt{e}'$ *then* $\Gamma \vdash \mathtt{e}' : \mathtt{T}'$ *for some* $\mathtt{T}' <: \mathtt{T}$.

Theorem 4.5 (Progress)**.** *Let* $\mathtt{e}$ *be a closed run-time expression. If* $\vdash \mathtt{e} : \mathtt{T}$*, for some* $\mathtt{T}$*, and* $\mathtt{e} \longrightarrow \mathtt{e}'$ *for some* $\mathtt{e}'$*, then either* $\mathtt{e}'$ *is a final value, or* $\mathtt{e}'$ *can be reduced.*

5. 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 a richer Java-like syntax (and consider all methods as public) and we will denote object composition operation with `<-`. In [6] we presented the example of graphical widgets to show how incomplete objects and object compositions can be used to implement scenarios where usually the design pattern *command* [17] is used. That example can be straightforwardly transposed into this new language. The examples shown here are only sketched, i.e., we will not always show all the fields of the classes and the complete implementation of the methods, since we concentrate on the parts which are relevant to object composition functionalities of CompObJ; we observe that the scenarios considered here (as well as the ones we considered in [6]) are typical case studies considered in the literature in the context of object composition and delegation (we refer to Section 6).

Here, we will re-implement the stream example of [6] and show how the new object composition mechanism is more flexible. Typically, stream libraries are implemented using the pattern *decorator* [17]. 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 object composition and redefining methods we can easily implement a stream library, as sketched in Listing 1: the specific stream specializations rely on the methods provided during object composition (using `next`) and redefine them. In order to show how delegation is implemented in our language, we introduced also the method `readBuffer` both in `CompressStream` and in `BufferedStream`. These two methods, in spite of having the same name, are completely unrelated (we also used different signatures). The operational semantics (Section 4) guarantees that the right implementation will be invoked, depending on the context in which this method is invoked; for instance, the method `read` in `BufferedStream` invokes `readBuffer`, and at run time the version defined in `BufferedStream` will be selected (thus run-time type errors are avoided). The same holds when `readBuffer` is invoked in the method `uncompress` in `CompressStream`: the version of `readBuffer` in `CompressStream` will be selected at run time.

This example also gives an insight of the programming style which is induced by the design principles of our language: `FileStream` is not modeled as an incomplete class since it can be implemented with all the functionalities. Similarly, we could have a `SocketStream` as a complete class for writing to/reading from the network. On the contrary, `CompressStream` and `BufferedStream` rely on another stream and thus they are incomplete classes (and their methods are redefining). It is then clear that `CompressStream` and `BufferedStream` can be re-used independently from

```
class Stream {
  void write(byte[] b);
  byte[] read();
}

class FileStream inherits Stream {
  public FileStream(String filename) { ... }
  void write(byte[] b) { ... }
  byte[] read() { ... }
}

class CompressStream inherits Stream {
  redef void write(byte[] b) { next.write(compress(b)); }
  redef byte[] read() { return uncompress(next.read()); }
  byte[] compress(byte[] b) {...}
  byte[] uncompress(byte[] b) { ... readBuffer(size, b); ... }
  void readBuffer(int len, byte[] b) {...}
}

class BufferedStream inherits Stream {
  Buffer buff;
  redef void write(byte[] b) {
    if (buff.isFull()) next.write(b);
    else buff.append(b);
  }
  redef byte[] read() {
    if (buff.size() > 0) return readBuffer();
    ...
  }
  byte[] readBuffer() {...}
}
```

Listing 1: The implementation of streams using redefining methods.

the actual stream implementation; on the other hand, a `FileStream` can be "decorated" with further functionalities, but it could also be used as it is.

Here it is a possible object composition using the classes in Listing 1:

```
new CompressStream() <-
  (new BufferedStream() <- new FileStream("foo.txt"));
```

We build a compressed-buffered stream starting from a file stream. Implementing this scenario with the decorator pattern would require more programming, and the relations among the classes and objects would not be clear.

Differently from our previous work [6], in this language object composition can also generate an incomplete object, and we can compose two incomplete objects. Thus, the composition could also be written as follows (note the parenthesis):

```
(new CompressStream() <- new BufferedStream())
  <- new FileStream("foo.txt");
```

This means that the construction of "decorations" can also be handled in another method of the program, say `buildDecorations` (thus exploiting modular programming) and the resulting incomplete objects can then be assembled later in the program in order to build a complete object:

```
buildDecorations() <- new FileStream("foo.txt");
```

We can then reuse the same decorations (i.e., the composed incomplete object) also for other stream compositions, e.g., with `SocketStream`.

The ability of object composition to create also incomplete objects can be useful also in other scenarios; for instance, we might write an `EncryptStream` which redefines `write` and `read` and requires the methods `encrypt` and `decrypt` which will then be

```
class TreeIterator {                    class TextJustifier {
  void doAll() {                          void action() { // implemented
    firstElem();                            ...
    while (!isDone()) {                   }
      action();                         }
      nextElem();
    }
  }
  void firstElem() { ... }
  void nextElem() { ... }
  boolean isDone() { ... }
  void action(); // required
}
```

Listing 2: The implementation of iterator and text justifier.

provided by an object implementing the actual encryption algorithm, e.g., `RSAEncrypter`, `DESEncrypter`, etc. Composing an incomplete object `EncryptStream` and a complete object `RSAEncrypter` still produces an incomplete object (due to the redefining methods `write` and `read`).

We now show how to implement the programming scenario of iterator and text justifier (which can be seen as an example of pattern *strategy* [17]) borrowed from [27]. Suppose we want to implement the text justification in a document editor, which will have to iterate over the document structure. In Java we could make `TextJustifier` a subclass of `TreeIterator` so that it implements the abstract method `action` (which implements the actual strategy); however, this would make `TextJustifier` a subtype of `TreeIterator` which is an unwanted side-effect (and from the design point of view it might be wrong: the methods of `TreeIterator` would pollute the interface of `TextJustifier`). Alternatively, we could insert a field in `TreeIterator`, say `actionImpl`, and make it forward the method `action` to `actionImpl`. Besides this breaking the delegation [18], it would require `actionImpl` to be declared with some interface containing `action`, and thus, again, `TextJustifier` would require to implement that interface. Note that using anonymous inner-classes for creating a "proxy" that implements an interface without polluting the outer class interface would not solve the problem of broken delegation either.

In our language we can define the two classes as in Listing 2, without any form of relation or coupling between the them. We believe our solution is simpler than the approaches presented in [27] and, in general, than a wrapper solution (such as, e.g., [7, 10]). Thus we can compose two instances as follows:

```
new TreeIterator() <- new TextJustifier();
```

Note that this way the two classes belong to two separate hierarchies, and their interfaces and structure are not polluted with unwanted methods.

Following the scenario [27], we could also implement different iteration strategies, e.g., for pre-order and post-order visit. However, differently from [27], we could also do this without changing `TreeIterator` itself, by defining the following two incomplete classes which use redefining methods:

```
class PreOrder {                        class PostOrder {
  redef void firstElem() { ... }          redef void firstElem() { ... }
  redef void nextElem() { ... }           redef void nextElem() { ... }
}                                       }
```

Again, note that there is no subclass relations between these classes and `TreeIterator`. Then we can build the iteration functionality separately, to be finally composed with the text justification functionality:

```
iteration = (new PreOrder() <- new TreeIterator());
...
iteration <- new TextJustifier();
```

Thanks to delegation, when the `TreeIterator` object will invoke `nextElem` the redefining method in `PreOrder` will be selected, and when it will invoke `action`, the method defined in `TextJustifier` will be selected.

In the above examples the fact that no specific relation is needed in the class design highlights one of the important features of our approach: the object composition can model variations of the object behavior which might have not been anticipated in the class hierarchy.

It would be easy to implement other scenarios with our linguistic constructs, for instance, the *adapter* pattern [17], which are typically implemented with object composition; differently from manual implementations, adapters implemented with our object composition would benefit from delegation. Similarly, also logging functionalities (a typical example of *aspects*, see, e.g., [12]) could be straightforwardly implemented with object composition, redefining methods and delegation.

As a final note, we observe that the main capabilities of CompObJ could be emulated by a manual implementation of patterns. However, our aim was to model these capabilities as linguistic constructs (with a type system and a semantics, proved sound) and to integrate them in the static type system of mainstream languages. With incomplete objects, we tried to shorten the distance between the language features and the design patterns, so that their implementation can be smoother [8] and possibly more efficient [9].

6. Related Work

There are some similarities between our incomplete objects and approaches based on *delegation* [24, 30], which rely on object composition and method delegation as a more flexible and run-time version of class inheritance and method overriding: every object has a list of *parent* objects and when an object cannot answer a message it forwards it to its parents until there is an object that can process the message. However, a drawback of these approaches is that run-time type errors ("message-not-understood") can arise when no delegates are able to process the forwarded message [31]. For this reason, some linguistic approaches were studied to deal with delegation and type safety properties.

In [22] delegation is presented in the model of the language *Darwin*; however, in [22] 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 required methods can be provided by any object, independently from its class.

In [26] a model based on *delegation layers* is presented where all the features that are typical of class-based languages (inheritance, delegation, late binding and subtype polymorphism) automatically apply to sets of collaborating classes and objects. In [26] there is a high flexibility concerning the hierarchy of involved instances since delegation layers allow expressing configurations that cannot be modeled with delegation alone. On the other hand, this approach results in a more complex semantics concerning objects interaction/relation.

In [27] new abstractions for object references and composition are introduced, which provide explicit linguistic support for combining different composition properties on-demand. The model is statically typed and allows the programmer to express several kinds of composition semantics in the interval between object composition and inheritance. Also in this case, the several linguistic constructs can show more complex semantics concerning objects interaction/relation. Note that in our approach we also achieve *transparent redirection* [27], thus avoiding the before mentioned *self* *problem* [24] and *broken delegation* [18]. It would be interesting to investigate whether with our small set of linguistic mechanisms we can encode the several mechanisms of [27].

Incomplete objects, as a language construct, are more general-purpose than *wrappers* (see, e.g., [7, 10]) and, indeed, wrappers could be actually implemented through incomplete objects. Another form of wrapping of methods is the one offered by the *delegates* of C#, i.e., objects pointing to one method or to a set of methods, that will be executed when invoked appropriately on the delegate. Delegates can be seen as complementary to incomplete objects, which implement a different form of reuse, allowing to customize a prototype (i.e., an incomplete object) in more than one way via object composition. A further construct of C# that deals with some form of incompleteness is the one of *partial classes*, that makes it possible to subdivide a class definition among two or more files. However, this mechanism is a static one, while our object composition is dynamic.

Objective-C provides *categories*, a run-time mechanism for modifying existing code: 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. 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 [14] are composable units containing only methods, and they were proposed as an add-on to traditional class-based inheritance for 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.

There are some relations between *aspects* [12] 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 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. Thus, 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 [1] a general model for object composition is proposed, which is based on the design of classes in an aspect-oriented style. The authors do not formalize their model within a calculus, but the main feature is to compose dynamically the overall behavior of an object from the multiple "aspects" that abstract the variant behavior [2]. The main difference with respect to our language is that for them the run-time behavior is codified in aspects, while we internalize it in a Java-like setting by exploiting incomplete classes and object composition.

The language *gbeta* [15] supports a mechanism called "object metamorphosis" to specialize dynamically an existing object: a class is applied to it as a constraint and the object becomes an instance of that class. The main difference between the gbeta 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. The language gbeta also supports dynamic class composition [25] while in our language we act on object composition.

Roles [19, 23] are a conceptual abstraction that can be used in object-oriented systems to implement specific entities within a domain. Both roles and incomplete objects have an inherent compositional nature, but the two approaches are rather different both for features and for intention. First of all, once objects are composed, they cannot be de-composed (although we might consider studying such an operation and how this affects the static type system). On the contrary, roles can be attached to base objects and detached; in particular, upon removal, a role is also destroyed. This is another important difference with respect to our object composition: objects in our language keep their own identity and life cycle (and they can be used in many object composition), while roles can be attached to one base object only and they "live" only when they are part of such a base object. Moreover, role definitions also specify the class of their base objects, and this couples them to these classes, while in our approach the type of objects in composition is not known in advance.

In [32] linguistic abstractions for component-oriented programming are added on top of FJ: while this notion of component inspired our view of objects, in [32] components are distinct linguistic constructs from the notion of objects as class instances. Moreover, type safety in [32] is guaranteed only at the cost of a number of syntactic restrictions.

In [20] the language *MorphJ* offers the mechanism of *nested patterns* to implement a form of "class morphing", which enables safe static reflection over members of a type, in order to generate new classes in a meta-programming style. Generic classes are used to abstract over the structure of methods in other classes and they can be type checked separately from the classes that instantiate it, in a type safe way. The main point of similarity between [20] and our approach is the aim of integrating abstraction linguistic constructs into mainstream languages with static typing. However, such constructs are at the class level in MorphJ, whereas they work dynamically at object composition level in CompObJ.

7. Conclusions

We presented a core language CompObJ with incomplete objects, object composition and delegation, by integrating a mechanism of object composition into a statically typed class-based language. We achieve the flexibility of the object composition typical of object-based language, while retaining the safety of Java-like languages. The underlying idea of CompObJ is to transpose at run time the features of abstract classes in standard class-based languages: if `C` is an abstract class in a Java-like language, then a variable `c` can be safely declared with type `C`: since the type system prohibits to instantiate abstract classes, at run time the variable `c` will only refer to subclasses of `C` which are not abstract (i.e., they implement all the abstract methods of `C`). Analogously, in CompObJ, if `C` is an "incomplete" class, we can safely instantiate `C` (i.e., an incomplete object): method invocation cannot be performed on objects which are incomplete (the type system prevents this); however, that instance can be used in object compositions, and when a composed object is complete (all requirements are fulfilled) it can be used for method invocations.

The functionalities provided by the dynamic method redefinition, together with the delegation mechanism, further unleash the flexibility of object composition. By representing objects as lists of subobjects, we can explicitly deal with the "scope" of a method invocation. Our solution avoids possible name clashing and accidental overrides, and it is much more implementation-oriented than the dictionaries of [28] and simpler than the one of [3].

In [6] we showed how consultation can coexist with delegation in the context of incomplete objects. Consultation is not only useful as a preliminary study for appreciating how the composition mechanism can be integrated in a Java-like, class-based setting but it is interesting in its own, since it provides the programmer with more control on method invocation. The same technique can be applied also to CompObJ to provide the programmer with choice between consultation and delegation.

In [4] we presented I-Java, an extension of the Java language with a very simple notion of incomplete objects and object composition (with consultation mechanism only). We implemented a preprocessor that, given a program that uses our language extension, produces standard Java code (the preprocessor is available at `http://i-java.sf.net`). The integration in Java of incomplete objects with delegation according to the presented approach is currently under development and the I-Java implementation will be the starting point. The code generated by I-Java does not contain much overhead with respect to manual implementations (e.g., of patterns); in particular, method forwarding for incomplete methods would be present also in standard design pattern implementations. The only additional overhead is the one due to `myThis`, a special variable that is used to simulate the binding of `this` and to achieve the flexibility of incomplete objects that can be composed with objects independently from their types. `myThis` is also used to access the head of the list representing a composed object (again if these capabilities were to be implemented manually, the overhead would be the same). When implementing CompObJ we think we can reuse this special variable to implement the *findredef* function, i.e., to achieve the delegation mechanism for method invocation. Again, we believe that this additional overhead would be the same as the one of a manual implementation to achieve the semantics of delegation.

The present proposal seems to be a useful approach to deal with the problem of dynamic reconfiguration of mobile code [5] both in the context of service oriented programming and in web services; in these scenarios, mechanisms enabling service composition and reconfiguration, based on types, could be implemented through incomplete objects. In this direction, we plan to investigate how our approach to object composition can be exploited in calculi that incorporate session types in an object-oriented framework, such as [13].

Finally, interaction of our linguistic constructs with Java *generics* can be an interesting future research subject allowing generic class instances to take part to object compositions.

Acknowledgments We warmly thank Viviana Bono for her invaluable contribution to the development of the present approach. Our thanks to Tina Fasulo for the helpful insights she provided when working on her master thesis. Finally, we thank the anonymous referees for their suggestions for improving the presentation.

References

[1] C. Babu and D. Janakiram. Method Driven Model: A Unified Model for an Object Composition Language. *ACM SIGPLAN Notices*, 39(8):61–71, 2004.

[2] C. Babu, W. Jaques, and D. Janakiram. DynOCoLa: Enabling Dynamic Composition of Object Behaviour. In *RAM-SE*, 2005.

[3] L. Bettini, V. Bono, and S. Likavec. Safe and Flexible Objects with Subtyping. *Journal of Object Technology*, 10(4):5–29, 2005.

[4] L. Bettini, V. Bono, and E. Turin. I-Java: an extension of Java with incomplete objects and object composition. In *Software Composition*, volume 5634 of *LNCS*, pages 27–44. Springer, 2009.

[5] L. Bettini, V. Bono, and B. Venneri. MoMi: a calculus for mobile mixins. *Acta Informatica*, 42(2-3):143–190, 2005.

[6] L. Bettini, V. Bono, and B. Venneri. Delegation by object composition. *Science of Computer Programming*, 76(11):992–1014, 2011.

[7] L. Bettini, S. Capecchi, and E. Giachino. Featherweight Wrap Java: wrapping objects and methods. *Journal of Object Technology*, 7(2):5–29, 2008.

[8] J. Bishop. Language features meet design patterns: raising the abstraction bar. In *ROA*, pages 1–7. ACM, 2008.

[9] J. Bishop and R. N. Horspool. On the Efficiency of Design Patterns Implemented in C# 3.0. In *TOOLS*, volume 11 of *LNBIP*, pages 356–371. Springer, 2008.

[10] M. Büchi and W. Weck. Generic wrappers. In *ECOOP*, volume 1850 of *LNCS*, pages 201–225. Springer, 2000.

[11] C. Chambers. Object-Oriented Multi-Methods in Cecil. In *ECOOP*, volume 615 of *LNCS*, pages 33–56. Springer, 1992.

[12] D. Crawford. *Communications of the ACM archive - Special Issue on Aspect-Oriented Programming*, volume 44. ACM, 2001.

[13] M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session Types for Object-Oriented Languages. In *ECOOP*, volume 4067 of *LNCS*, pages 328–352. Springer, 2006.

[14] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *ACM TOPLAS*, 28(2):331–388, 2006.

[15] E. Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Århus, Denmark, 1999.

[16] K. Fisher and J. C. Mitchell. A Delegation-based Object Calculus with Subtyping. In *FCT*, volume 965 of *LNCS*, pages 42–61. Springer, 1995.

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

[18] W. Harrison, H. Ossher, and P. Tarr. Using Delegation for Software and Subject Composition. Technical Report RC 20946, IBM Thomas J. Watson Research Center, 1997.

[19] S. Herrmann. A precise model for contextual roles: The programming language ObjectTeams/Java. *Applied Ontology*, 2(2):181–207, 2007.

[20] S. S. Huang and Y. Smaragdakis. Expressive and safe static reflection with MorphJ. In *PLDI*, pages 79–89. ACM, 2008.

[21] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.

[22] G. Kniesel. Type-Safe Delegation for Run-Time Component Adaptation. In *ECOOP*, volume 1628 of *LNCS*, pages 351–366. Springer, 1999.

[23] B. B. Kristensen and K. Østerbye. Roles: Conceptual abstraction theory and practical language issues. *Theory and Practice of Object Sytems*, 2(3):143–160, 1996.

[24] H. Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. *ACM SIGPLAN Notices*, 21(11):214–214, 1986.

[25] A. B. Nielsen and E. Ernst. Optimizing Dynamic Class Composition in a Statically Typed Language. In *TOOLS*, volume 11 of *LNBIP*, pages 161–177. Springer, 2008.

[26] K. Ostermann. Dynamically composable collaborations with delegation layers. In *ECOOP*, volume 2374 of *LNCS*, pages 89–110. Springer, 2002.

[27] K. Ostermann and M. Mezini. Object-Oriented Composition Untangled. In *OOPSLA*, pages 283–299. ACM, 2001.

[28] J. Riecke and C. Stone. Privacy via Subsumption. *Information and Computation*, 172:2–28, 2002.

[29] A. Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):438–479, Sept. 1996.

[30] D. Ungar and R. B. Smith. Self: The power of simplicity. *ACM SIGPLAN Notices*, 22(12):227–242, 1987.

[31] J. Viega, B. Tutt, and R. Behrends. Automated Delegation is a Viable Alternative to Multiple Inheritance in Class Based Languages. Technical Report CS-98-03, UVa Computer Science, 1998.

[32] M. Zenger. Type-Safe Prototype-Based Component Evolution. In *ECOOP*, volume 2374 of *LNCS*, pages 470–497. Springer, 2002.