JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 7, No. 2, Special Issue OOPS Track at SAC 2007, February 2008

# Featherweight Wrap Java: wrapping objects and methods

**Lorenzo Bettini**, bettini@dsi.unifi.it
Dipartimento di Informatica, Università di Torino,
**Sara Capecchi**, capecchi@dmi.unict.it
Dipartimento di Matematica e Informatica, Università di Catania,
**Elena Giachino**, giachino@di.unito.it
Dipartimento di Informatica, Università di Torino.

*This work has been partially supported by the MIUR project EOS-DUE.*

We present a language extension, which integrates in a Java like language a mechanism for dynamically extending object behaviors without changing their type. Our approach consists in moving the addition of new features from class (static) level to object (dynamic) level: the basic features of entities (representing their structure) are separated from the additional ones (wrapper classes whose instances represent run-time added behaviors). At run-time, these entities can be dynamically composed by instantiating wrapper objects which are attached to basic entities. Wrappers permit specializing (wrapping) methods at run-time: the method of the wrapper will be automatically executed after the method of the wrapped objects, using a delegation mechanism. Furthermore, wrapped methods can return values and values returned by the wrapped methods are transparently made available in the wrapper methods. We formalize our extension by adding the new constructs to Featherweight Java and we prove that the core language so extended (Featherweight Wrap Java) is type safe.

## 1 INTRODUCTION

Class inheritance is a key feature in Object-Oriented Programming, since it provides means for code reusability and, from the type perspective, it allows the programmer to address flexibility in a safe way. However, class inheritance is essentially a static mechanism: the relation between a parent and a derived class is established statically, once and for all; should this relation be changed, then the program has to be modified and recompiled. The only dynamic feature is represented by *dynamic binding*, i.e., the dynamic selection of a specific method implementation according to the run-time type of an object. This may not suffice for representing the dynamic evolution of objects that behave differently depending on their internal state, the context where they are executing or the entities they interact with. All these possible behaviors may not be completely predictable in advance and they are likely to change after the application has already been developed and used. While trying to forecast all the possible evolutions of system entities, classes are often designed with too many responsibilities, most of which are basically not used.

Cite this document as follows: Lorenzo Bettini, Sara Capecchi, Elena Giachino: *Featherweight Wrap Java: wrapping objects and methods*, in Journal of Object Technology, vol. 7, no. 2, Special Issue OOPS Track at SAC 2007, February 2008, pp. 5–29
http://www.jot.fm/issues/issues_2008_2/article1

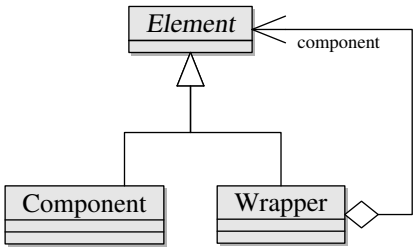



Figure 1: Wrapper and component classes.

Furthermore, the number of subclasses tend to grow dramatically when trying to compose different functionalities into single modules. With this respect, object composition is often advocated as an alternative to class inheritance, in that it is defined at run-time and it enables dynamic object code reuse by assembling the existing components [17].

Furthermore, in many situations, when deriving from a base class, we need to *specialize* methods (i.e., add some behavior while still relying on the implementation of the base class) instead of overriding them. For instance, the method `paint` in a GUI framework is a callback method that is invoked by the framework when the contents of a window have to be redrawn on the screen. The programmer is required to extend this method in order to take care of drawing the contents that are specific of the application, while standard graphical items (menu, toolbars, button, etc.) are drawn by the implementation of the superclasses in the framework. Thus, the programmer has to explicitly call `super.paint()` in his redefined method, otherwise the window will not be correctly redrawn. However, there are no means to formally enforce the call to the super version, but (informally) documenting the framework. The design pattern *decorator* [17] is often used in contexts where we need to specialize ("decorate") object behaviors at run-time; however, it still requires manual programming, and, again, there's no static guarantee that the version of the method in the base class is called.

On the other hand, object based languages use object composition and *delegation*, a more flexible mechanism, to reuse code (see, e.g., [30, 19, 11], and the calculi [15, 1, 2]). Every object has a list of *parent* objects: when it 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 [31]. In order to preserve the benefits of static type safety, several solutions have been proposed in the literature such as design patterns [17] and language extensions integrating in class based languages more flexible mechanisms, such as, e.g., mixins [7], generic types [9], delegation [23].

In this paper, we propose an extension of *FJ* (*Featherweight Java*) [21], we called *FWJ* (*Featherweight Wrap Java*), which integrates in a Java like language a mechanism for dynamically extending object behaviors without changing their type. Our approach consists in moving the addition of new features from class (static) level to object (dynamic) level: we propose a mechanism that separates in different entities the basic features of elements (representing their structure) from the additional ones (representing their run-time added behaviors). At run-time, these entities can be dynamically composed. The basic idea of

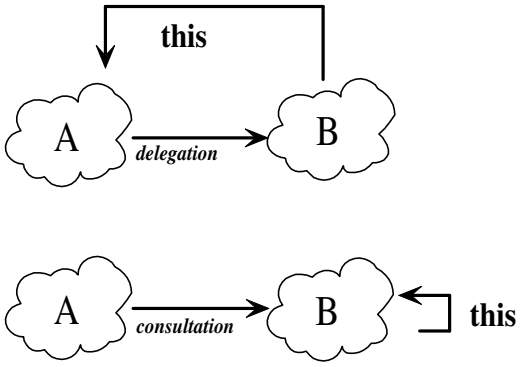



Figure 2: Delegation and consultation.

our approach can be summarized as follows: at run time an object (called *component*) is embedded in another object (called *wrapper*) that associates to the component additional features (see Figure 1). A wrapper object's interface is conforming to the one of the component so the power of polymorphism can be exploited: we can assign to a variable `a` of type `Element` a wrapper object that embeds an instance of `Element`, called `component`. Every time a client invokes a method `m` on `a` that belongs to the interface `Element`, it (automatically and transparently) forwards the method call to its attribute `component`, possibly adding code. These class and object structures are very similar to those proposed in the design pattern *decorator* [17]. However, the decorator pattern only implements a *consultation* mechanism and does not extend the language: it requires manual programming. Conversely, by using our language extension the implementation of decorator is straightforward; furthermore, we provide a *delegation* based mechanism.

Let us notice that, in the literature (e.g., [17]), the term *delegation*, originally introduced by Lieberman [24], 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 `this` can be adopted for the execution of the body of *m*: with *delegation*, `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 *m*, then also this call will be executed binding `this` to *A*; with *consultation*, during the execution of the body the implicit parameter is always bound to the receiver *B* (Figure 2).

Following the key idea discussed above we propose to extend a Java like language by adding the following new constructs to define wrapper classes and create wrapper objects (in the following, `I` is used to refer to interfaces):

- `class D wraps I` *bodyD*. In *bodyD* the programmer can add fields and new methods w.r.t. `I` and can implement ("wrap") some methods belonging to `I`, using the keyword `after` in method definition: the wrapper method implementation will be executed after the wrapped method implementation. This way we obtain *method specialization* (i.e., a method redefinition does not completely overrides the previous definition). This construct also states that instances of class `D` can be used to

wrap, at run-time, instances of `I`. Since a new subtyping relation, namely `D` is a subtype of `I`, is introduced by this construct, then a wrapper object can wrap another wrapper. Notice that not all methods of `I` need to be implemented in *bodyD*. However, this does not cause run-time errors (e.g., "message-not-understood") because instances of `D` can only be used to wrap instances (or, in turn, wrapper objects) of a class implementing (all methods of) `I`, and methods not implemented in the wrapper class are simply forwarded to the wrapped component.

- Concerning object creation, an instantiation of a wrapper class can only be of the shape `newwrap D`$(x, f_1, \ldots, f_n)$ where $x$ is an instance of `I` (the object to be wrapped) and $f_1, \ldots, f_n$ are the parameters for the wrapper's constructor. Notice that two different wrappers can wrap the same component.

FWJ was inspired by [5], which, however, presented only an implementation schema (with less features than our language extension) and no real formalization. The first version of FWJ was presented in [4]; in this paper we deal with methods that return values (in [4] only `void` methods were considered), and provide mechanisms to deal with returned values during the method invocation chain. In particular, since we have already treated `void`, in order to simplify the presentation of FWJ, in this version, we will only consider non-`void` methods. Moreover, we show the main properties of FWJ (namely, FWJ is type safe) together with the proofs of crucial cases (Section 3). To further simplify and shorten the presentation of our language, in this paper, we consider only method invocation by delegation, since it requires a more involved (and interesting) technical treatment of the substitution of `this`; invocation by consultation can be smoothly added to the language (as shown in [4]) since `this` can be handled just as in FJ.

In the previous version of FWJ, since we were considering only `void` methods, we were only concerned about the order in which methods of wrappers and components were executed in the semantics. In this version, since we deal with methods returning a value, we need to handle return values, and in particular, since the method invocation chain is transparent to the programmer, we need to provide an automatic mechanism to access the value implicitly returned by the previous invocation within the method invocation chain. Notice that we do not want to impose a decision about the actual value returned to the original caller: we do not want the returned value to be the value returned by the last (or the first) invocation in the chain. Since we want our wrapper mechanism to be flexible, we let the programmer of wrappers decide what to return. Thus, we provide the programmer with a means to access to the value returned by the previous invocation in the method chain: every wrapped method has an implicit variable, `ret`, that, similarly to `this`, is initialized automatically with the value returned by the previous method invocation in the chain. In [4] we allowed wrapper methods also to be executed "before" the methods of the component (using the keyword `before`); however, in the scenario of methods returning a value, this method invocation mechanism makes no sense, in particular the `ret` automatic variable would not be initialized in some contexts (e.g., during the execution of the first method invocation on the wrapper). For this reason, in this version of the language we will not consider wrapped methods declared as `before`.

In Listing 1 we present a code snippet using our wrappers. This implements a scenario

```
interface TextElement { String render(); }

class SimpleText implements TextElement {
      String data;
      String render() { return data; }
}

class HtmlBold wraps TextElement {
      after String render() {
            return "<b>" + ret + "</b>";
      }
}

class HtmlColor wraps TextElement {
      String color;
      after String render() {
            return "<font color=\"" + color + "\">" + ret + "</font>";
      }
}

TextElement e =
  newwrap HtmlColor(newwrap HtmlBold(new SimpleText("Hello World!")), "green");
System.out.println(e.render());
```

**Listing 1:** The formatting scenario using wrappers (constructors are not shown).

where wrappers are used to implement text decorated with specific HTML styles (such as bold and color). Different wrapper formatters can be composed at run-time and when the method `render` is called on the wrapped text, the style tags will surround the actual text (or the further formatted text). In particular, the above code will print the string:

```
<font color="green"><b>Hello World!</b></font>
```

Notice how the implicit variable `ret` is used to add a style to a (possibly already formatted) text.

From the typing point of view, the resulting wrapped object will have the type of the most external wrapper (e.g., in the example of Listing 1, the type of `HtmlColor`); this will be reflected in the type system (rule T-NEWWRAP, Figure 7). However, as a good programming technique to make code reuse easier, it is suggested not to treat the wrapped object as an instance of `HtmlColor`, but as an instance of the wrapped interface (see Listing 1, where the wrapped object is assigned to a reference of type `TextElement`).

In Listing 2 we show an example which exploits the delegation mechanism. The hierarchy is composed by an interface `CallRate` which abstracts the methods common to call rates (a method `bill` which calculates the total bill and `displayRate` which returns a

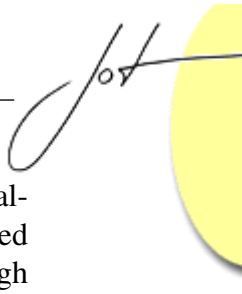
```
interface CallRate {
    int bill(int m);
    String displayRate();
}

class SimpleRate implements CallRate {
    int rate;
    after String displayRate() { return "Call Rate: " + rate + " cents/min;"; }

    after int bill(int m){
        System.out.println(displayRate());
        return m * rate;
    }
}

class FreqCall wraps CallRate {
    int top;
    after String displayRate() {
        return "50 % for bills > " + top + " euros; " + ret;
    }

    after int bill(int m){
        if (ret > top) return (ret − (0,5*(ret−top)));
        else return ret;
    }
}
```

**Listing 2:** The CallRate example using wrappers and delegation (constructors are not shown).

string describing the kind of rate/discount), a component class `SimpleRate` and a wrapper class `FreqCall` implementing a discount for phone rates. Discounts are modeled with wrapper classes since they make sense only when applied to rates, moreover with our approach we can obtain customized rates combining different wrapper-discounts. In class `SimpleRate` method `bill` calls method `displayRate` to show how the total amount is calculated. For instance, let us consider the following code:

```
CallRate r = newwrap FreqCall(new SimpleRate(15), 50);
int b = r.bill(400);
```

During the execution of method `bill`, the string `"50 % for bills > 50 euros; Call Rate 15 cents/min;"` will be displayed; indeed when the control reaches the call of `displayRate` in the implementation of `bill` in `SimpleRate`, thanks to delegation, the implicit object `this` will be bound to the original caller, i.e., the `FreqCall` wrapper, thus the string describing the discount will be displayed as well. This would not happen without delegation.



The example of Listing 2 also highlights another feature of wrapped methods: although the order of method invocation is fixed (the method body of the wrapper is called after the method body of the wrapped object), the `ret` mechanism is flexible enough to allow the programmer to build the result as he sees fit. In fact, in the example, the result of `SimpleRate.displayRate` is appended to the end of the string built in `FreqCall.displayRate`, in spite of the latter being executed after the former. Thus, the absence of wrapper methods declared as `before`, as in [4], does not limit the flexibility of methods that return values.

We refer the reader to [4] for further examples of use of wrappers (with `void` methods).

This paper is organized as follows: Section 2 formalizes our extension by adding our new constructs to *Featherweight Java* and Section 3 presents the formal properties of our FJ extension (in particular, the type safety of our language); Section 4 concludes the paper and compares our approach to related works.

## 2 FEATHERWEIGHT WRAP JAVA

This section presents syntax, typing rules and operational semantics of FWJ (*Featherweight Wrap Java*), a minimal imperative core calculus for Java, based on *Featherweight Java* (abbreviated with FJ), extended to the wrapper class constructs. FJ [21, 29] 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 `this`, subtyping and field access[1]. 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", [29], pag. 248).

Besides the innovative features of the wrapper construct, FWJ models interfaces and some imperative features like field assignment and sequential composition. Our imperative model is inspired by the one in [12]. For the sake of completeness we show the complete syntax, typing and semantics of FJ augmented with interfaces, imperative features and wrapper constructs; comments will stress on elements added by our extension. We assume the reader is familiar with FJ (we refer to [21] for details on standard characteristics), thus we will focus on the novel aspects of FWJ w.r.t. FJ.

The abstract syntax of FWJ is given in Fig. 3. The metavariables `C` and `D` range over class names (we adopt the convention that `C` is an ordinary class name and `D` is a wrapper class name; we will use `E` when we do not want to specify the kind of class we are referring to) and `I` ranges over interface names (the metavariable `T` denotes a class name, a wrapper class name or an interface name); `f` and g range over attribute names; `x` ranges over method parameter names; and `e` ranges over expressions. Following FJ, we assume

[1] FJ also includes up and down casts; however, since these features are completely orthogonal to our context, they are omitted in FWJ.

| | | | |
|---|---|---|---|
| L | ::= | class C extends C implements $\overline{\texttt{I}}$ {$\overline{\texttt{T}}$ $\overline{\texttt{f}}$; K; $\overline{\texttt{M}}$} | classes |
| H | ::= | interface I extends $\overline{\texttt{I}}$ {$\overline{\texttt{SG}}$} | interfaces |
| DCL | ::= | class D wraps I {I g; $\overline{\texttt{T}}$ $\overline{\texttt{f}}$; KD; $\overline{\texttt{M}}$ $\overline{\texttt{N}}$} | wrapper classes |
| SG | ::= | T m ($\overline{\texttt{T}}$ $\overline{\texttt{x}}$); | signatures |
| K | ::= | C($\overline{\texttt{T}}$ $\overline{\texttt{f}}$, $\overline{\texttt{T}}'$ $\overline{\texttt{f}}'$){super($\overline{\texttt{f}}$); this.$\overline{\texttt{f}}'$=$\overline{\texttt{f}}'$;} | constructors |
| KD | ::= | D(I g; $\overline{\texttt{T}}$ $\overline{\texttt{f}}$){this.g=g; this.$\overline{\texttt{f}}$=$\overline{\texttt{f}}$;} | wrapper constructors |
| M | ::= | T m ($\overline{\texttt{T}}$ $\overline{\texttt{x}}$){e} | methods |
| N | ::= | after T m ($\overline{\texttt{T}}$ $\overline{\texttt{x}}$){e} | wrapper methods |
| e | ::= | x \| e.f \| e.m($\overline{\texttt{e}}$) \| new C($\overline{\texttt{e}}$) \| e;e \| newwrap D(e,$\overline{\texttt{e}}$) \| e.f = e | expressions |

Figure 3: FWJ syntax

that the set of variables includes the special variable `this`, which cannot be used as the name of a method's formal parameter (this restriction is imposed by the typing rules). Instead, `this` is considered to be implicitly bound in any method declaration. Note that since we treat `this` in method bodies as an ordinary variable, no special syntax for it is required. The same holds for the other implicit variable `ret` in wrapper class methods, implicitly bound to the value returned by the method invocation on the component.

As in FJ, we write "$\overline{\texttt{e}}$" as a shorthand for a possibly empty sequence "$\texttt{e}_1,\ldots,\texttt{e}_n$" (and similarly for C, T, x) and "$\overline{\texttt{M}}$" as a shorthand for "$\texttt{M}_1 \ldots \texttt{M}_n$" (and similarly for N). Given a sequence $\overline{\texttt{e}}$, $|\overline{\texttt{e}}|$ represents its length. We abbreviate operations on pair of sequences by writing "$\overline{\texttt{T}}$ $\overline{\texttt{f}}$" for "$\texttt{T}_1$ $\texttt{f}_1;\ldots;\texttt{T}_n$ $\texttt{f}_n$", where $n$ is the length of $\overline{\texttt{T}}$ and $\overline{\texttt{f}}$. The empty sequence is denoted by $\bullet$. We assume that sequences of attributes, method parameters and method declarations do not contain duplicate names. A class table $CT$ is a mapping from class names to class declarations. Then a program is a pair $(CT, \texttt{e})$ 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 some usual sanity conditions: $(i)$ $CT(\texttt{C}) = \texttt{class C} \ldots$ for every $\texttt{C} \in dom(CT)$ $(ii)$ for every class name C (except `Object`) appearing anywhere in $CT$, we have $\texttt{C} \in dom(CT)$; $(iii)$ there are no cycles in the transitive closure of the `extends` relation. Thus, in the following, instead of writing $CT(\texttt{C}) = \texttt{class} \ \ldots$ we will simply write class C ..... The same conditions apply also to wrapper classes and interfaces.

The *wrapper class declaration* class D wraps I {I g; $\overline{\texttt{T}}$ $\overline{\texttt{f}}$; KD; $\overline{\texttt{M}}$ $\overline{\texttt{N}}$} defines a wrapper class of name D. The new class has a single constructor KD, a field g that specifies the wrapped object component, a sequence of fields $\overline{\texttt{f}}$ and a set of standard methods $\overline{\texttt{M}}$ and a set of wrapped methods $\overline{\texttt{N}}$ (in particular, the wrapped methods must belong to the wrapped interface I, as checked by the typing rules). The *wrapper constructor declaration* $\texttt{D}(\texttt{I g}, \overline{\texttt{T}}\ \overline{\texttt{f}})\{\texttt{this.g} = \texttt{g}; \ \texttt{this.}\overline{\texttt{f}} = \overline{\texttt{f}};\}$ takes parameters corresponding to the component and the instance variables, and its body consists of the corresponding assignments. Note that in FWJ the wrapped object is explicitly denoted in the class declaration, but it could be implicit in an implementation. Note also that, for the sake of simplicity, the calculus does not allow a wrapper class to have a superclass. Nevertheless, the type system could be straightforwardly extended to handle wrapper class inheritance. The *wrap-*

$$\texttt{T} <: \texttt{T} \qquad \frac{\texttt{T} <: \texttt{T}' \quad \texttt{T}' <: \texttt{T}''}{\texttt{T} <: \texttt{T}''} \qquad \frac{\texttt{interface I extends } \overline{\texttt{I}}\{\ldots\}}{\texttt{I} <: \texttt{I}' \; \forall \texttt{I}' \in \overline{\texttt{I}}}$$

$$\frac{\texttt{class C extends C}' \texttt{ implements } \overline{\texttt{I}} \; \{\ldots\}}{\texttt{C} <: \texttt{C}' \quad \texttt{C} <: \texttt{I} \; \forall \texttt{I} \in \overline{\texttt{I}}} \qquad \frac{\texttt{class D wraps I} \; \{\ldots\}}{\texttt{D} <: \texttt{I}}$$

Figure 4: FWJ: subtyping

$$\mathit{fields}(\texttt{Object}) = \bullet \qquad \frac{\texttt{class D wraps I } \{\texttt{I g}; \; \overline{\texttt{T}} \; \overline{\texttt{f}}; \; \ldots\}}{\mathit{fields}(\texttt{D}) = \overline{\texttt{T}} \; \overline{\texttt{f}}} \qquad \frac{\texttt{class D wraps I } \{\texttt{I g}; \; \ldots\}}{\mathit{comp}(\texttt{D}) = \texttt{I g}}$$

$$\frac{\texttt{class C extends C}' \texttt{ implements I}\{\overline{\texttt{T}} \; \overline{\texttt{f}}; \; \texttt{K}; \; \overline{\texttt{M}}\} \qquad \mathit{fields}(\texttt{C}') = \overline{\texttt{T}}' \; \overline{\texttt{g}}}{\mathit{fields}(\texttt{C}) = \overline{\texttt{T}}' \; \overline{\texttt{g}}, \; \overline{\texttt{T}} \; \overline{\texttt{f}}}$$

$$\frac{\texttt{class C extends C}' \texttt{ implements } \overline{\texttt{I}}\{\overline{\texttt{T}} \; \overline{\texttt{f}}; \; \texttt{K}; \; \overline{\texttt{M}}\} \quad \overline{\texttt{M}} = \{\texttt{T}_j \; \texttt{m}_j \; (\overline{\texttt{T}}'_j \; \overline{\texttt{x}}_j)\{\ldots\}^{\; j \in J}\}}{\mathit{sign}(\texttt{C}) = \{\texttt{T}_j \; \texttt{m}_j \; (\overline{\texttt{T}}'_j \; \overline{\texttt{x}}_j)^{\; j \in J}\} \cup \mathit{sign}(\texttt{C}')}$$

$$\frac{\texttt{interface I extends } \overline{\texttt{I}} \; \{\overline{\texttt{SG}}\}}{\mathit{sign}(\texttt{I}) = \{\overline{\texttt{SG}}\} \cup \mathit{sign}(\overline{\texttt{I}})}$$

$$\frac{\texttt{class D wraps I } \{\cdots\}}{\mathit{wrap}(\texttt{D}) = \texttt{I}} \qquad \frac{\texttt{class C extends C}' \texttt{ implements } \overline{\texttt{I}}\{\cdots\}}{\mathit{ord}(\texttt{C})}$$

Figure 5: FWJ: lookup functions

*per method declaration* `after T m` $(\overline{\texttt{T}} \; \overline{\texttt{x}})\{\texttt{e}\}$ defines a method and specifies how that the method body `e` will be executed after the call on the component. In method declarations, the return value is the one produced by reducing `e`.

FWJ basic *expressions* are standard FJ expressions: variables, field selection, method invocation, object creation. Instead, sequences of expressions (`e;e`) and field updating (`e.f=e`) characterize our imperative framework. Finally, `newwrap` $\texttt{D}(\texttt{e}, \overline{\texttt{e}})$ creates a new instance of the wrapper class `D` wrapping the object `e`; the other parameters $\overline{\texttt{e}}$ are required to initialize `D`'s fields.

The *subtyping* relation induced by the class declarations in the program, denoted by $<:$, is formally defined in Figure 4. In FWJ, subtyping is straightforwardly extended to deal with interfaces and wrapper classes.

For the typing and reduction rules, we need a few auxiliary lookup function definitions (Figure 5). The function $\mathit{fields}(\texttt{C})$ applied to a standard class `C` or to a wrapper class `D`, return the fields of the class argument: indeed in the case of a wrapper class `D` we consider only the fields declared in `D`. The special field corresponding to the wrapped object of a wrapper class `D` is written $\mathit{comp}(\texttt{D})$. The lookup function of method signatures for classes and interfaces is *sign* (Figure 5). Here again, $\mathit{sign}(\overline{\texttt{I}})$ denotes $\mathit{sign}(\texttt{I}_1) \cup \ldots \cup \mathit{sign}(\texttt{I}_n)$. The function $\mathit{wrap}(\texttt{D})$ denotes the interface wrapped by the wrapper class `D` and the predicate $\mathit{ord}(\texttt{E})$ returns true if `E` is an ordinary class. Notice that $\mathit{wrap}(\texttt{E})$ is defined if and only if $!\mathit{ord}(\texttt{E})$; however, having both $\mathit{wrap}(\texttt{E})$ and $\mathit{ord}(\texttt{E})$ makes presentation of rules cleaner.

$$\frac{\texttt{class C extends C}' \texttt{ implements } \overline{\texttt{I}}\{\overline{\texttt{T}}\ \overline{\texttt{f}};\ \texttt{K};\ \overline{\texttt{M}}\} \qquad \texttt{T m}\ (\overline{\texttt{T}}'\ \overline{\texttt{x}})\{\texttt{e}\} \in \overline{\texttt{M}}}{\mathit{mBody}(\texttt{m},\texttt{C}) = (\overline{\texttt{x}},\ \texttt{e})}$$

$$\frac{\texttt{class C extends C}' \texttt{ implements } \overline{\texttt{I}}\{\overline{\texttt{T}}\ \overline{\texttt{f}};\ \texttt{K};\ \overline{\texttt{M}}\} \qquad \texttt{m} \notin \overline{\texttt{M}}}{\mathit{mBody}(\texttt{m},\texttt{C}) = \mathit{mBody}(\texttt{m},\texttt{C}')}$$

$$\frac{\texttt{class D wraps I}\ \{\texttt{I g};\ \overline{\texttt{T}}\ \overline{\texttt{f}};\ \texttt{KD};\ \overline{\texttt{M}}\ \overline{\texttt{N}}\} \qquad \texttt{after T m}\ (\overline{\texttt{T}}'\ \overline{\texttt{x}})\{\texttt{e}\} \in \overline{\texttt{N}}}{\mathit{mBody}(\texttt{m},\texttt{D}) = (\overline{\texttt{x}},\ \texttt{e})}$$

$$\frac{\texttt{class D wraps I}\ \{\texttt{I g};\ \overline{\texttt{T}}\ \overline{\texttt{f}};\ \texttt{KD};\ \overline{\texttt{M}}\ \overline{\texttt{N}}\} \qquad \texttt{m} \notin \overline{\texttt{N}} \qquad \texttt{T m}\ (\overline{\texttt{T}}'\ \overline{\texttt{x}}) \in \mathit{sign}(\texttt{I})}{\mathit{mBody}(\texttt{m},\texttt{D}) = (\overline{\texttt{x}},\texttt{ret})}$$

$$\frac{\texttt{class D wraps I}\ \{\texttt{I g};\ \overline{\texttt{T}}\ \overline{\texttt{f}};\ \texttt{KD};\ \overline{\texttt{M}}\ \overline{\texttt{N}}\} \qquad \texttt{T m}\ (\overline{\texttt{T}}'\ \overline{\texttt{x}})\{\texttt{e}\} \in \overline{\texttt{M}}}{\mathit{mBody}(\texttt{m},\texttt{D}) = (\overline{\texttt{x}},\ \texttt{e})}$$

$$\frac{\texttt{class C extends C}' \texttt{ implements } \overline{\texttt{I}}\{\cdots\} \qquad \texttt{T m}\ (\overline{\texttt{T}}\ \overline{\texttt{x}}) \in \mathit{sign}(\texttt{C})}{\mathit{mType}(\texttt{m},\texttt{C}) = \overline{\texttt{T}} \to \texttt{T}}$$

$$\frac{\texttt{interface I extends } \overline{\texttt{I}}\{\cdots\} \qquad \texttt{T m}\ (\overline{\texttt{T}}\ \overline{\texttt{x}}) \in \mathit{sign}(\texttt{I})}{\mathit{mType}(\texttt{m},\texttt{I}) = \overline{\texttt{T}} \to \texttt{T}}$$

$$\frac{\texttt{class D wraps I}\ \{\texttt{I g};\ \overline{\texttt{T}}\ \overline{\texttt{f}};\ \texttt{KD};\ \overline{\texttt{M}}\ \overline{\texttt{N}}\} \qquad \texttt{T m}\ (\overline{\texttt{T}}'\ \overline{\texttt{x}}) \in \mathit{sign}(\texttt{I})}{\mathit{mType}(\texttt{m},\texttt{D}) = \overline{\texttt{T}}' \to \texttt{T}}$$

$$\frac{\texttt{class D wraps I}\ \{\texttt{I g};\ \overline{\texttt{T}}\ \overline{\texttt{f}};\ \texttt{KD};\ \overline{\texttt{M}}\ \overline{\texttt{N}}\} \qquad \texttt{T m}\ (\overline{\texttt{T}}'\ \overline{\texttt{x}})\{\texttt{e}\} \in \overline{\texttt{M}}}{\mathit{mType}(\texttt{m},\texttt{D}) = \overline{\texttt{T}}' \to \texttt{T}}$$

Figure 6: FWJ: lookup functions for methods.

$$\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x}) \qquad \text{(T-VAR)} \qquad \frac{\Gamma \vdash \mathtt{e} : \mathtt{E} \qquad \mathit{fields}(\mathtt{E}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\Gamma \vdash \mathtt{e}.\mathtt{f}_i : \mathtt{T}_i} \qquad \text{(T-FIELD)}$$

$$\frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{E}_0 \qquad \mathit{mType}(\mathtt{m}, \mathtt{E}_0) = \overline{\mathtt{T}} \to \mathtt{T} \qquad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}}' \qquad \overline{\mathtt{T}}' <: \overline{\mathtt{T}}}{\Gamma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) : \mathtt{T}} \qquad \text{(T-INVK)}$$

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}}' \qquad \overline{\mathtt{T}}' <: \overline{\mathtt{T}}}{\Gamma \vdash \mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}}) : \mathtt{C}} \qquad \text{(T-NEW)}$$

$$\frac{\begin{array}{lll} \mathit{comp}(\mathtt{D}) = \mathtt{I}\ \mathtt{g} & \Gamma \vdash \mathtt{e} : \mathtt{E} & \mathtt{E} <: \mathtt{I} \\ \mathit{fields}(\mathtt{D}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} & \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}}' & \overline{\mathtt{T}}' <: \overline{\mathtt{T}} \end{array}}{\Gamma \vdash \mathtt{newwrap}\ \mathtt{D}(\mathtt{e}, \overline{\mathtt{e}}) : \mathtt{D}} \qquad \text{(T-NEWWRAP)}$$

$$\frac{\Gamma \vdash \mathtt{e}_1 : \mathtt{T}' \qquad \Gamma \vdash \mathtt{e}_2 : \mathtt{T}''}{\Gamma \vdash \mathtt{e}_1; \mathtt{e}_2 : \mathtt{T}''} \qquad \text{(T-SEQ)} \qquad \frac{\Gamma \vdash \mathtt{e}_1.\mathtt{f} : \mathtt{T} \qquad \Gamma \vdash \mathtt{e}_2 : \mathtt{T}' \qquad \mathtt{T}' <: \mathtt{T}}{\Gamma \vdash \mathtt{e}_1.\mathtt{f} = \mathtt{e}_2 : \mathtt{void}} \qquad \text{(T-ASSIGN)}$$

Figure 7: FWJ: typing rules for expressions

The function *mBody* (Figure 6) returns the body of the method `m` in class `C` as a pair of a sequence of parameters $\overline{\mathtt{x}}$ and an expression `e`. Notice that, since inheritance is not allowed for wrapper classes, we do not search for the body in superclasses (as for standard classes); instead when a wrapper class does not define a method belonging to the wrapped interface *mBody* returns `ret`. The lookup function for method types, *mType*, uses *sign* for classes and interfaces; in case of wrapper classes, if a wrapped method is not defined in the wrapper class, it is searched for in the wrapped interface.

## FWJ: typing

An *environment* $\Gamma$ is a finite mapping from variables to types, written $\overline{\mathtt{x}} : \overline{\mathtt{T}}$. The syntax of types is extended with `void`, the type of expressions that do not return a value, i.e., assignments. The typing judgment for expressions has the form $\Gamma \vdash \mathtt{e} : \mathtt{T}$, read "in the environment $\Gamma$, expression `e` has type `T`". We abbreviate typing judgment on sequences in the obvious way, writing $\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}}$ as shorthand for $\Gamma \vdash \mathtt{e}_1 : \mathtt{T}_1, \ldots, \Gamma \vdash \mathtt{e}_n : \mathtt{T}_n$, and writing $\overline{\mathtt{T}} <: \overline{\mathtt{T}}'$ as shorthand for $\mathtt{T}_1 <: \mathtt{T}'_1, \ldots, \mathtt{T}_n <: \mathtt{T}'_n$. The typing rules are syntax directed, with one rule for each form of expression.

Typing rules for expressions (Figure 7) are straightforward: (T-FIELD), (T-INVK) and (T-NEW) are as in FJ and (T-SEQ) and (T-ASSIGN) are standard in an imperative setting. The novel rule is (T-NEWWRAP) that also checks that the type of the actual parameter `e` associated to the component, is a subtype of the interface `I` wrapped by the constructor's class `D`. Moreover, this rule assigns to the wrapped object the type of the wrapper.

The rule for methods (Figure 8) in ordinary classes (M-OK), besides the standard check that the type of the body is a subtype of the declared return type, checks that, in the case of overriding, if a method with the same name is declared in the superclass, then it has the same type (this corresponds to the *override* predicate in [29]). Similarly, if the signature of a method occurs in some of the implemented interfaces, then it must

$$\frac{\texttt{class C extends C}' \texttt{ implements } \overline{\texttt{I}}\ \{\cdots\} \qquad \overline{\texttt{x}} : \overline{\texttt{T}}, \texttt{this} : \texttt{C} \vdash \texttt{e} : \texttt{T}_0 \qquad \texttt{T}_0 <: \texttt{T} \\ \text{if } \mathit{mType}(\texttt{m}, \texttt{C}') = \overline{\texttt{T}}' \to \texttt{T}', \text{ then } \overline{\texttt{T}}' = \overline{\texttt{T}} \ \wedge\ \texttt{T}' = \texttt{T} \\ \forall \texttt{I}_j \in \overline{\texttt{I}}, \text{ if } \mathit{mType}(\texttt{m}, \texttt{I}_j) = \overline{\texttt{T}}'' \to \texttt{T}'', \text{ then } \overline{\texttt{T}}'' = \overline{\texttt{T}} \ \wedge\ \texttt{T}'' = \texttt{T}}{\texttt{T m } (\overline{\texttt{T}}\ \overline{\texttt{x}})\{\texttt{e}\} \text{ OK IN C}} \quad \text{(M-OK)}$$

$$\frac{\texttt{class D wraps I}\ \{\cdots\} \qquad \overline{\texttt{x}} : \overline{\texttt{T}}, \texttt{this} : \texttt{D}, \texttt{ret} : \texttt{T} \vdash \texttt{e} : \texttt{T}_0 \qquad \texttt{T}_0 <: \texttt{T} \\ \mathit{mType}(\texttt{m}, \texttt{D}) = \mathit{mType}(\texttt{m}, \texttt{I})}{\texttt{after T m } (\overline{\texttt{T}}\ \overline{\texttt{x}})\{\texttt{e}\} \text{ OK IN D}} \quad \text{(DM-OK)}$$

$$\frac{\texttt{class D wraps I}\ \{\cdots\} \quad \overline{\texttt{x}} : \overline{\texttt{T}}, \texttt{this} : \texttt{D} \vdash \texttt{e} : \texttt{T}_0 \quad \texttt{T}_0 <: \texttt{T} \quad \texttt{T m } (\overline{\texttt{T}}\ \overline{\texttt{x}}) \notin \mathit{sign}(\texttt{I})}{\texttt{T m } (\overline{\texttt{T}}\ \overline{\texttt{x}})\{\texttt{e}\} \text{ OK IN D}} \quad \text{(DM-OK-2)}$$

$$\frac{\texttt{K} = \texttt{C}(\overline{\texttt{T}}\ \overline{\texttt{g}}, \overline{\texttt{T}}'\ \overline{\texttt{f}})\{\texttt{super}(\overline{\texttt{g}}); \texttt{this.}\overline{\texttt{f}} = \overline{\texttt{f}};\} \qquad \mathit{fields}(\texttt{C}') = \overline{\texttt{T}}\ \overline{\texttt{g}} \\ \mathit{sign}(\overline{\texttt{I}}) \subseteq \mathit{sign}(\texttt{C}) \qquad \overline{\texttt{M}} \text{ OK IN C}}{\texttt{class C extends C}' \texttt{ implements } \overline{\texttt{I}}\{\overline{\texttt{T}}'\ \overline{\texttt{f}};\ \texttt{K};\ \overline{\texttt{M}}\} \text{ OK}} \quad \text{(CLASS-OK)}$$

$$\frac{\texttt{KD} = \texttt{D}(\texttt{I g}, \overline{\texttt{T}}\ \overline{\texttt{f}})\{\texttt{this.g} = \texttt{g}; \texttt{this.}\overline{\texttt{f}} = \overline{\texttt{f}};\} \qquad \overline{\texttt{N}} \text{ OK IN D} \qquad \overline{\texttt{M}} \text{ OK IN D}}{\texttt{class D wraps I}\ \{\texttt{I g};\ \overline{\texttt{T}}\ \overline{\texttt{f}};\ \texttt{KD};\ \overline{\texttt{M}}\ \overline{\texttt{N}}\} \text{ OK}} \quad \text{(DCLASS-OK)}$$

Figure 8: FWJ: typing rules for methods and classes

$$\begin{array}{lcl} \texttt{v} & ::= & \iota \mid \texttt{voidValue} \\ \texttt{e} & ::= & \texttt{e.f} \mid \texttt{e.m}(\overline{\texttt{e}}) \mid \texttt{new C}(\overline{\texttt{e}}) \mid \texttt{newwrap D}(\texttt{e}, \overline{\texttt{e}}) \mid \texttt{e;e} \mid \\ & & \texttt{e.f} = \texttt{e} \mid \texttt{v} \mid \texttt{deleg}(\iota.\texttt{m}(\overline{\iota}), \iota) \mid \texttt{Ret}(\texttt{e}, \texttt{e}, \iota) \end{array}$$

Figure 9: FWJ: runtime values and expressions

have the same type. We define two rules for methods declared in wrapper classes, both similar to the previous one. The rule (DM-OK), for wrapped methods, also inserts the type for `ret` (i.e., the return type of the method itself) in the type environment; moreover, it checks that the wrapped method is actually part of the wrapped interface and that the signature of the wrapped method is the same of the one in the wrapped interface, $\mathit{mType}(\texttt{m}, \texttt{D}) = \mathit{mType}(\texttt{m}, \texttt{I})$. The second one, (DM-OK-2), is for standard methods, i.e., those that do not belong to the wrapped interface. Since there is no inheritance between wrapper classes, differently from rule (M-OK), we do not have to check the signature of possible overridden methods. Rule (CLASS-OK) for ordinary classes is similar to FJ: it checks that the constructor applies `super` to the fields of the superclass and initializes the fields declared in this class, and that each method declaration in the class is OK; moreover, it also checks that the class implements all the methods of its interfaces. Notice that, in rule (DCLASS-OK) for wrapper classes, we do not have to check that methods belonging to the wrapped interface and not implemented in the wrapper class are defined, because they are automatically called on the component.

## FWJ: semantics

In order to properly model imperative features we introduce the concepts of heap $\mathscr{H}$ and addresses $\iota$: $\mathscr{H}$ is a *heap* mapping *addresses* to *objects*; *Addresses*, ranged over by

$$\frac{\mathscr{H}(\iota) = (\mathtt{E}, [\overline{\mathtt{f}} : \overline{\iota}])}{\iota.\mathtt{f}_i \parallel \mathscr{H} \longrightarrow \iota_i \parallel \mathscr{H}} \quad \text{(R-FIELD)}$$

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \iota \notin \mathrm{Dom}(\mathscr{H})}{\mathtt{new}\ \mathtt{C}(\overline{\iota}) \parallel \mathscr{H} \longrightarrow \iota \parallel \mathscr{H} \cup \{\iota \mapsto (\mathtt{C}, [\overline{\mathtt{f}} : \overline{\iota}])\}} \quad \text{(R-NEW)}$$

$$\frac{\mathit{fields}(\mathtt{D}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \mathit{comp}(\mathtt{D}) = \mathtt{I}\ \mathtt{g} \qquad \iota' \notin \mathrm{Dom}(\mathscr{H})}{\mathtt{newwrap}\ \mathtt{D}(\iota, \overline{\iota}) \parallel \mathscr{H} \longrightarrow \iota' \parallel \mathscr{H} \cup \{\iota' \mapsto (\mathtt{D}, [\mathtt{g} : \iota, \overline{\mathtt{f}} : \overline{\iota}])\}} \quad \text{(R-NEWWRAP)}$$

$$\mathtt{v}; \mathtt{e} \parallel \mathscr{H} \longrightarrow \mathtt{e} \parallel \mathscr{H} \quad \text{(R-SEQ)}$$

$$\frac{\mathscr{H}(\iota) = (\mathtt{E}, [..., \mathtt{f}_i : \iota_i, ...])}{\iota.\mathtt{f}_i = \iota_i' \parallel \mathscr{H} \longrightarrow \mathtt{voidValue} \parallel \mathscr{H}[\mathscr{H}(\iota) \mapsto (\mathtt{E}, [..., \mathtt{f}_i : \iota_i', ...])]} \quad \text{(R-ASSIGN)}$$

$$\frac{\mathscr{H}(\iota) = (\mathtt{C}, [\overline{\mathtt{f}} : \overline{\iota}']) \qquad \mathit{ord}(\mathtt{C}) \qquad \mathit{mBody}(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e})}{\iota.\mathtt{m}(\overline{\iota}) \parallel \mathscr{H} \longrightarrow \mathtt{e}\,[\mathtt{this} \leftarrow \iota, \overline{\mathtt{x}} \leftarrow \overline{\iota}] \parallel \mathscr{H}} \quad \text{(R-INVK)}$$

$$\frac{\mathscr{H}(\iota) = (\mathtt{D}, \ldots) \quad \mathit{wrap}(\mathtt{D}) = \mathtt{I} \quad \mathtt{m} \notin \mathit{sign}(\mathtt{I}) \quad \mathit{mBody}(\mathtt{m}, \mathtt{D}) = (\overline{\mathtt{x}}, \mathtt{e})}{\iota.\mathtt{m}(\overline{\iota}) \parallel \mathscr{H} \longrightarrow \mathtt{e}\,[\mathtt{this} \leftarrow \iota, \overline{\mathtt{x}} \leftarrow \overline{\iota}] \parallel \mathscr{H}} \quad \text{(R-DINVK)}$$

$$\frac{\mathscr{H}(\iota) = (\mathtt{D}, [\mathtt{g} : \iota', \ldots]) \qquad \mathit{wrap}(\mathtt{D}) = \mathtt{I} \qquad \mathtt{m} \in \mathit{sign}(\mathtt{I})}{\iota.\mathtt{m}(\overline{\iota}) \parallel \mathscr{H} \longrightarrow \mathtt{Ret}(\mathtt{deleg}(\iota'.\mathtt{m}(\overline{\iota}), \iota), \iota.\mathtt{m}(\overline{\iota}), \iota) \parallel \mathscr{H}} \quad \text{(R-DA-INVK)}$$

$$\frac{\mathscr{H}(\iota') = (\mathtt{C}, \ldots) \quad \mathit{ord}(\mathtt{C}) \quad \mathit{mBody}(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e}) \quad \mathscr{H}(\iota) = (\mathtt{D}, \ldots) \quad \mathit{wrap}(\mathtt{D}) = \mathtt{I}}{\mathtt{deleg}(\iota'.\mathtt{m}(\overline{\iota}), \iota) \parallel \mathscr{H} \longrightarrow ((\mathtt{e}[\mathtt{this} \Leftarrow_{\mathtt{I}} \iota])\,[\mathtt{this} \leftarrow \iota', \overline{\mathtt{x}} \leftarrow \overline{\iota}]) \parallel \mathscr{H}} \quad \text{(R-DEL-1)}$$

$$\frac{\mathscr{H}(\iota') = (\mathtt{D}, [\mathtt{g} : \iota'', \ldots]) \qquad !\mathit{ord}(\mathtt{D})}{\mathtt{deleg}(\iota'.\mathtt{m}(\overline{\iota}), \iota) \parallel \mathscr{H} \longrightarrow \mathtt{Ret}(\mathtt{deleg}(\iota''.\mathtt{m}(\overline{\iota}), \iota), \iota'.\mathtt{m}(\overline{\iota}), \iota) \parallel \mathscr{H}} \quad \text{(R-DEL-2)}$$

$$\frac{\mathscr{H}(\iota') = (\mathtt{D}, \ldots) \qquad \mathit{wrap}(\mathtt{D}) = \mathtt{I} \qquad \mathit{mBody}(\mathtt{m}, \mathtt{D}) = (\overline{\mathtt{x}}, \mathtt{e})}{\mathtt{Ret}(\iota, \iota'.\mathtt{m}(\overline{\iota}), \iota'') \parallel \mathscr{H} \longrightarrow ((\mathtt{e}[\mathtt{this} \Leftarrow_{\mathtt{I}} \iota''])\,[\mathtt{this} \leftarrow \iota', \mathtt{ret} \leftarrow \iota, \overline{\mathtt{x}} \leftarrow \overline{\iota}]) \parallel \mathscr{H}} \quad \text{(R-RET)}$$

Figure 10: FWJ: computation rules

the metavariable $\iota$, are the elements of the denumerable set $\mathbf{I}$; *Objects* are denoted by $(\mathtt{E}, [\mathtt{f}_1 : \iota_1, \ldots, \mathtt{f}_p : \iota_p])$, where $\mathtt{E}$ is the class of the object and $[\mathtt{f}_1 : \iota_1, \ldots, \mathtt{f}_p : \iota_p]$ maps field names to addresses. The result of a term evaluation depends on the heap thus we define the operational semantics of FWJ in terms of transitions between configurations of the shape $\mathtt{e} \parallel \mathscr{H}$ where $\mathtt{e}$ is a runtime expression, i.e., the code under evaluation. *Runtime expressions* (see Fig. 9) are defined starting from to the grammar defining expressions by adding *values*, removing variables, and adding $\mathtt{deleg}(\iota'.\mathtt{m}(\overline{\iota}), \iota)$ and $\mathtt{Ret}(\mathtt{e}, \iota'.\mathtt{m}(\overline{\iota}), \iota)$. A *value* is either an address or the value `voidValue` which represents the outcome of the evaluation of redexes that do not return a value, like the assignment redex $\iota.\mathtt{f} = \iota'$.

The reduction relation is of the form "$\mathtt{e} \parallel \mathscr{H} \longrightarrow \mathtt{e}' \parallel \mathscr{H}'$", read "configuration $\mathtt{e} \parallel \mathscr{H}$ reduces to configuration $\mathtt{e}' \parallel \mathscr{H}'$ in one step". We write $\longrightarrow^\star$ for the reflexive and transitive closure of $\longrightarrow$. The reduction rules, by using the standard notions of computation rules (Figure 10) and congruence rules (Figure 11), ensure that the computation is carried on according to a call-by-value reduction strategy. Rule R-NEW for object creation stores the newly created object at a fresh address of the heap and returns the address. Object's

$$\frac{\mathtt{e} \parallel \mathscr{H} \longrightarrow \mathtt{e}' \parallel \mathscr{H}'}{\mathtt{e.f} \parallel \mathscr{H} \longrightarrow \mathtt{e}'\mathtt{.f} \parallel \mathscr{H}'} \qquad \frac{\mathtt{e} \parallel \mathscr{H} \longrightarrow \mathtt{e}' \parallel \mathscr{H}'}{\mathtt{e.m}(\overline{\mathtt{e}}) \parallel \mathscr{H} \longrightarrow \mathtt{e}'\mathtt{.m}(\overline{\mathtt{e}}) \parallel \mathscr{H}'}$$

$$\frac{\mathtt{e}_i \parallel \mathscr{H} \longrightarrow \mathtt{e}_i' \parallel \mathscr{H}'}{\iota_0\mathtt{.m}(\overline{\iota}, \mathtt{e}_i, \overline{\mathtt{e}}) \parallel \mathscr{H} \longrightarrow \iota_0\mathtt{.m}(\overline{\iota}, \mathtt{e}_i', \overline{\mathtt{e}}) \parallel \mathscr{H}'} \qquad \frac{\mathtt{e} \parallel \mathscr{H} \longrightarrow \mathtt{e}' \parallel \mathscr{H}'}{\mathtt{Ret}(\mathtt{e}, \iota\mathtt{.m}(\overline{\iota}), \iota') \parallel \mathscr{H} \longrightarrow \mathtt{Ret}(\mathtt{e}', \iota\mathtt{.m}(\overline{\iota}), \iota') \parallel \mathscr{H}'}$$

$$\frac{\mathtt{e}_i \parallel \mathscr{H} \longrightarrow \mathtt{e}_i' \parallel \mathscr{H}'}{\mathtt{new\ C}(\overline{\iota}, \mathtt{e}_i, \overline{\mathtt{e}}) \parallel \mathscr{H} \longrightarrow \mathtt{new\ C}(\overline{\iota}, \mathtt{e}_i', \overline{\mathtt{e}}) \parallel \mathscr{H}'} \qquad \frac{\mathtt{e}_i \parallel \mathscr{H} \longrightarrow \mathtt{e}_i' \parallel \mathscr{H}'}{\mathtt{newwrap\ D}(\overline{\iota}, \mathtt{e}_i, \overline{\mathtt{e}}) \parallel \mathscr{H} \longrightarrow \mathtt{newwrap\ D}(\overline{\iota}, \mathtt{e}_i', \overline{\mathtt{e}}) \parallel \mathscr{H}'}$$

$$\frac{\mathtt{e}_1 \parallel \mathscr{H} \longrightarrow \mathtt{e}_1' \parallel \mathscr{H}'}{\mathtt{e}_1;\mathtt{e}_2 \parallel \mathscr{H} \longrightarrow \mathtt{e}_1';\mathtt{e}_2 \parallel \mathscr{H}'} \qquad \frac{\mathtt{e}_1 \parallel \mathscr{H} \longrightarrow \mathtt{e}_1' \parallel \mathscr{H}'}{\mathtt{e}_1\mathtt{.f} = \mathtt{e}_2 \parallel \mathscr{H} \longrightarrow \mathtt{e}_1'\mathtt{.f} = \mathtt{e}_2 \parallel \mathscr{H}'}$$

$$\frac{\mathtt{e}_2 \parallel \mathscr{H} \longrightarrow \mathtt{e}_2' \parallel \mathscr{H}'}{\iota_1\mathtt{.f} = \mathtt{e}_2 \parallel \mathscr{H} \longrightarrow \iota_1\mathtt{.f} = \mathtt{e}_2' \parallel \mathscr{H}'}$$

Figure 11: FWJ: congruence rules

fields are initialized as specified by the class constructor. Similarly for the wrapper object creation (rule R-NEWWRAP), where also the special field corresponding to the wrapped object is initialized.

Concerning invocation of a method m on a receiver $\iota$ with actual parameter values $\overline{\iota}$ we distinguish the following cases: (1) the object at $\iota$ is an instance of an ordinary class (R-INVK) or is a wrapper object and m is not in the wrapped interface (R-DINVK): $\iota$.m($\overline{\iota}$) is replaced with the evaluation of the body of m; (2) the object at $\iota$ is a wrapper object and m is in the wrapped interface (R-DA-INVK): evaluates the body of m at $\iota$ after having invoked m by delegation on the wrapped object $\iota'$ (using deleg and Ret, as explained in the following). Note that, when the method m is in the wrapped interface but it is not implemented in the wrapper class D, the invocation of m on D reduces to ret (see *mBody*(m,D) in Figure 6). This is to model the fact that methods not implemented in a wrapper class are forwarded to the wrapped component.

In order to implement the chain of invocation of wrapper and component methods by delegation, we need the runtime expressions deleg($\iota'$.m($\overline{\iota}$), $\iota$) and Ret(e, $\iota'$.m($\overline{\iota}$), $\iota$). We first define [this $\Leftarrow_{\mathtt{I}}$ $\iota$] (Definition 2.1) as a new form of substitution that, when applied to an expression e, replaces every occurrence of this.m($\overline{\mathtt{e}}$), such that m $\in$ *sign*(I), with $\iota$.m($\overline{\mathtt{e}}$).

**Definition 2.1 (Substitution this $\Leftarrow_{\mathtt{I}}$ $\iota$)** *We define the substitution* this $\Leftarrow_{\mathtt{I}}$ $\iota$ *on FWJ expressions as follows:*

- x [this $\Leftarrow_{\mathtt{I}}$ $\iota$] = x
- this.m($\overline{\mathtt{e}}$) [this $\Leftarrow_{\mathtt{I}}$ $\iota$] = $\iota$.m($\overline{\mathtt{e}}$ [this $\Leftarrow_{\mathtt{I}}$ $\iota$]) *if* m $\in$ *sign*(I)
- this.m($\overline{\mathtt{e}}$) [this $\Leftarrow_{\mathtt{I}}$ $\iota$] = this.m($\overline{\mathtt{e}}$ [this $\Leftarrow_{\mathtt{I}}$ $\iota$]) *if* m $\notin$ *sign*(I)
- e.m($\overline{\mathtt{e}}$) [this $\Leftarrow_{\mathtt{I}}$ $\iota$] = e [this $\Leftarrow_{\mathtt{I}}$ $\iota$].m($\overline{\mathtt{e}}$ [this $\Leftarrow_{\mathtt{I}}$ $\iota$]) *if* e $\neq$ this
- e.f [this $\Leftarrow_{\mathtt{I}}$ $\iota$] = e [this $\Leftarrow_{\mathtt{I}}$ $\iota$].f
- new C($\overline{\mathtt{e}}$) [this $\Leftarrow_{\mathtt{I}}$ $\iota$] = new C($\overline{\mathtt{e}}$ [this $\Leftarrow_{\mathtt{I}}$ $\iota$])
- e$_1$;e$_2$ [this $\Leftarrow_{\mathtt{I}}$ $\iota$] = e$_1$ [this $\Leftarrow_{\mathtt{I}}$ $\iota$];e$_2$ [this $\Leftarrow_{\mathtt{I}}$ $\iota$]

- $\texttt{newwrap D}(\texttt{e},\overline{\texttt{e}})\,[\texttt{this} \Leftarrow_{\texttt{I}} \iota] \;=\; \texttt{newwrap D}(\texttt{e}\,[\texttt{this} \Leftarrow_{\texttt{I}} \iota], \overline{\texttt{e}}\,[\texttt{this} \Leftarrow_{\texttt{I}} \iota])$
- $\texttt{e}_1.\texttt{f} = \texttt{e}_2\,[\texttt{this} \Leftarrow_{\texttt{I}} \iota] \;=\; \texttt{e}_1\,[\texttt{this} \Leftarrow_{\texttt{I}} \iota].\texttt{f} = \texttt{e}_2\,[\texttt{this} \Leftarrow_{\texttt{I}} \iota]$

The redex $\texttt{Ret}(\texttt{e}, \iota'.\texttt{m}(\overline{\iota}), \iota)$ evaluates e until it reaches a value (an identifier); after that, it evaluates the method invocation $\iota'.\texttt{m}(\overline{\iota})$ by using the substitution $\texttt{this} \Leftarrow_{\texttt{I}} \iota$ and by also replacing ret with the value produced by e. This implements the fact that the method of the wrapper is called after the method on the component returned, and ret is automatically updated. The unfolding of method invocation by delegation in a wrapper composition is performed by the redex $\texttt{deleg}(\iota'.\texttt{m}(\overline{\iota}), \iota)$: when $\iota'$ is itself a wrapper object, then there is a recursive call on the component object stored at $\iota''$, which is wrapped by $\iota'$ (rule R-DEL-2) by leaving an outer Ret redex. Otherwise, the recursion terminates by evaluating the body of the method in the ordinary class (still using $\texttt{this} \Leftarrow_{\texttt{I}} \iota$). Only at that point, the outer Rets will be reduced.

Let us show an example of method invocation with wrappers, to see how deleg and Ret work together. Suppose you have the following (well-typed) method invocation, $\texttt{newwrap D}_1(\texttt{newwrap D}_2(\texttt{new C}())).\texttt{m}()$, (where for simplicity we do not use parameters nor class fields) and let us see how it evolves by using the above semantic rules (in the following reduction steps we denote the body of m in the class E with $\texttt{e}_\texttt{E}$ and its final result with $\iota'_\texttt{E}$; while we denote the identifier of the object of class E with $\iota_\texttt{E}$):

$\texttt{newwrap D}_1(\texttt{newwrap D}_2(\texttt{new C}())).\texttt{m}() \parallel \emptyset \longrightarrow$
$\texttt{newwrap D}_1(\texttt{newwrap D}_2(\iota_\texttt{C})).\texttt{m}() \parallel \{\iota_\texttt{C} \mapsto (\texttt{C})\} \longrightarrow$
$\texttt{newwrap D}_1(\iota_{\texttt{D}_2}).\texttt{m}() \parallel \{\iota_\texttt{C} \mapsto (\texttt{C}), \iota_{\texttt{D}_2} \mapsto (\texttt{D}_2, [\texttt{g} : \iota_\texttt{C}])\} \longrightarrow$
$\iota_{\texttt{D}_1}.\texttt{m}() \parallel \{\iota_\texttt{C} \mapsto (\texttt{C}), \iota_{\texttt{D}_2} \mapsto (\texttt{D}_2, [\texttt{g} : \iota_\texttt{C}]), \iota_{\texttt{D}_1} \mapsto (\texttt{D}_1, [\texttt{g} : \iota_{\texttt{D}_2}])\} \longrightarrow$
$(\text{denote } \mathscr{H} = \{\iota_\texttt{C} \mapsto (\texttt{C}), \iota_{\texttt{D}_2} \mapsto (\texttt{D}_2, [\texttt{g} : \iota_\texttt{C}]), \iota_{\texttt{D}_1} \mapsto (\texttt{D}_1, [\texttt{g} : \iota_{\texttt{D}_2}])\})$
$\texttt{Ret}(\texttt{deleg}(\iota_{\texttt{D}_2}.\texttt{m}(), \iota_{\texttt{D}_1}), \iota_{\texttt{D}_1}.\texttt{m}(), \iota_{\texttt{D}_1}) \parallel \mathscr{H} \longrightarrow$
$\texttt{Ret}(\texttt{Ret}(\texttt{deleg}(\iota_\texttt{C}.\texttt{m}(), \iota_{\texttt{D}_1}), \iota_{\texttt{D}_2}.\texttt{m}(), \iota_{\texttt{D}_1}), \iota_{\texttt{D}_1}.\texttt{m}(), \iota_{\texttt{D}_1}) \parallel \mathscr{H} \longrightarrow$
$\texttt{Ret}(\texttt{Ret}((\texttt{e}_\texttt{C}[\texttt{this} \Leftarrow_{\texttt{I}} \iota_{\texttt{D}_1}])\,[\texttt{this} \leftarrow \iota_\texttt{C}], \iota_{\texttt{D}_2}.\texttt{m}(), \iota_{\texttt{D}_1}), \iota_{\texttt{D}_1}.\texttt{m}(), \iota_{\texttt{D}_1}) \parallel \mathscr{H} \longrightarrow^\star$
$(\text{assume } (\texttt{e}_\texttt{C}[\texttt{this} \Leftarrow_{\texttt{I}} \iota_{\texttt{D}_1}])\,[\texttt{this} \leftarrow \iota_\texttt{C}] \parallel \mathscr{H} \;\longrightarrow^\star \iota'_\texttt{C} \parallel \mathscr{H}')$
$\texttt{Ret}(\texttt{Ret}(\iota'_\texttt{C}, \iota_{\texttt{D}_2}.\texttt{m}(), \iota_{\texttt{D}_1}), \iota_{\texttt{D}_1}.\texttt{m}(), \iota_{\texttt{D}_1}) \parallel \mathscr{H}' \longrightarrow$
$\texttt{Ret}((\texttt{e}_{\texttt{D}_2}[\texttt{this} \Leftarrow_{\texttt{I}} \iota_{\texttt{D}_1}])\,[\texttt{this} \leftarrow \iota_{\texttt{D}_2}, \texttt{ret} \leftarrow \iota'_\texttt{C}], \iota_{\texttt{D}_1}.\texttt{m}(), \iota_{\texttt{D}_1}) \parallel \mathscr{H}' \longrightarrow^\star$
$(\text{assume } (\texttt{e}_{\texttt{D}_2}[\texttt{this} \Leftarrow_{\texttt{I}} \iota_{\texttt{D}_1}])\,[\texttt{this} \leftarrow \iota_{\texttt{D}_2}, \texttt{ret} \leftarrow \iota'_\texttt{C}] \parallel \mathscr{H}' \;\longrightarrow^\star \iota'_{\texttt{D}_2} \parallel \mathscr{H}'')$
$\texttt{Ret}(\iota'_{\texttt{D}_2}, \iota_{\texttt{D}_1}.\texttt{m}(), \iota_{\texttt{D}_1}) \parallel \mathscr{H}'' \longrightarrow$
$(\texttt{e}_{\texttt{D}_1}[\texttt{this} \Leftarrow_{\texttt{I}} \iota_{\texttt{D}_1}])\,[\texttt{this} \leftarrow \iota_{\texttt{D}_1}, \texttt{ret} \leftarrow \iota'_{\texttt{D}_2}] \parallel \mathscr{H}'' \longrightarrow \ldots$

Notice how deleg ensures that the methods in wrappers and components are executed in the correct order, and how Ret correctly updates the value of ret.

## 3 SOUNDNESS

In this section we prove that our language extension enjoys type safety. In order to formally state and prove this result we have to define the notions of *well typed runtime expression* and *well formed heap*. We consider the environment $\Gamma$ to be extended to map also addresses to types, written $\overline{\iota} : \overline{\texttt{T}}$. The typing judgment for runtime expressions has

**Typing rules for runtime expressions:**

$$\Gamma \vdash \iota : \Gamma(\iota) \qquad \text{(T-RE-ADDR)}$$

$$\Gamma \vdash \texttt{voidValue} : \texttt{void} \qquad \text{(T-RE-VOID)}$$

$$\frac{\Gamma \vdash \iota'.\texttt{m}(\overline{\iota}) : \texttt{T} \qquad \Gamma \vdash \iota : \texttt{D} \qquad \Gamma \vdash \iota' : \texttt{E} \qquad \texttt{E} <: \mathit{wrap}(\texttt{D})}{\Gamma \vdash \texttt{deleg}(\iota'.\texttt{m}(\overline{\iota}), \iota) : \texttt{T}} \qquad \text{(T-RE-DELEG)}$$

$$\frac{\Gamma \vdash \texttt{e} : \texttt{T} \qquad \Gamma \vdash \iota.\texttt{m}(\overline{\iota}) : \texttt{T} \qquad \Gamma \vdash \iota : \texttt{D} \qquad \Gamma \vdash \iota' : \texttt{E} \qquad \texttt{E} <: \mathit{wrap}(\texttt{D})}{\Gamma \vdash \texttt{Ret}(\texttt{e}, \iota'.\texttt{m}(\overline{\iota}), \iota) : \texttt{T}} \qquad \text{(T-RE-RET)}$$

**Well-formed heap:**

$$\frac{\begin{array}{l}\mathrm{Dom}(\mathscr{H}) \subseteq \mathrm{Dom}(\Gamma) \\ \forall \iota \in \mathrm{Dom}(\mathscr{H}), \\ \quad \text{if } \mathscr{H}(\iota) = (\texttt{E}, [\texttt{f}_1 : \iota_1, \cdots, \texttt{f}_n : \iota_n]), \\ \quad \text{then } \Gamma(\iota) = \texttt{E} \\ \quad \wedge \left( \begin{array}{l} (\mathit{fields}(\texttt{E}) = \texttt{T}_1\ \texttt{f}_1, \cdots, \texttt{T}_n\ \texttt{f}_n) \ \vee \\ (\mathit{comp}(\texttt{E}) = \texttt{T}_1\ \texttt{f}_1, \mathit{fields}(\texttt{E}) = \texttt{T}_2\ \texttt{f}_2, \cdots, \texttt{T}_n\ \texttt{f}_n) \end{array} \right) \\ \quad \wedge\ \forall\ \texttt{T}_i\ \texttt{f}_i,\ 1 \leq i \leq n,\ \ \Gamma(\iota_i) <: \texttt{T}_i \end{array}}{\Gamma \models \mathscr{H}} \qquad \text{(WF-HEAP)}$$

Figure 12: Typing rules for runtime expressions and well-formed heap

the form $\Gamma \vdash \texttt{e} : \texttt{T}$, read "in the environment $\Gamma$, runtime expression e has type T". Most of typing rules have already been given in Figure 7 while typing rules for addresses, $\texttt{deleg}(\iota'.\texttt{m}(\overline{\iota}), \iota)$ and $\texttt{Ret}(\texttt{e}, \iota'.\texttt{m}(\overline{\iota}), \iota)$ are given in Figure 12. Rule (T-RE-DELEG) checks method invocation and also checks that the class of the receiver implements the interface of the second parameter of deleg: this condition will ensure that the substitution $\texttt{this} \Leftarrow_{\texttt{I}} \iota$ (Definition 2.1) is type safe, when applied by the computation rule (R-DEL-1). Rule (T-RE-RET) is similar, but it also checks that the first parameter (the expression e) has the same type of the method invocation: this reflects the fact that the value produced by e will be substituted to ret (computation rule (R-RET)). The judgment for well formed heap (Figure 12) has the form $\Gamma \models \mathscr{H}$, read "heap $\mathscr{H}$ is well formed with respect to environment $\Gamma$". The associated rule ensures that the environment $\Gamma$ maps all the addresses defined in the heap into the type of the corresponding objects. It also ensures that the fields of all the objects stored in the heap contain appropriate values. We notice that the typing rules are syntax directed, with one rule for each form of expression.

The type-soundness theorem (Theorem 3.9) is proved by using the standard technique of subject reduction (Theorem 3.5) and progress (Theorem 3.8). Informally speaking we prove the following result: if e is a well typed closed expression and the configuration "$\texttt{e} \parallel \emptyset$" reduces to a normal form "$\texttt{e}' \parallel \mathscr{H}$", then (1) the heap $\mathscr{H}$ is well formed; (2) the runtime expression $\texttt{e}'$ is well typed and the type of $\texttt{e}'$ is a subtype of the type of e and (3) $\texttt{e}'$ is a value v. We first develop some auxiliary lemmas.

**Lemma 3.1 (Weakening)** *Let* $\Gamma \vdash \texttt{e} : \texttt{T}$. *If* $\Gamma' \supseteq \Gamma$, *then* $\Gamma' \vdash \texttt{e} : \texttt{T}$.

**Proof.** Straightforward induction on the derivation of $\Gamma \vdash \mathtt{e} : \mathtt{T}$. □

**Lemma 3.2** (*mType*) *If* $mType(\mathtt{m}, \mathtt{T}_1) = \overline{\mathtt{T}} \to \mathtt{T}$ *then* $mType(\mathtt{m}, \mathtt{T}_2) = \overline{\mathtt{T}} \to \mathtt{T}$ *for all* $\mathtt{T}_2 <: \mathtt{T}_1$.

**Proof.** Straightforward induction on the derivation of $\mathtt{T}_2 <: \mathtt{T}_1$. □

**Lemma 3.3 (Substitution)** *If* $\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{T}} \vdash \mathtt{e} : \mathtt{T}$ *and* $\Gamma \vdash \overline{\iota} : \overline{\mathtt{T}}'$ *where* $\overline{\mathtt{T}}' <: \overline{\mathtt{T}}$, *then* $\Gamma \vdash \mathtt{e}[\overline{\mathtt{x}} \leftarrow \overline{\iota}] : \mathtt{T}'$ *for some* $\mathtt{T}' <: \mathtt{T}$.

**Proof.** By induction on the derivation of $\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{T}} \vdash \mathtt{e} : \mathtt{T}$. The interesting case is method invocation (the other cases are straightforward).

**Case** $\mathtt{e.m}(\overline{\mathtt{e}})$**.** Since $\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{T}} \vdash \mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{T}$ for some $\mathtt{T}$, we have that

$$\begin{array}{ll} \Gamma, \overline{\mathtt{x}} : \overline{\mathtt{T}} \vdash \mathtt{e} : \mathtt{T}_0 & mType(\mathtt{T}_0, \mathtt{m}) = \overline{\mathtt{T}}' \to \mathtt{T} \\ \Gamma, \overline{\mathtt{x}} : \overline{\mathtt{T}} \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}}_0 & \overline{\mathtt{T}}_0 <: \overline{\mathtt{T}}' \end{array}$$

By induction hypothesis on $\mathtt{e}$ and $\overline{\mathtt{e}}$ we have that $\Gamma \vdash \mathtt{e}[\overline{\mathtt{x}} \leftarrow \overline{\iota}] : \mathtt{T}'_0$ for some $\mathtt{T}'_0 <: \mathtt{T}_0$, and $\Gamma \vdash \overline{\mathtt{e}}[\overline{\mathtt{x}} \leftarrow \overline{\iota}] : \overline{\mathtt{T}}'_0$ for some $\overline{\mathtt{T}}'_0 <: \overline{\mathtt{T}}_0$. From $\mathtt{T}'_0 <: \mathtt{T}_0$ and Lemma 3.2 we have that $mType(\mathtt{T}'_0, \mathtt{m}) = \overline{\mathtt{T}}' \to \mathtt{T}$, and from transitivity of subtyping $\overline{\mathtt{T}}'_0 <: \overline{\mathtt{T}}'$. Applying rule (T-INVK) we get that $\Gamma \vdash (\mathtt{e.m}(\overline{\mathtt{e}}))[\overline{\mathtt{x}} \leftarrow \overline{\iota}] : \mathtt{T}$ which proves the result.

□

**Lemma 3.4 (Substitution** $\mathtt{this} \Leftarrow_{\mathtt{I}} \iota$**)** *If* $\Gamma, \mathtt{this} : \mathtt{E} \vdash \mathtt{e} : \mathtt{T}$ *and* $\Gamma \vdash \iota : \mathtt{D}$ *where* $wrap(\mathtt{D}) = \mathtt{I}$ *and* $\mathtt{E} <: \mathtt{I}$, *then* $\Gamma, \mathtt{this} : \mathtt{E} \vdash \mathtt{e}[\mathtt{this} \Leftarrow_{\mathtt{I}} \iota] : \mathtt{T}$.

**Proof.** The substitution $\mathtt{this} \Leftarrow_{\mathtt{I}} \iota$ (Definition 2.1) substitutes $\iota$ to $\mathtt{this}$ only in expressions $\mathtt{this.m}(\overline{\iota})$ when $\mathtt{m} \in sign(\mathtt{I})$. The result follows from the fact that, since $\mathtt{E}, \mathtt{D} <: \mathtt{I}$, then $\forall \mathtt{n} \in sign(\mathtt{I})\ mType(\mathtt{n}, \mathtt{E}) = mType(\mathtt{n}, \mathtt{D})$ by Lemma 3.2. □

**Theorem 3.5 (Subject reduction)** *If* $\Gamma \models \mathscr{H}$, $\Gamma \vdash \mathtt{e} : \mathtt{T}$ *and* $\mathtt{e} \parallel \mathscr{H} \longrightarrow \mathtt{e}' \parallel \mathscr{H}'$, *then there exists* $\Gamma' \supseteq \Gamma$ *such that* $\Gamma' \models \mathscr{H}'$, $\Gamma' \vdash \mathtt{e}' : \mathtt{T}'$, *for some* $\mathtt{T}'$ *such that* $\mathtt{T}' <: \mathtt{T}$.

**Proof.** The proof is by induction on a derivation of $\mathtt{e} \parallel \mathscr{H} \longrightarrow \mathtt{e}' \parallel \mathscr{H}'$, with a case analysis on the reduction rule used. We proceed by a case analysis on computation rules; for congruence rules simply use the induction hypothesis.

**Case** (R-FIELD)**.** We have that $\iota.\mathtt{f}_i \parallel \mathscr{H} \longrightarrow \iota_i \parallel \mathscr{H}$, where $\mathscr{H}(\iota) = (\mathtt{E}, [\ldots, \mathtt{f}_i : \iota_i, \ldots])$. From $\Gamma \vdash \iota.\mathtt{f}_i : \mathtt{T}_i$ we get $\Gamma \vdash \iota : \mathtt{E}'$ and $fields(\mathtt{E}') = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$. Since $\Gamma \models \mathscr{H}$, then $\mathtt{E}' = \mathtt{E}$ and $\Gamma(\iota_i) <: \mathtt{T}_i$.

**Case** (R-NEWWRAP)**.** We have that

$$\texttt{newwrap}\ \texttt{D}(\iota,\overline{\iota}) \parallel \mathscr{H} \longrightarrow \iota' \parallel \mathscr{H} \cup \{\iota' \mapsto (\texttt{D},[\texttt{g}:\iota,\overline{\texttt{f}}:\overline{\iota}])\},$$

where $\mathit{fields}(\texttt{D}) = \overline{\texttt{T}}\,\overline{\texttt{f}}$, $\mathit{comp}(\texttt{D}) = \texttt{I}\ \texttt{g}$ and $\iota' \notin \mathrm{Dom}(\mathscr{H})$. From rule (T-NEWWRAP), we have that $\Gamma \vdash \iota : \texttt{E} <: \texttt{I}$, $\Gamma \vdash \overline{\iota} : \overline{\texttt{T}}' <: \overline{\texttt{T}}$. Let $\Gamma'$ be such that $\Gamma' = \Gamma \cup \{\iota' : \texttt{D}\}$. Then $\Gamma' \vdash \iota' : \texttt{D}$ and $\Gamma' \models \mathscr{H}'$, where $\mathscr{H}' = \mathscr{H} \cup \{\iota' \mapsto (\texttt{D},[\texttt{g}:\iota,\overline{\texttt{f}}:\overline{\iota}])\}$.

**Case** (R-NEW)**.** Similar to the previous one.

**Case** (R-SEQ)**.** Immediate.

**Case** (R-ASSIGN)**.** We have that

$$\iota.\texttt{f}_i \ = \ \iota_i' \parallel \mathscr{H} \longrightarrow \texttt{voidValue} \parallel \mathscr{H}[\mathscr{H}(\iota) \mapsto (\texttt{E},[\ldots,\texttt{f}_i \ : \iota_i',\ldots])],$$

where $\mathit{fields}(\texttt{E}) = \overline{\texttt{T}}\ \overline{\texttt{f}}$ and $\mathscr{H}(\iota) = (\texttt{E},[\ldots \texttt{f}_i \ : \iota_i \ldots])$. Since $\Gamma \vdash \iota.\texttt{f}_i = \iota_i' : \texttt{void}$, from typing rule (T-ASSIGN) we get $\Gamma \vdash \iota.\texttt{f}_i : \texttt{T}_i$ and $\Gamma \vdash \iota_i' : \texttt{T}_i'$, with $\texttt{T}_i' <: \texttt{T}_i$. Then $\iota_i' : \texttt{T}_i' \in \Gamma$ and $\Gamma \models \mathscr{H}'$, where $\mathscr{H}' = \mathscr{H}[\mathscr{H}(\iota) \mapsto (\texttt{E},[\ldots,\texttt{f}_i \ : \iota_i',\ldots])]$. Moreover $\Gamma \vdash \texttt{voidValue} : \texttt{void}$.

**Case** (R-INVK)**.** The last applied rule is

$$\frac{\mathscr{H}(\iota) = (\texttt{C},[\overline{\texttt{f}} \ : \overline{\iota}'])\qquad \mathit{ord}(\texttt{C})\qquad \mathit{mBody}(\texttt{m},\texttt{C}) = (\overline{\texttt{x}},\texttt{e})}{\iota.\texttt{m}(\overline{\iota}) \parallel \mathscr{H} \longrightarrow \texttt{e}\,[\texttt{this} \leftarrow \iota, \overline{\texttt{x}} \leftarrow \overline{\iota}] \parallel \mathscr{H}}$$

Since $\Gamma \vdash \iota.\texttt{m}(\overline{\iota}) : \texttt{T}$, from typing rule (T-INVK) we get $\Gamma \vdash \iota : \texttt{C}$, $\mathit{mType}(\texttt{m},\texttt{C}) = \overline{\texttt{T}} \to \texttt{T}$, $\Gamma \vdash \overline{\iota} : \overline{\texttt{E}}$, $\overline{\texttt{E}} <: \overline{\texttt{T}}$. Since class C is OK, then method m is OK IN C and, from (M-OK) $\Gamma, \overline{\texttt{x}} : \overline{\texttt{T}}, \texttt{this} : \texttt{D} \vdash \texttt{e} : \texttt{T}_0 <: \texttt{T}$. Therefore, from Lemma 3.3 we have $\Gamma \vdash \texttt{e}\,[\texttt{this} \leftarrow \iota, \overline{\texttt{x}} \leftarrow \overline{\iota}] : \texttt{T}' <: \texttt{T}_0 <: \texttt{T}$.

**Case** (R-DINVK)**.** Similar to the previous one.

**Case** (R-DA-INVK)**.** The last applied rule is

$$\frac{\mathscr{H}(\iota) = (\texttt{D},[\texttt{g}:\iota',\ldots])\qquad \mathit{wrap}(\texttt{D}) = \texttt{I}\qquad \texttt{m} \in \mathit{sign}(\texttt{I})}{\iota.\texttt{m}(\overline{\iota}) \parallel \mathscr{H} \longrightarrow \texttt{Ret}(\texttt{deleg}(\iota'.\texttt{m}(\overline{\iota}),\iota),\iota.\texttt{m}(\overline{\iota}),\iota) \parallel \mathscr{H}}$$

Since $\Gamma \vdash \iota.\texttt{m}(\overline{\iota}) : \texttt{T}$, from rule (T-INVK) (and well-formedness of $\mathscr{H}$), we get that $\Gamma \vdash \iota : \texttt{D}$, $\mathit{mType}(\texttt{m},\texttt{D}) = \overline{\texttt{T}} \to \texttt{T}$, $\Gamma \vdash \overline{\iota} : \overline{\texttt{E}} <: \overline{\texttt{T}}$, for some $\overline{\texttt{E}}$. Since D is OK and $\Gamma \models \mathscr{H}$, then $\Gamma \vdash \iota' : \texttt{E} <: \texttt{I}$ for some E. Since $\texttt{m} \in \mathit{sign}(\texttt{I})$ and $\texttt{D},\texttt{E} <: \texttt{I}$, then, from Lemma 3.2 we have that $\mathit{mType}(m,\texttt{D}) = \mathit{mType}(m,\texttt{E}) = \overline{\texttt{T}} \to \texttt{T}$, and thus $\Gamma \vdash \iota'.\texttt{m}(\overline{\iota}) : \texttt{T}$. Moreover, by using (T-RE-DELEG), we have $\Gamma \vdash \texttt{deleg}(\iota'.\texttt{m}(\overline{\iota}),\iota) : \texttt{T}$. Thus, use (T-RE-RET) to obtain $\Gamma \vdash \texttt{Ret}(\texttt{deleg}(\iota'.\texttt{m}(\overline{\iota}),\iota),\iota.\texttt{m}(\overline{\iota}),\iota) : \texttt{T}$.

**Case** (R-DEL-1)**.** Similar to (R-INVK) by also using Lemma 3.4 that can be applied since $\Gamma \vdash \texttt{deleg}(\iota'.\texttt{m}(\overline{\iota}),\iota) : \texttt{T}$ implies that $\texttt{C} <: \texttt{I}$.

**Case** (R-DEL-2)**.** Similar to (R-DA-INVK). By hypothesis, $\Gamma \vdash \texttt{deleg}(\iota'.\texttt{m}(\overline{\iota}),\iota) : \texttt{T}$; D is OK and $\Gamma \models \mathscr{H}$ imply that $\Gamma(\iota'') <: \mathit{wrap}(\texttt{D})$ and thus (by Lemma 3.2) we use (T-RE-DELEG) to obtain $\Gamma \vdash \texttt{deleg}(\iota''.\texttt{m}(\overline{\iota}),\iota) : \texttt{T}$. $\Gamma \vdash \texttt{deleg}(\iota'.\texttt{m}(\overline{\iota}),\iota) : \texttt{T}$ implies that $\Gamma \vdash \iota'.\texttt{m}(\overline{\iota}) : \texttt{T}$ and $\texttt{D} <: \mathit{wrap}(\iota)$, and thus we can use (T-RE-RET).

**Case** (R-RET)**.** Similar to (R-DEL-1); $\Gamma \vdash \texttt{Ret}(\iota,\iota'.\texttt{m}(\overline{\iota}),\iota'') : \texttt{T}$ implies that $\Gamma \vdash \iota : \texttt{T}$, $\Gamma \vdash \iota'.\texttt{m}(\overline{\iota}) : \texttt{T}$ (thus we can apply Lemma 3.3) and $\texttt{D} <: \mathit{wrap}(\iota'')$ (thus we can apply Lemma 3.4).

□

**Corollary 3.6 (Subject reduction)** *Let* $\Gamma \models \mathscr{H}$, $\Gamma \vdash \mathtt{e} : \mathtt{T}$, $\mathtt{e} \parallel \mathscr{H} \longrightarrow \mathtt{e}' \parallel \mathscr{H}'$, *and* $\Gamma \models \mathscr{H}'$. *Then* $\Gamma \vdash \mathtt{e}' : \mathtt{T}'$, *for some* $\mathtt{T}'$ *such that* $\mathtt{T}' <: \mathtt{T}$.

**Lemma 3.7** *Given the configuration* $\mathtt{e} \parallel \mathscr{H}$ *such that* $\Gamma \vdash \mathtt{e} : \mathtt{T}$ *and* $\Gamma \models \mathscr{H}$, *then*

1. *If* $\mathtt{e} = \iota.\mathtt{f}_i$ *then* $\mathscr{H}(\iota) = (\mathtt{E}, [\ldots, \mathtt{f}_i : \iota_i, \ldots])$ *for some* $\mathtt{E}$;
2. *If* $\mathtt{e} = \iota.\mathtt{m}(\overline{\iota})$ *then* $\mathscr{H}(\iota) = (\mathtt{E}, \ldots)$ *for some* $\mathtt{E}$, $mBody(\mathtt{m}, \mathtt{E}) = (\overline{\mathtt{x}}, \mathtt{e})$ *and* $|\overline{\mathtt{x}}| = |\overline{\iota}|$

**Proof.**

1. Follows directly from well-formedness of heap
2. From $\Gamma \vdash \mathtt{e} : \mathtt{T}$ we have that $\Gamma \vdash \iota : \mathtt{E}$, $mType(\mathtt{m}, \mathtt{E}) = \overline{\mathtt{T}} \to \mathtt{T}$, $\Gamma \vdash \overline{\iota} : \overline{\mathtt{T}}'$ and $\overline{\mathtt{T}}' <: \overline{\mathtt{T}}$;
   - $\mathtt{E} = \mathtt{C}$ for some class `class C extends C' implements` $\overline{\mathtt{I}}$ $\{\overline{\mathtt{T}}\,\overline{\mathtt{f}}$; K; $\overline{\mathtt{M}}\}$. From $mType(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{T}} \to \mathtt{T}$ we have that $\mathtt{T}\ \mathtt{m}\ (\overline{\mathtt{T}}'\ \overline{\mathtt{x}}) \in sign(\mathtt{C})$, and by definition of *sign*, either $\mathtt{m} \in \overline{\mathtt{M}}$ or $\mathtt{T}\ \mathtt{m}\ (\overline{\mathtt{T}}'\ \overline{\mathtt{x}}) \in sign(\mathtt{C}')$ and in both cases *mBody* is defined (Figure 6), $mBody(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e})$; $|\overline{\mathtt{x}}| = |\overline{\iota}|$ follows from $\overline{\mathtt{T}}' <: \overline{\mathtt{T}}$;
   - $\mathtt{E} = \mathtt{D}$ for some wrapper class `class D wraps I` $\{\mathtt{I}\ \mathtt{g}$; $\overline{\mathtt{T}}\,\overline{\mathtt{f}}$; KD; $\overline{\mathtt{M}}\,\overline{\mathtt{N}}\}$. Similar to the previous case by noticing that $mBody(\mathtt{m}, \mathtt{D}) = (\overline{\mathtt{x}}, \mathtt{ret})$ in case $\mathtt{T}\ \mathtt{m}\ (\overline{\mathtt{T}}'\ \overline{\mathtt{x}}) \in sign(\mathtt{I})$ and $\mathtt{m} \notin \overline{\mathtt{N}}$.

□

**Theorem 3.8 (Progress)** *Let* $\Gamma \models \mathscr{H}$ *and* $\Gamma \vdash \mathtt{e} : \mathtt{T}$, *where* e *is a run-time expression. Then either* e *is a value or there exist* $\mathtt{e}'$ *and* $\mathscr{H}'$ *such that* $\mathtt{e} \parallel \mathscr{H} \longrightarrow \mathtt{e}' \parallel \mathscr{H}'$.

**Proof.** The proof is by straightforward induction on the derivation of $\Gamma \vdash \mathtt{e} : \mathtt{T}$ using Lemma 3.7 in crucial cases of field selection and method invocation. □

**Theorem 3.9 (Soundness)** *If* $\emptyset \vdash \mathtt{e} : \mathtt{T}$ *and* $\mathtt{e} \parallel \emptyset \longrightarrow^{\star} \mathtt{e}' \parallel \mathscr{H}$, *with* $\mathtt{e}' \parallel \mathscr{H}$ *normal form, then there exist* $\Gamma$ *and* $\mathtt{T}' <: \mathtt{T}$ *such that* $\Gamma \models \mathscr{H}$, $\Gamma \vdash \mathtt{e}' : \mathtt{T}'$, *and* $\mathtt{e}'$ *is a value.*

**Proof.** By Corollary 3.6 and Theorem 3.8. □

## 4 CONCLUSIONS AND RELATED WORKS

In this paper we presented an extension for Java like languages that permits specializing method behaviors with the dynamic mechanisms of wrappers, which act at run-time, at object level (thus fostering flexibility), and are type safe. We believe that wrappers help in keeping the design and the implementation of specific software systems simpler (less programming required) and safer (more conditions checked statically), in particular in those situations where class inheritance shows its limits due to its static nature and leads

to code that is harder to maintain. Typically, this is the case when the class construct is responsible of too many concerns, while, instead, behavior definitions and behavior structuring should be kept separated by relying on the constructs provided by the language [26]. Following this philosophy we proposed our language extension.

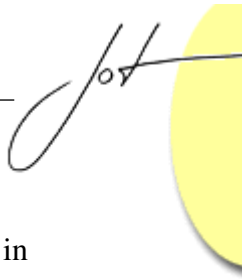
Wrappers are useful in all those cases where the software architecture is structured in layers where each layer is independent and interchangeable with each other. We have already cited the case of stream libraries. The other recurrent case is the one of a communication protocol stack (e.g., the TCP/IP stack) where each layer of the stack takes care of dealing with specific type of information contained in the headers of a packet. Each layer, upon writing, will add specific header information, and upon reading, will remove specific header information, and will pass the rest of the packet to the other layers in the stack. Finally, wrappers allow the programmer to implement small software components each one dealing with a specific single goal and to glue them together dynamically (this recalls the UNIX philosophy).

There can be some drawbacks in using wrappers, though: first of all, a class that acts as a wrapper must be designed and written as a wrapper from the beginning; turning an existing class into a wrapper might not be straightforward, in case that class has already many methods, since those methods should be manually analyzed to see where to use the keyword `ret`. Furthermore, once a class is designed as a wrapper it cannot be used to create standalone instances: each wrapper, upon creation, requires an object to wrap. We think that these drawbacks are only due to the specialized features the wrappers provide and are based upon: wrappers are higher level constructs and should be used in the right context to solve specific problems (just like design patterns).

Furthermore, wrappers are thought to add behavior to existing objects, and cannot be removed once objects are wrapped; this is basically a legacy from the decorator pattern. However, concerning this point, we might think of extending FWJ with linguistic constructs that also permit removing wrappers. This issue is still under investigation.

Finally another limitation (again inherited by decorator pattern) is that our approach can be easily applied to existing class hierarchies, by adding new wrapper classes, only when there is a common interface. We are studying an inference mechanism which detects the set of common methods w.r.t. set of classes in order to exploit our approach also where there is not a common interface.

The implementation of the proposed language extension is still under study. We think of implementing wrappers only with a preprocessor, i.e., without modifying the virtual machine: given a program written in the extended Java with wrappers, the preprocessor will generate standard Java code with the same semantics (w.t.r. a formalization of this translation will be studied and proved correct). Probably, the declarative parts might rely on the Java annotation features. As for methods, we will need to change the method signatures and arguments during translation, since we need to correctly bind `this` for delegation, and the `ret` implicit variable. Furthermore, we will need to take into consideration also possible interactions with other features of the language we did not consider here, first of all, generic types.



In the rest of the section we will review main related works in the literature.

Aspects [22] permit isolating functionalities in typed entities that can be injected in other types. These functionalities can be introduced in specific *entry points* such as instances creation and method calls. As wrapper objects, aspects can enclose in a module features that can be reused through a program in different points. On the other hand aspects are conceptually separated from wrapper objects: aspects are used to implement crosscutting concerns (pieces of code which affect other parts of the program such as exceptions) while wrappers represent classes of *decorators* for other objects.

Concerning method specialization mechanism our approach offers a similar computational "feel" as the *Beta* inheritance [25] that is designed to avoid the replacement of a method by a completely different method in subclasses (via standard overriding). In Beta a method in a subclass can add some behaviors but cannot replace the version defined in the superclass: the version defined in the subclass can be called using `inner` keyword. The difference between the two approaches is that wrappers are applied to objects and can be dynamically composed in several ways, while Beta inheritance is a static mechanism concerning class definition. The same difference holds between Wrap Java and the approach in [20] (which integrates Beta inheritance style in Java) and GBETA [14]. In the language GBETA dynamic combination of classes coexists with static type checking (the type of dynamically created classes is partially known statically, in that it is known that the dynamic class is a subtype of a given, statically known class).

[31, 23] extend Java like languages with consultation and delegation respectively, thus extending object behaviors by forwarding messages to objects with a different type/interface. To this aim classes defining *delegator* objects usually have an explicit reference to the *delegate* object. Instead, the use of wrapper objects is transparent to component objects and classes. Thus, our approach is different from the one adopted in [31, 23] since component objects are not aware about wrapper objects: in this way the programmer can manage multiple nested or disjoint wrappers. The transparency of wrapper object makes code more reusable since every instance of an object implementing interface `I` can be wrapped by objects of type `I`, while in [31, 23] the delegating class /object has to explicitly manage the delegate attribute. On the other hand, in our approach, the use of delegation is restricted to the methods belonging to the "wrapped" interface `I`.

*Generic wrappers* [10] are a language construct for Java to aggregate objects at runtime. Generic wrappers implement inheritance at object level: a wrapper object is a subtype of the wrapped object (called *wrappee*) and wrapper methods override wrappee's ones. In order to properly model overriding some consistency check between the wrapper and the wrappee are made at run-time, possibly throwing exceptions at the moment of wrapper instantiation. Finally, generic wrappers implement only consultation (the authors claim that also delegation can be easily modeled) and do not allow disjunctive wrapping (two wrappers cannot be attached to the same wrappee object).

[27] proposes a model where all the features that proved so useful for single classes/objects (inheritance, delegation, late binding and subtype polymorphism) automatically apply to sets of collaborating classes and objects. The main difference between

this approach and ours is that we restrict wrapping and method style forwarding to objects implementing a common interface while in [27] there is more flexibility concerning the hierarchy of involved instances and more expressive power since delegation layers permit 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.

Both wrapper objects and the *mixin based* approach [8, 16, 6] encapsulate extensions in classes making them reusable. As for standard inheritance, the main difference with wrapper objects is that when we compose one or more mixins with a class, a new class, and therefore a new type, is created, while, in wrapper objects, the composition is moved at instance level so neither new classes nor new types are created. We also observe that the dynamic flexibility achieved by wrapper objects relies in that it enables a run-time decoration of objects, while mixins permit to statically "decorate" code (classes). A mixin-based approach that is more similar to our wrappers is the one of *incomplete objects* [3], i.e., instances of mixins that can be completed at run-time by composition with complete objects (instances of classes) or with single methods. However, incomplete objects do not have the mechanisms of automatic delegation/consultation/specialization of our wrappers.

*Fickle* [13] implements a dynamic object *reclassification* (i.e., objects can change class membership at run-time) in order to represent object evolutions and changes. For example, an instance of class `Student` can change class at run-time becoming an instance of class `Employee`. Our approach is different in that we let incrementally add features to a kernel of essential features that do not change (these are the ones belonging to the Component classes) instead, in Fickle an object loses the features that were typical of the previous role, when it is reclassified. Our proposal is oriented to model a dynamic notion of *non-exclusive roles* [18] rather than a dynamic change of mutually exclusive types.

[28] proposes *compound references*, a new abstraction for object references, which provides explicit linguistic support for combining different composition properties on-demand. The model is statically typed and allows the programmer to express a various kind of composition semantics in the interval between object composition and inheritance. We share with the authors the view that new fine grained linguistic features, that can be combined, can increase the flexibility in object oriented languages thus avoiding the need of many design patterns originally proposed to overcome lack of adequate constructs in a language.

**Acknowledgments** We would like to warmly thank Betti Venneri who encouraged us to write this paper and gave us crucial suggestions on this language extension. We thank the anonymous referees for providing helpful hints for improving the paper.

## REFERENCES

[1] C. Anderson, F. Barbanera, and M. Dezani-Ciancaglini. Alias and Union Types for Delegation. *Annals of Mathematics, Computing & Teleinformatics*, 1(1):1–18, 2003.

[2] C. Anderson and S. Drossopoulou. $\delta$ - an imperative object based calculus. Workshop USE, Malaga, 2002.

[3] L. Bettini, V. Bono, and S. Likavec. Safe and Flexible Objects. In *Proc. of SAC 2005, Special Track on Object-Oriented Programming Languages and Systems (OOPS)*, pages 1268–1273. ACM Press, 2005.

[4] L. Bettini, S. Capecchi, and E. Giachino. Featherweight Wrap Java. In *Proc. of SAC 2007, Special Track on Object-Oriented Programming Languages and Systems (OOPS)*, pages 1094–1100. ACM Press, 2007.

[5] L. Bettini, S. Capecchi, and B. Venneri. Extending Java to dynamic object behaviors. In *Proc. of Int. Workshop on Object-Oriented Developments (WOOD)*, volume 82 of *ENTCS*. Elsevier, 2003.

[6] V. Bono, A. Patel, and V. Shmatikov. A Core Calculus of Classes and Mixins. In R. Guerraoui, editor, *Proceedings ECOOP'99*, number 1628 in LCNS, pages 43–66. Springer-Verlag, 1999.

[7] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, University of Utah, 1992.

[8] G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. OOPSLA '90*, pages 303–311, 1990.

[9] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, pages 183–200, Oct. 1998.

[10] M. Büchi and W. Weck. Generic wrappers. In E. Bertino, editor, *ECOOP 2000, 14th European Conference on Object-Oriented Programming*, volume 1850 of *LNCS*, pages 201–225. Springer-Verlag, 2000.

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

[12] F. Damiani, E. Giachino, P. Giannini, N. Cameron, and S. Drossopoulou. A State Abstraction for Coordination in Java-like Languages. In *Electronic proceedings of FTfJP'06 (http://www.cs.ru.nl/ftfjp)*, 2006.

[13] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More dynamic object reclassification: *Fickle*$_{\|}$. *ACM Transactions on Programming Languages and Systems*, 24(2):153–191, 2002.

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

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

[16] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proc. POPL '98*, pages 171–183, 1998.

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

[18] G. Ghelli and D. Palmerini. Foundations for extensible objects with roles, extended abstract. In *FOOL*, 1999.

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

[20] D. S. Goldberg, R. B. Findler, and M. Flatt. Super and inner: together at last! In *Proc. of OOPSLA '04*, pages 116–129. ACM Press, 2004.

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

[22] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *ECOOP '97*, volume 1241 of *LNCS*, pages 220–242. Springer, 1997.

[23] G. Kniesel. *Darwin - A Unified Model of Sharing for Object-Oriented Programmning*. PhD thesis, University of Bonn, 1999.

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

[25] O. L. Madsen, B. Moller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Language*. Addison-Wesley, 1993.

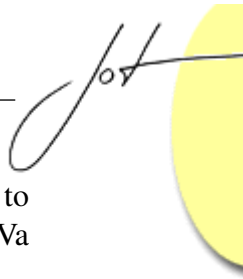
[26] M. Mezini. *Variational Object-Oriented Programming Beyond Classes and Inheritance*. Springer, 1998.

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

[28] K. Ostermann and M. Mezini. Object-oriented composition untangled. In *OOPSLA*, pages 283–299, 2001.

[29] B. C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, MA, 2002.

[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.

## ABOUT THE AUTHORS

**Lorenzo Bettini** is a Researcher in Computer Science at Dipartimento di Informatica, University of Torino, Italy. His research focuses on theory, extension and implementation of mobile code and object-oriented languages, and on distributed applications.
He can be reached at http://www.lorenzobettini.it.

**Sara Capecchi** is a research fellow in Computer Science at Dipartimento di Matematica e Informatica, Università di Catania, Italy. Her research interests are types, calculi and extensions for object-oriented languages. She can be reached at capecchi@dmi.unict.it.

**Elena Giachino** is a PhD student in Computer Science at Dipartimento di Informatica, Università di Torino, Italy, and at Laboratoire PPS, Université Paris VII, France. Her research interests are types and calculi for object-oriented languages and concurrent processes. She can be reached at giachino@di.unito.it.