Science of Computer Programming 76 (2011) 992–1014

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Delegation by object composition☆

Lorenzo Bettini[a,*], Viviana Bono[a], Betti Venneri[b]

[a] Dipartimento di Informatica, Università di Torino, Italy

[b] Dipartimento di Sistemi e Informatica, Università di Firenze, Italy

ARTICLE INFO

Article history:
Received 20 July 2009
Received in revised form 19 October 2009
Accepted 2 December 2009
Available online 29 April 2010

Keywords:
Language extensions
Featherweight Java
Object composition
Delegation
Consultation

ABSTRACT

Class inheritance and method overriding, as provided by standard class-based languages, are often not flexible enough to represent objects with some dynamic behavior. In this respect, object composition equipped with different forms of method body lookup is often advocated as a more flexible alternative to class inheritance since it takes place at run time, thus permitting the behavior of objects to be specialized dynamically. In this paper, we illustrate Incomplete Featherweight Java (IFJ), an extension of Featherweight Java with a novel linguistic construct, the incomplete object. Incomplete objects require some missing methods which can be provided at run time by composition with another (complete) object. Furthermore, we present two mechanisms for the method body lookup on (composed) objects, one based on delegation and the other based on consultation. Thanks to the design of the language, the consultation-based lookup is a simple extension of the delegation-based one. Both mechanisms are disciplined by static typing, therefore the language enjoys type safety (which implies no "message-not-understood" run-time errors) and avoids possible accidental overrides due to method name clashes.

1. Introduction

Design patterns were introduced as "programming recipes" to overcome some of the limitations of class-based object-oriented languages. Indeed, standard mechanisms provided by class-based object-oriented languages, such as inheritance and dynamic binding, usually do not suffice for representing dynamic behavior of objects (we refer to [35,18] and to the references therein for an insightful review of the limitations of inheritance). Most of the design patterns in [22] rely on *object composition* as an alternative to class inheritance, since it is defined at run time and it enables dynamic object code reuse by assembling existing components. Patterns exploit the programming style consisting in writing small software components (units of reuse), that can be composed at run time in several ways to achieve software reuse. However, design patterns require manual programming, which is prone to errors.

Differently from class-based languages, object-based languages do use object composition, and *delegation* as the mechanism for method call, to reuse code (see, e.g., the languages [36,23,16], and the calculi [21,2]). Every object has a list of *parent* objects: when an object cannot answer a message it forwards it to its parents until there is an object 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 [37]. We refer to Kniesel [26] for an overview of problems when combining delegation with a static type discipline.

Our goal is the design of a class-based language that offers a form of object composition with the following goals in mind:

- maintaining the class-based type discipline;

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

* Corresponding author.
E-mail addresses: bettini@di.unito.it (L. Bettini), bono@di.unito.it (V. Bono), venneri@dsi.unifi.it (B. Venneri).

0167-6423/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.04.006

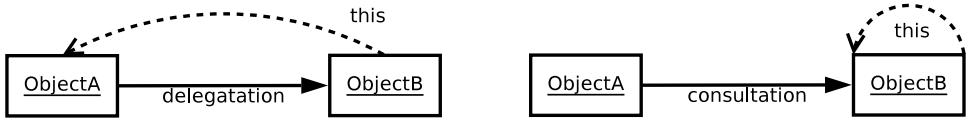

Fig. 1. Binding of `this` in delegation and consultation.

- gaining some of the flexibility of the object-based paradigm;
- preventing the message-not-understood error statically;
- preventing possible name clashes for methods with the same name but different signatures
- preserving the nominal type system of Java-like languages.

In [10,6] we introduced a novel linguistic construct, the *incomplete object*. The programmer, besides standard classes, can define *incomplete* classes whose instances are *incomplete* objects that can be composed in an object-based fashion. Hence, in our calculi it is possible: (i) to instantiate standard classes, obtaining fully fledged objects ready to be used; (ii) to instantiate incomplete classes, obtaining *incomplete objects* that can be composed (by *object composition*) with complete objects, thus yielding new complete objects at run time; (iii) in turn, to use a complete object obtained by composition to be composed with other incomplete objects. Therefore, objects are not only instances of classes (possibly incomplete), but they are also prototypes that can be used, via the object composition, to create new objects at run time, while ensuring statically that the composition is type safe. Then, we can use incomplete and complete objects as our re-usable building blocks to assemble at run time, on the fly, brand new objects.

Our two proposals are for Java-like languages and differ from each other with respect to the method body lookup: the first one [10] combines incomplete objects with *consultation*, and the second one [6] with *delegation* [29].[1] 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 *n*, then also this call will be executed binding `this` to *A*; while with consultation, during the execution of the body, the implicit parameter is always bound to the receiver *B*. This is depicted in Fig. 1. Delegation is more powerful as it endows *dynamic method redefinition*. Both our proposals result type safe, therefore they capture statically message-not-understood errors. In particular, our proposals are two versions of *Incomplete Featherweight Java* (IFJ), as extensions of Featherweight Java [25,31].

One of our key design choices is to integrate object composition within the nominal subtyping mechanism that is typical for mainstream languages like Java (and C++). This feature makes the extension conservative with respect to the core Java, since it does not affect those parts of the programs that do not use incomplete objects. Furthermore, incomplete classes can rely on standard class inheritance to reuse code of parent classes (although this kind of inheritance does not imply subtyping in our setting). Thus, incomplete objects provide two forms of code reuse: *vertical* (i.e., the code reuse achieved via standard class inheritance) and *horizontal* (i.e., the one achieved via object composition). Finally, in order to enhance run-time flexibility in composing objects, we implicitly use structural subtyping during composition: an incomplete object can be composed with any (complete) object providing all the requested methods (that is, the signatures must match), no matter what its class is. Therefore, the language extension we propose is not a manual implementation of the object composition. In the case of a manual implementation, the object should be stored in a class field, thus forcing it to belong to a specific class hierarchy. Alternatively, one could use type `Object`, and then call methods using Java Reflection APIs or down-casts; however, this solution is not type safe, since exceptions can be thrown at run time due to missing methods (we will show further details concerning possible manual implementations in Section 3).

Incomplete classes, besides standard method definitions, can declare some methods as "incomplete", either *abstract* or *redefining*. Abstract methods are similar to abstract methods in class-based inheritance: the body of these methods must be provided during object composition. Redefining methods are similar to method overriding in class-based inheritance: they expect to redefine a method of another object. Then, we call these methods "redefining" because they will be the active part in the redefinition when an incomplete object will be composed with a complete object. We call the corresponding overridden methods of the complete object *redefined*. Moreover, just like in class-based inheritance, there is a way in an overridden method to access the previous implementation (e.g., `super` in Java): in a redefining method we can access the redefined version with the special variable `next`. We can think of `next` as an "horizontal" version of `super`, that is, a reference to the method body that is present in the complete object of an object composition: the method version in the incomplete object redefines the one present in the complete one. This is a form of incompleteness other than the one based on the abstraction of a method, because it assumes for an incomplete object that in a "future" composition a complete object will provide the method to be redefined.

All these incomplete methods, abstract and redefined, must be provided during object composition by complete objects. Thus, object composition is the run-time version of class inheritance and delegation in composed objects correspond to dynamic binding for method invocation in standard derived classes. We see this as a sort of dynamic inheritance since

[1] We note that in the literature (e.g., [22]), the term *delegation*, originally introduced by Lieberman [29], is given different interpretations and it is often confused with the term *consultation*.

L	::=	`class C extends C {`$\overline{\texttt{C}}$ $\overline{\texttt{f}}$`; K;` $\overline{\texttt{M}}$`}`	classes
A	::=	`class C abstracts C {`$\overline{\texttt{C}}$ $\overline{\texttt{f}}$`; K;` $\overline{\texttt{M}}$ $\overline{\texttt{N}}$ $\overline{\texttt{R}}$`}`	incomplete classes
K	::=	`C(`$\overline{\texttt{C}}$ $\overline{\texttt{f}}$`){super(`$\overline{\texttt{f}}$`); this.`$\overline{\texttt{f}}$`=`$\overline{\texttt{f}}$`;}`	constructors
M	::=	`C m (`$\overline{\texttt{C}}$ $\overline{\texttt{x}}$`){return e; }`	methods
N	::=	`C m (`$\overline{\texttt{C}}$ $\overline{\texttt{x}}$`);`	abstract methods
R	::=	`redef C m (`$\overline{\texttt{C}}$ $\overline{\texttt{x}}$`){return e; }`	redefining methods
e	::=	`x` $\mid$ `e.f` $\mid$ `e.m(`$\overline{\texttt{e}}$`)` $\mid$ `new C(`$\overline{\texttt{e}}$`)` $\mid$ `e` $\leftarrow\!\!+$ `e`	expressions
v	::=	$\langle$`l, l`$\rangle$	values
l	::=	`new C(`$\overline{\texttt{v}}$`) ::` ϵ $\mid$ `new C(`$\overline{\texttt{v}}$`) :: l`	run-time object list

Fig. 2. IFJ syntax; run-time syntax appears shaded.

it implies both substitutivity (that is, a composed object can be used where a standard object is expected) and dynamic code reuse (since composition permits supplying at run time the missing methods with those of other objects). Namely, substitutivity for composed objects is achieved by extending Java subtyping, while dynamic code reuse corresponds to the extension of standard subclassing. Therefore, we can model some features related to dynamic object evolution: while incomplete classes separate the object invariant behavior from the variant one at compile time, at run-time object composition customizes the unpredictable behavior based on dynamic conditions (for instance, the object state) in a type-safe way. In particular, some behavior that was not foreseen when the class hierarchy was implemented may be supplied dynamically by making use of already existing objects, thus generating an unanticipated reuse of code and a sharing of components.

In this paper, firstly we extend the delegation-based proposal [6] by revising and improving the formal part and by including the full proof of type soundness. Providing delegation, instead of consultation, enhances the flexibility of object composition in IFJ and makes dynamic method redefinition effective (in fact, in [10] we were not able to provide method redefinition). Moreover, it requires an interesting technical treatment to achieve a type-safe implementation. For instance, we need to avoid possible name clashes for methods with the same name but with different signatures (possibly due to the subtyping, [21]) and possible accidental method overrides (when a method in the incomplete object, which is not redefining, has the same name and signature of a method of the complete object). In order to deal with such problems, we employ a static annotation procedure (based on static types) which is used in the operational semantics to bind the self-object `this` correctly in the method bodies.

Then, we show how consultation can be added to the calculus. Consultation was not only useful as a preliminary study for appreciating how the composition mechanism can be integrated in a Java-like, class-based setting (the main goal of [10]), but it is interesting in its own, since it provides the programmer with more control on method invocation (see Section 7).

The main contributions of this paper, with respect to [6], are *inter alia*: (i) putting together the consultation-based and the delegation-based proposals, by using an unique model of object composition; (ii) a full formalization of the delegation-based proposal, including a complete proof of the type safety property. The type safety property for the version with both delegation and consultation follows in a straightforward way, as discussed at the end of Section 7.

The paper is organized as follows. Section 2 defines the calculus for object composition with delegation. Section 3 illustrates the application of our proposal to recurrent programming scenarios. Sections 4 and 5 develop the typing system and the operational semantics, respectively. Section 6 presents the full proof of the type safety. Section 7 shows how consultation can be implemented in the calculus, in addition to delegation. Section 8 discusses some related works and Section 9 concludes.

2. Incomplete Featherweight Java

In this section we present the core language IFJ (*Incomplete Featherweight Java*), which is an extension of FJ (*Featherweight Java*) [25,31] with incomplete objects, dynamic object composition and delegation. FJ 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.[2] 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", [31], page 248). In particular, the Java features that are omitted in FJ, e.g., visibility of methods, method overloading, abstract classes/interfaces, are orthogonal to our extension, too, while retaining the core features of Java typing. We observe that the interaction with Java generics can be an interesting ongoing research subject when generic class instances take part to object composition.

Although we assume the reader is familiar with FJ, we will briefly comment on the FJ part and then we will focus on the novel aspects introduced by IFJ.

The abstract syntax of the IFJ constructs is given in Fig. 2 and it is just the same as FJ extended with incomplete classes, abstract methods, redefining methods and object composition (and some run-time expressions that are not written by the

[2] FJ also includes up- and down-casts; however, since these features are completely orthogonal to our extension with incomplete objects, they are omitted in IFJ.

programmer, but are produced by the semantics, that we will discuss later in Section 5). As in FJ, we will use the overline notation for possibly empty sequences (e.g., "$\overline{\texttt{e}}$" is a shorthand for a possibly empty sequence "$\texttt{e}_1, \ldots, \texttt{e}_n$"). The empty sequence is denoted by $\bullet$.

Following FJ, we assume that the set of variables includes the special variable `this` (implicitly bound in any method declaration), which cannot be used as the name of a method's formal parameter (this restriction is imposed by the typing rules). In IFJ we also introduce the special variable, `next`: in a redefining method body, with `next`, one can access the "redefined" object. For instance, in a redefining method `m` we can access the redefined version with `next.m()`.[3] Thus, `next` is the dynamic (and horizontal) version of `super` (intended as in the full Java language, not only as the call to the superclass constructor).[4] Just like `this`, `next` is a variable that will be implicitly bound in `redef` methods. Note that since we treat `this` and `next` in method bodies as ordinary variables, no special syntax for them is required.

A class declaration `class C extends D` $\{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K};\ \overline{\texttt{M}}\}$ consists of its name `C`, its superclass `D` (which must always be specified, even if it is `Object`), a list of field names $\overline{\texttt{C}}\ \overline{\texttt{f}}$ with their types, the constructor `K`, and a list of method definitions $\overline{\texttt{M}}$. The fields of `C` are added to the ones declared by `D` and its superclasses and are assumed to have distinct names. The constructor declaration shows how to initialize all these fields with the received values. A method definition `M` specifies the name, the signature and the body of a method; a body is a single `return` statement since FJ is a functional core of Java.

An incomplete class declaration `class C abstracts D` $\{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K};\ \overline{\texttt{M}}\ \overline{\texttt{N}}\ \overline{\texttt{R}}\}$ inherits from a standard (or incomplete) class and, apart from adding new fields and adding/overriding methods, it can declare some methods as "incomplete". There are two kinds of incomplete methods:

- "abstract" methods: the incomplete class declares only the signature of these "expected" methods; the body of these methods must be provided during object composition;
- "redefining" methods: although the body of these methods is provided by the incomplete class, they are still incomplete since the special variable `next` will be bound during object composition. We call these methods "redefining" because they will be the active part in the redefinition when an incomplete object (of an incomplete class) will be composed with a complete object. We then call the corresponding overridden methods of the complete object "redefined".

Standard classes cannot inherit from incomplete classes (this is checked by the type system, see Section 4). The main idea of our language is that an incomplete class can be instantiated, leading to *incomplete objects*. Method invocation and field selection cannot be performed on incomplete objects.[5]

In the following, we will write $\texttt{m} \notin \overline{\texttt{M}}$ to mean that the method definition of the name `m` is not included in $\overline{\texttt{M}}$. The same convention will be used for abstract method signatures $\overline{\texttt{N}}$ and for redefining methods $\overline{\texttt{R}}$.

An incomplete object expression $\texttt{e}_1$ can be composed at run time with a complete object expression $\texttt{e}_2$; this operation, denoted by $\texttt{e}_1 \leftarrow\!\!+\ \texttt{e}_2$, is called *object composition*. The key idea is that $\texttt{e}_1$ can be composed with a complete object $\texttt{e}_2$ that provides all the requested methods, independently from the class of $\texttt{e}_2$ (of course, the method signatures must match). Then, in $\texttt{e}_1 \leftarrow\!\!+\ \texttt{e}_2$, $\texttt{e}_1$ must be an incomplete object and $\texttt{e}_2$ must be a complete object expression (these requirements are checked by the type system); indeed, $\texttt{e}_2$ can be, in turn, the result of another object composition. The object expression $\texttt{e}_1 \leftarrow\!\!+\ \texttt{e}_2$ represents a brand new (complete) object that consists of the sub-object expressions $\texttt{e}_1$ and $\texttt{e}_2$; in particular, the objects of these subexpressions are not modified during the composition. This also highlights the roles of incomplete and complete objects as re-usable building blocks for new objects at run time, while retaining their identity and state.

We do not allow object composition operations leading to incomplete objects, i.e., incomplete objects can only be fully completed. However, for instance, object compositions of the shape $(\texttt{e}_1 \leftarrow\!\!+\ \texttt{e}_2) \leftarrow\!\!+\ \texttt{e}_3$, where $\texttt{e}_2$ is incomplete in the methods provided by $\texttt{e}_3$, can be obtained as $\texttt{e}_1 \leftarrow\!\!+\ (\texttt{e}_2 \leftarrow\!\!+\ \texttt{e}_3)$ in IFJ. Furthermore, we prohibit the object composition between two complete objects; the semantics and the type system can be extended in order to deal with such an operation in a type-safe way, but we prefer to keep the core calculus and its formal theory simple in this presentation.

Finally, values, denoted by `v` and `u`, are fully evaluated object creation terms. The object representation of IFJ is different from FJ in that fully evaluated objects can be also compositions of many objects. Thus, objects are represented as lists of terms `new C`$(\overline{\texttt{v}})$ (i.e., expressions that are passed to the constructor are values, too). For instance, `new C`$(\overline{\texttt{v}})$ :: `new D`$(\overline{\texttt{u}})$ :: ϵ represents the composition of the incomplete object of class `C` with a standard complete object of class `D` (ϵ denotes the empty list). During method invocation, this list is scanned starting from the leftmost object in search for the called method (of course, in a well-typed program this search will terminate successfully). However, in order to implement delegation in a type-safe way, we need to keep the position in the list where we found the called method. Technically, this is implemented by using a pair of object lists: the first one will be the part of the object list scanned during method invocation, and the second one will be the entire composed object, i.e., the complete list. The basic idea is to use the complete list when binding `this` for redefined methods, and use the current position of the scanned list when binding `this` for methods that are not redefined (this solves the problem of accidental name clashes) and for binding `next` in redefining methods. This run-time representation of objects will be further explained when presenting the operational semantics of the calculus in Section 5.

---

[3] An alternative choice could have been to allow a redefining method to access only the redefined version of the method, and not the whole "next" object; however, our choice is in line with the mainstream programming languages, although it is considered poor style [33].

[4] If `super` would be added with its full meaning to FJ, it could coexist with `next`.

[5] Actually, field selection might be safely performed on incomplete objects but would make little sense.

$$\mathtt{T} <: \mathtt{T} \qquad \frac{\mathtt{T}_1 <: \mathtt{T}_2 \quad \mathtt{T}_2 <: \mathtt{T}_3}{\mathtt{T}_1 <: \mathtt{T}_3} \qquad \frac{\texttt{class C extends D } \{\ldots\}}{\mathtt{C} <: \mathtt{D}}$$

$$\frac{\texttt{class C abstracts D } \{\ldots\} \quad \texttt{class D extends E } \{\ldots\}}{\langle \mathtt{C} \rangle <: \mathtt{D}}$$

$$\frac{\texttt{class C abstracts D } \{\ldots\} \quad \texttt{class D abstracts E } \{\ldots\}}{\langle \mathtt{C} \rangle <: \langle \mathtt{D} \rangle}$$

Fig. 3. Subtyping rules.

As in FJ, a class table *CT* is a mapping from class names to class declarations. Then a program is a pair (*CT*, 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) $CT(\mathtt{C}) = \texttt{class C } \ldots$ for every $\mathtt{C} \in dom(CT)$; (ii) for every class name C (except `Object`) appearing anywhere in *CT*, we have $\mathtt{C} \in dom(CT)$; (iii) there are no cycles in the transitive closure of the `extends` relation. Thus, in the following, instead of writing $CT(\mathtt{C}) = \texttt{class } \ldots$ we will simply write $\texttt{class C } \ldots$.

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

The subtype relation $<:$ (defined for any class table *CT*) on classes (types) is induced by the standard subclass relation extended in order to relate incomplete objects (Fig. 3). First of all, we consider an incomplete class $\texttt{class C abstracts D } \{\ldots\}$; if D is a standard class, since C can make some methods of D incomplete, then it is obvious that an incomplete object of class C cannot be used in place of an object of class D. Thus, `abstracts` implements subclassing without subtyping. In spite of this, since inheritance and subtyping need not to be connected, we will still refer to D as the superclass of C. Instead, when the incomplete object is composed with a complete object (providing all the methods requested by C), then its type is $\langle \mathtt{C} \rangle$, and it can be used in place of an object of class D (see the fourth rule). Since, as said above, we do not permit object composition on a complete object, then a complete object can never be used in place of an incomplete one. Instead, subtyping holds on their completed versions (last rule). We could introduce subtyping between incomplete objects: this would require checking that the subtype does not have more incomplete methods than the supertype (contra-variance on requirements); this is the subject of future work.

3. Programming examples

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

3.1. Graphical widgets

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

The design pattern *command* [22] is useful for implementing these scenarios, since it permits the parametrization of widgets over the event handlers, and the same event handler can be re-used for similar widgets (e.g., the handler for the event "save file" can be associated with a button, a menu item, or a keyboard shortcut). Thus, they delegate to this object the actual implementation of the action semantics, while the action widget itself abstracts from it. This decouples the action visual representation from the action controller implementation.

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

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

```
class Action {
  void run() { }
  void display() { }
  void enable(boolean b) { }
}

class Button abstracts Action {
  void run(); // incomplete method
  void display() {
    // redefined to draw the button
  }
  redef void enable(boolean b) {
    next.enable(b);
    // enable/disable button
  }
}

class SaveActionDelegate {
    void run() {
        // implementation
        enable(false);
    }
    void enable(boolean b) {
        // implementation
    }
}

class MenuItem abstracts Action {
  void run(); // incomplete method
  void display() {
    // redefined to show the item
  }
  redef void enable(boolean b) {
    next.enable(b);
    // enable/disable item
  }
}

class KeyboardAccel abstracts Action {
  void run(); // incomplete method
  void display() {
    // redefined to hook key combination
  }
}

class Frame {
    void addToMenu(Action a) {...}
    void addToToolbar(Action a) {...}
    void setKeybAcc(Action a) {...}
}

SaveActionDelegate deleg =
  new SaveActionDelegate();
myFrame.addToMenu
  (new MenuItem("save") <- deleg);
myFrame.addToToolbar
  (new Button("save") <- deleg);
myFrame.setKeybAcc
  (new KeyboardAccel("Ctrl+S") <- deleg);
```

Listing 1: The implementation of action and action delegates with incomplete objects and object composition.

```
class Button extends Action {
  Runnable deleg;
  void run() { deleg.run(); }
  void display() {
    // redefined to draw the button
  }
}

class Button extends Action {
  Object deleg;
  void run() {
    // reflection or casts!
  }
  void enable(boolean b) { ... }
}
```

Listing 2: Possible manual implementations in Java.

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

We now concentrate on the dynamic redefinition of `enable`. This method is used to enable/disable the graphical widget (e.g., buttons and menu items can be shaded when disabled) and also actions (when a document is saved, the action can be disabled until the document is modified). When `run` is executed in `SaveActionDelegate`, the method also invokes `enable`. In a composed object, since we implement delegation, it is guaranteed that the redefining version of the method will be called (note, however, that the redefining versions will also call `enable` on `next`). We refer to Section 5.2.1 for an example of reduction involving this example.

We now investigate some possible manual implementations in Java of this scenario (Listing 2), showing that our proposal is not simply syntactic sugar. With standard Java features, one could write the `Button` class with a field, say `deleg`, on which we call the method `run`. This approach requires `deleg` to be declared with a class or interface that provides such a method, say `Runnable`. However, this solution would not be as flexible as our incomplete objects, since one can then assign to `deleg` only objects belonging to the `Runnable` hierarchy. This might not be optimal in case of reuse of existing code (that cannot be modified); in particular, this scenario is not ideal for unanticipated code reuse. The solution can be refined to deal with these problems by introducing some *Adapters* [22], but this requires additional programming. On the other hand, if one wanted to keep the flexibility, one should declare `deleg` of type `Object`, and then call the method `run` by using Java Reflection APIs, (e.g., `getMethod`), or down-casts; however, this solution is not type safe, since exceptions can be thrown at run time due to missing methods. Finally, implementing delegation manually would even be harder: it would require to modify all the methods in order to pass explicitly "another" `this`, i.e., the one bound to the original sender (we refer to Section 1 and Fig. 1).

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

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

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

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

Listing 3: The implementation of streams using redefined methods.

3.2. Streams

There are situations when one needs to add functionalities to an object dynamically. The design pattern *decorator* [22] is typically used to deal with these scenarios: at run time an object (called *component*) is embedded in another object (*decorator*) that associates to the component additional features (by relying also on the implementation of the component). Since a decorator is a component itself, this can be used to create a chain of decorators.

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

Although this pattern is useful in practice, it still requires manual programming. With method redefinition we could easily implement a stream library, as sketched in Listing 3: the specific stream specializations rely on the methods provided during object composition (using `next`) and redefine them. In order to show how delegation is implemented in our language, we introduced also the method `readBuffer` both in `CompressStream` and in `BufferedStream`. These two methods, in spite of having the same name, are completely unrelated (we also used different signatures). The operational semantics (Section 5) guarantees that the right implementation will be invoked, depending on the context in which this method is invoked; for instance, the method `read` in `BufferedStream` invokes `readBuffer`, and at run time the version defined in `BufferedStream` will be selected (thus run-time type errors are avoided). The same holds when `readBuffer` is invoked in the method `uncompress` in `CompressStream`: the version of `readBuffer` in `CompressStream` will be selected at run time (we refer to Section 5.2.2 for an example of reduction involving this example).

This example also shows the two different designs in modeling our stream framework: `FileStream` is not modeled as an incomplete class since it can be implemented with all the functionalities; on the contrary, `CompressStream` and `BufferedStream` rely on another stream and thus they are incomplete classes (and their methods are redefining). It is then clear that `CompressStream` and `BufferedStream` can be re-used independently from the actual stream implementation; on the contrary, a `FileStream` can be "decorated" with further functionalities, but it could also be used as it is.

Here it is a possible object composition using the classes in Listing 3:

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

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

4. Typing

In order to define the typing rules and the lookup functions, we extend the sequence notation also to method definitions:

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

$$\mathit{fields}(\texttt{Object}) = \bullet \qquad \mathit{fields}(\langle \texttt{C} \rangle) = \mathit{fields}(\texttt{C})$$

$$\frac{\texttt{class C extends D } \{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K};\ \overline{\texttt{M}}\} \qquad \mathit{fields}(\texttt{D}) = \overline{\texttt{D}}\ \overline{\texttt{g}}}{\mathit{fields}(\texttt{C}) = \overline{\texttt{D}}\ \overline{\texttt{g}}, \overline{\texttt{C}}\ \overline{\texttt{f}}}$$

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

$$\frac{\texttt{class C extends D } \{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K};\ \overline{\texttt{M}}\} \qquad \mathit{sign}(\texttt{D}) = \langle \texttt{S}, \emptyset \rangle}{\mathit{sign}(\texttt{C}) = \langle \mathit{sign}(\overline{\texttt{M}}) \cup \texttt{S}, \emptyset \rangle}$$

$$\frac{\texttt{class C abstracts D } \{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K};\ \overline{\texttt{M}}\ \overline{\texttt{N}}\ \overline{\texttt{R}}\} \qquad \mathit{sign}(\texttt{D}) = \langle \texttt{S}_1, \texttt{S}_2 \rangle}{\mathit{sign}(\texttt{C}) = \langle \mathit{sign}(\overline{\texttt{M}}) \cup (\texttt{S}_1 - (\mathit{sign}(\overline{\texttt{N}}) \cup \mathit{sign}(\overline{\texttt{R}}))), \mathit{sign}(\overline{\texttt{N}}) \cup \mathit{sign}(\overline{\texttt{R}}) \cup (\texttt{S}_2 - \mathit{sign}(\overline{\texttt{M}})) \rangle}$$

$$\frac{\mathit{sign}(\texttt{C}) = \langle \texttt{S}_1, \texttt{S}_2 \rangle}{\mathit{sign}(\langle \texttt{C} \rangle) = \langle \texttt{S}_1 \cup \texttt{S}_2, \emptyset \rangle} \qquad \frac{\texttt{m} : \overline{\texttt{B}} \to \texttt{B} \in \texttt{S}}{\mathit{mtype}(\texttt{m}, \texttt{S}) = \overline{\texttt{B}} \to \texttt{B}}$$

$$\frac{\mathit{sign}(\texttt{T}) = \langle \texttt{S}_1, \texttt{S}_2 \rangle}{\mathit{mtype}(\texttt{m}, \texttt{T}) = \mathit{mtype}(\texttt{m}, \texttt{S}_1 \cup \texttt{S}_2)}$$

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

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

$$\frac{\texttt{class C abstracts D } \{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K};\ \overline{\texttt{M}}\ \overline{\texttt{N}}\ \overline{\texttt{R}}\} \qquad \texttt{B m } (\overline{\texttt{B}}\ \overline{\texttt{x}})\{\texttt{return e;}\} \in \overline{\texttt{M}} \text{ or } \texttt{redef B m } (\overline{\texttt{B}}\ \overline{\texttt{x}})\{\texttt{return e;}\} \in \overline{\texttt{R}}}{\mathit{mbody}(\texttt{m}, \texttt{C}) = (\overline{\texttt{x}}, \texttt{e})}$$

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

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

Fig. 4. Lookup functions.

represents a sequence of method definitions:

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

The signatures of the above method definitions will be denoted, in a compact form, by $\overline{\texttt{m} : \overline{\texttt{C}} \to \texttt{C}}$ or simply by $\mathit{sign}(\overline{\texttt{M}})$. The same convention will be used for redefining methods and for abstract method definitions (and their corresponding signatures). To lighten the notation, in the following we will assume a fixed class table *CT* and then $<:$ is the subtype relation induced by *CT*. We will write $\overline{\texttt{C}} <: \overline{\texttt{D}}$ as a shorthand for $\texttt{C}_1 <: \texttt{D}_1 \wedge \cdots \wedge \texttt{C}_n <: \texttt{D}_n$.

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

A *signature set*, denoted by S, is a set of method signatures of the shape $\texttt{m} : \overline{\texttt{C}} \to \texttt{C}$. The *signature* of a class C, denoted by $\mathit{sign}(\texttt{C})$, is a pair of signature sets $\langle \texttt{S}_1, \texttt{S}_2 \rangle$, where the first set is the signature set of the complete methods and the second set is the signature set of the incomplete methods (both abstract and redefining). Of course, for standard classes, the second set will be empty.

The lookup function $\mathit{fields}(\texttt{C})$ returns the sequence of the field names, together with the corresponding types, for all the fields declared in C and in its superclasses. The $\mathit{mtype}(\texttt{m}, \texttt{C})$ lookup function (where m is the method name we are looking for, and C is the class where we are performing the lookup) differs from the one of FJ in that it relies on the new lookup function *sign*; the lookup function $\mathit{sign}(\texttt{C})$ returns the signature of the class C by inspecting the signatures of its methods. In particular, since the superclass D of an incomplete class C can be, in turn, an incomplete class, the methods that are complete are those defined in C and those defined in D that are not made incomplete by C (i.e., $\mathit{sign}(\overline{\texttt{M}}) \cup (\texttt{S}_1 - (\mathit{sign}(\overline{\texttt{N}}) \cup \mathit{sign}(\overline{\texttt{R}})))$); conversely, the incomplete methods are the incomplete methods of C and those of D that are not defined in C (i.e., $\mathit{sign}(\overline{\texttt{N}}) \cup \mathit{sign}(\overline{\texttt{R}}) \cup (\texttt{S}_2 - \mathit{sign}(\overline{\texttt{M}}))$). Moreover, for a composed object of type $\langle \texttt{C} \rangle$, *sign* returns a signature where the first element is the union of the signature sets of its class and the second element is made empty; this reflects the fact that all the methods of the object are considered concrete. Since we introduced this lookup function, the definition of *mtype* is

***concrete* predicate**

$$\frac{\mathit{sign}(\mathtt{T}) = \langle \mathtt{S}, \emptyset \rangle}{\mathit{concrete}(\mathtt{T})} \qquad \mathit{concrete}(\mathtt{S})$$

Expression typing

$$\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x}) \tag{T-Var}$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{T} \qquad \mathit{fields}(\mathtt{T}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}} \qquad \mathit{concrete}(\mathtt{T})}{\Gamma \vdash \mathtt{e.f}_i : \mathtt{C}_i} \tag{T-Field}$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{T} \qquad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}} \qquad \mathit{mtype}(\mathtt{m}, \mathtt{T}) = \overline{\mathtt{B}} \to \mathtt{B} \qquad \overline{\mathtt{T}} <: \overline{\mathtt{B}} \qquad \mathit{concrete}(\mathtt{T})}{\Gamma \vdash \mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{B}} \tag{T-Invk}$$

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{f}} \qquad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}} \qquad \overline{\mathtt{T}} <: \overline{\mathtt{D}}}{\Gamma \vdash \mathtt{new\ C}(\overline{\mathtt{e}}) : \mathtt{C}} \tag{T-New}$$

$$\frac{\Gamma \vdash \mathtt{e}_1 : \mathtt{C} \qquad \mathit{sign}(\mathtt{C}) = \langle \mathtt{S}_1, \mathtt{S}_2 \rangle \qquad \mathtt{S}_2 \neq \emptyset \qquad \Gamma \vdash \mathtt{e}_2 : \mathtt{T} \qquad \mathit{sign}(\mathtt{T}) = \langle \mathtt{S}'_1, \emptyset \rangle \qquad \mathtt{S}_2 \subseteq \mathtt{S}'_1}{\Gamma \vdash \mathtt{e}_1 \leftarrow\!\!+ \mathtt{e}_2 : \langle \mathtt{C} \rangle} \tag{T-Comp}$$

***override* predicate**

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

Method and Class typing

$$\frac{\overline{\mathtt{x}} : \overline{\mathtt{B}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{e} : \mathtt{T} \qquad \mathtt{T} <: \mathtt{B} \qquad \mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K};\ \overline{\mathtt{M}}\} \qquad \mathit{override}(\mathtt{m}, \mathtt{D}, \overline{\mathtt{B}} \to \mathtt{B})}{\mathtt{B\ m}\ (\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\mathtt{return\ e;}\}\ \mathtt{OK\ IN\ C}} \tag{T-Method}$$

$$\frac{\overline{\mathtt{x}} : \overline{\mathtt{B}}, \mathtt{this} : \langle \mathtt{C} \rangle \vdash \mathtt{e} : \mathtt{T} \qquad \mathtt{T} <: \mathtt{B} \qquad \mathtt{class\ C\ abstracts\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K};\ \overline{\mathtt{M}}\ \overline{\mathtt{N}}\ \overline{\mathtt{R}}\} \qquad \mathit{override}(\mathtt{m}, \mathtt{D}, \overline{\mathtt{B}} \to \mathtt{B})}{\mathtt{B\ m}\ (\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\mathtt{return\ e;}\}\ \mathtt{OK\ IN\ C}} \tag{T-MethodA}$$

$$\frac{\mathtt{class\ C\ abstracts\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K};\ \overline{\mathtt{M}}\ \overline{\mathtt{N}}\ \overline{\mathtt{R}}\} \qquad \mathit{override}(\mathtt{m}, \mathtt{D}, \overline{\mathtt{B}} \to \mathtt{B})}{\mathtt{B\ m}\ (\overline{\mathtt{B}}\ \overline{\mathtt{x}});\ \mathtt{OK\ IN\ C}} \tag{T-AMethod}$$

$$\frac{\mathit{sign}(\mathtt{C}) = \langle \mathtt{S}_1, \mathtt{S}_2 \rangle \qquad \overline{\mathtt{x}} : \overline{\mathtt{B}}, \mathtt{this} : \langle \mathtt{C} \rangle, \mathtt{next} : \mathtt{S}_2 \vdash \mathtt{e} : \mathtt{E} \qquad \mathtt{E} <: \mathtt{B} \qquad \mathtt{class\ C\ abstracts\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K};\ \overline{\mathtt{M}}\ \overline{\mathtt{N}}\ \overline{\mathtt{R}}\} \qquad \mathit{override}(\mathtt{m}, \mathtt{D}, \overline{\mathtt{B}} \to \mathtt{B})}{\mathtt{redef\ B\ m}\ (\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\mathtt{return\ e;}\}\ \mathtt{OK\ IN\ C}} \tag{T-RMethod}$$

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

$$\frac{\mathtt{K} = \mathtt{C}(\overline{\mathtt{D}}\ \overline{\mathtt{g}}, \overline{\mathtt{C}}\ \overline{\mathtt{f}})\{\mathtt{super}(\overline{\mathtt{g}});\ \mathtt{this.}\overline{\mathtt{f}} = \overline{\mathtt{f}};\} \qquad \mathit{fields}(\mathtt{D}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}} \qquad \overline{\mathtt{M}}\ \mathtt{OK\ IN\ C} \qquad \overline{\mathtt{N}}\ \mathtt{OK\ IN\ C} \qquad \overline{\mathtt{R}}\ \mathtt{OK\ IN\ C}}{\mathtt{class\ C\ abstracts\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K};\ \overline{\mathtt{M}}\ \overline{\mathtt{N}}\ \overline{\mathtt{R}}\}\ \mathtt{OK}} \tag{T-AClass}$$

Fig. 5. Typing rules.

straightforward (with respect to the one of FJ [25]). Moreover, note that *mtype* is defined for both C and ⟨C⟩ and also for a method signature S; the last case is for handling the `next` variable, whose type is a signature set (see later in this section). Since *mtype* is the only lookup function defined on a signature set, it is not possible to perform field selection on `next`. The lookup function for method bodies, *mbody*, is basically the same of FJ extended to deal with incomplete classes (note that it returns an empty element • for abstract methods).

A type judgment of the form $\Gamma \vdash \mathtt{e} : \mathtt{T}$ states that "e has type T in the type environment Γ". A type environment is a finite mapping from variables (including `this` and `next`) to types, written $\overline{\mathtt{x}} : \overline{\mathtt{T}}$. Again, we use the sequence notation for abbreviating $\Gamma \vdash \mathtt{e}_1 : \mathtt{T}_1, \ldots, \Gamma \vdash \mathtt{e}_n : \mathtt{T}_n$ to $\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}}$. In order to treat the `next` special variable uniformly, we extend the set of types with signature sets (i.e., T ranges over class names, completed types and signature sets).

Typing rules (Fig. 5) are adapted from those of FJ in order to handle incomplete objects and object composition. In particular, field selection and method selection are allowed only on objects of concrete types, where a *concrete* type is either a standard class C, ⟨C⟩ or a signature set S (used for the case of `next` as explained below). The key rule (T-Comp) for dealing

with object composition is introduced. It checks that the left expression is actually an incomplete object ($S_2 \neq \emptyset$), and that the right one is a complete object that provides all the methods needed by the incomplete object. Note that the final type is the concrete type based on the original class of the incomplete object (we could have chosen the final type to be a structural combination of the types of the objects taking part in the composition, but our design choice is more suited to a nominal setting). This rule also shows that the typing of $\leftarrow\!\!+$ is structural, which is a key feature of the system, since it enhances the flexibility of object composition.

Also typing rules for methods and classes of FJ are adapted to deal with incomplete classes (we use the *override* predicate of [31] to check that the signature of a method is preserved by method overriding). When typing a method and a redefining method in an incomplete class C (with rules (T-MethodA) and (T-RMethod), respectively), we cannot simply assume C for the type of `this`, since we would not be able to type any method invocation on `this` in the methods of C (in fact, the rule (T-Invk) would fail since *concrete*(C) does not hold). Although we prohibit to invoke methods on incomplete objects, it is still safe to accept method invocations on `this` inside an incomplete class, since, at run time, `this` will be replaced by a complete object; thus, we will assume ⟨C⟩ for the type of `this`. This type assumption is also useful to consistently type `this` when it is passed to a method as an argument, used in an object composition, etc. For instance, consider the following incomplete class C (where D is a standard class):

```
class C abstracts D {
    void m(C x); void n(D x); void p() { this.m(this); } void q() { this.n(this); }
}
```

The method *p* is not well typed since we are attempting to pass `this` : ⟨C⟩ to a method expecting the incomplete type C; on the contrary, *q* is well typed since ⟨C⟩ <: D.

Note that, in order to type a `redef` method, we also need to assume a type for `next` when typing its body; it is safe to assume it has the signature set S_2, i.e., the signature set of incomplete methods. This is consistent with the way `next` is bound in the operational semantic rule for redefined method invocation (see Section 5). As noted before, thanks to the way lookup functions are defined (Fig. 4), the only operation that is possible on `next` is method invocation. Furthermore, the type of `next`, being a signature set, is always considered a concrete type, when typing method invocations.

Rule (T-Class) checks that a concrete class extends another concrete class and (T-AClass) checks that also the signatures of incomplete methods satisfy the *override* predicate.

Finally, although we extended the syntax of types to include also signature sets, we will use such types only to type `next` inside method bodies, and, in a well-typed expression, `next` can appear only as the receiver of a message: thus, expressions of the form `next.f` or `e.m(next)` cannot be type checked. This is due to the fact that the subtyping is not defined between a signature set and a class name or a complete type, and signature sets cannot be used by the programmer to write types. On the contrary, *mtype*, which is used in (T-Invk), is defined also for signature sets. Thus, by a straightforward inspection of the typing rules we have the following property.

Property 4.1 (*Occurrences of* `next`)**.** *In a well-typed expression,* `next` *can only appear as the receiver of a message.*

5. Operational semantics

The operational semantics, shown in Fig. 6, is defined by the reduction relation $\mathtt{e} \longrightarrow \mathtt{e}'$, read "e reduces to e′ in one step". The standard reflexive and transitive closure of $\longrightarrow$, denoted by $\longrightarrow^\star$, defines the reduction relation in many steps. We adopt a deterministic call-by-value semantics, analogous to the call-by-value strategy of FJ [31]. The congruence rules formalize how operators (method invocation, object creation, object composition and field selection) are reduced only when all their subexpressions are reduced to values (call-by-value).

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

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

Definition 5.1 (*Annotation Function*)**.** The annotation of e with respect to Γ, denoted by $\mathcal{A}[\![\mathtt{e}]\!]_\Gamma$, is defined on the syntax of e, by case analysis:

- $\mathcal{A}[\![\mathtt{x}]\!]_\Gamma = \mathtt{x}$;
- $\mathcal{A}[\![\mathtt{e.f}]\!]_\Gamma = \mathcal{A}[\![\mathtt{e}]\!]_\Gamma.\mathtt{f}$;
- $\mathcal{A}[\![\mathtt{e.m}(\overline{\mathtt{e}})]\!]_\Gamma = \mathcal{A}[\![\mathtt{e}]\!]_\Gamma.\mathtt{m}(\mathcal{A}[\![\overline{\mathtt{e}}]\!]_\Gamma)^{\overline{\mathtt{B}}\to\mathtt{B}}$ if $\Gamma \vdash \mathtt{e} : \mathtt{T}$ and $\mathit{mtype}(\mathtt{m}, \mathtt{T}) = \overline{\mathtt{B}} \to \mathtt{B}$;
- $\mathcal{A}[\![\mathtt{new\ C}(\overline{\mathtt{e}})]\!]_\Gamma = \mathtt{new\ C}(\mathcal{A}[\![\overline{\mathtt{e}}]\!]_\Gamma)$.

Given a method definition $\mathtt{B\ m}\ (\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\mathtt{return\ e;}\}$, in a class C, the annotation of the method body e is defined as $\mathcal{A}[\![\mathtt{e}]\!]_{\overline{\mathtt{x}}:\overline{\mathtt{B}},\mathtt{this:C}}$.

Redefining set

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

Reduction

$$\mathtt{new\ C}(\overline{\mathtt{v}}) \longrightarrow \langle \mathtt{new\ C}(\overline{\mathtt{v}}) :: \epsilon, \mathtt{new\ C}(\overline{\mathtt{v}}) :: \epsilon \rangle \tag{R-New}$$

$$\langle \mathtt{new\ C}(\overline{\mathtt{v}}) :: \epsilon, \mathtt{new\ C}(\overline{\mathtt{v}}) :: \epsilon \rangle \leftarrow\!\!+ \langle \mathtt{l}, \mathtt{l} \rangle \longrightarrow \langle \mathtt{new\ C}(\overline{\mathtt{v}}) :: \mathtt{l}, \mathtt{new\ C}(\overline{\mathtt{v}}) :: \mathtt{l} \rangle \tag{R-Comp}$$

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

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

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

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

Congruence rules

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

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

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

Fig. 6. Semantics of IFJ.

Given a method redefinition `redef B m` $(\overline{\mathtt{B}}\ \overline{\mathtt{x}})$`{return e; }`, in a class C, the annotation of the method body e is defined as

$\mathcal{A}[\![\mathtt{e}]\!]_{\overline{\mathtt{x}}:\overline{\mathtt{B}},\mathtt{this}:\mathtt{C},\mathtt{next}:\mathtt{S}_2}$ where $\mathit{sign}(\mathtt{C}) = \langle \mathtt{S}_1, \mathtt{S}_2 \rangle$.

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

In the following we will use e (and $\overline{\mathtt{e}}$) also for annotated expressions where not ambiguous.

In order to represent run-time objects, we use lists of standard FJ objects, of the shape $\mathtt{new\ C}(\overline{\mathtt{v}})$; moreover, in order to treat composed objects and standard objects uniformly, we represent a standard object with a list of only one element, $\mathtt{new\ C}(\overline{\mathtt{v}}) :: \epsilon$. The main idea of the semantics of method invocation is to search for the method definition in the (class of the) head of the list using the *mbody* lookup function. If this is found, by rule (R-Invk), then the method body is executed; otherwise, by rule (R-DInvk), the search continues on the following element of the list (of course, in a well-typed program, this search will succeed eventually). However, in order to implement delegation, we need also to keep the original complete composed object so that we can bind `this` correctly; this is the reason why we represent a run-time object as a pair of two object lists: the first one is used for searching for the method, while the second one is the entire composed object. In the following we show how to perform the binding of `this` in the method body correctly. The expression $[\overline{\mathtt{x}} \leftarrow \overline{\mathtt{u}}, \mathtt{this} \Leftarrow \langle \mathtt{new\ C}(\overline{\mathtt{v}}) :: \mathtt{l}, \mathtt{l}' \rangle]\mathtt{e}$ denotes the expression obtained from e by replacing $\mathtt{x}_1$ with $\mathtt{u}_1, \ldots, \mathtt{x}_n$ with $\mathtt{u}_n$ and `this` with $\langle \mathtt{new\ C}(\overline{\mathtt{v}}) :: \mathtt{l}, \mathtt{l}' \rangle$ using the substitution of Definition 5.3. For redefining methods we also replace `next`, using the standard replacement (rule (R-RInvk)).

Definition 5.2 (*findredef*)**.** Given a method name, an object list and a method type we define:

1. $\mathit{findredef}(\mathtt{m}, \epsilon, \overline{\mathtt{B}} \to \mathtt{B}) = \emptyset$;

```
class A abstracts ... {
    void m();
    void n(int i) {... m(); ...}
}
```

```
class B extends ... {
    void m() { if (n("foo")) ... }
    boolean n(String s) {...}
}
```

Listing 4: An incomplete class and a complete one (with similar homonymous methods).

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

Definition 5.3 ($\mathtt{this} \Leftarrow \langle \mathtt{l}, \mathtt{l}' \rangle$)**.** We define the substitution $\mathtt{this} \Leftarrow \langle \mathtt{l}, \mathtt{l}' \rangle$ on IFJ expressions as follows:

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

In order to understand how the binding of `this` works let us consider the following question: What object do we substitute for `this` in the method body? This is a crucial issue in order to perform field selection and method invocations on `this` correctly (and avoid the program getting stuck). Furthermore, we must apply delegation to methods that are redefined, but we also need to take care of possible name clashes among methods with different signatures (for these methods we must not apply delegation, otherwise the method invocation would not be sound).

First of all, field selections in a method body expect to deal with an object of the class where the field is defined (or of a subclass). Thus, it is sensible to substitute `this` with the sublist whose head $\mathtt{new\ C}(\overline{\mathtt{v}})$ is such that $mbody(\mathtt{m}, \mathtt{C})$ is defined. Thus, the first list implements the scope of `this` inside a method body: the scope is restricted to the visibility provided by the class where the method is defined. Another crucial case is when `this` occurs as a right-hand side expression, e.g., when passing this as a method argument; in this case the natural meaning is to refer to the current object "up to now" in the list, and the second list is useless in this context, thus we perform the substitution $[\mathtt{this} \Leftarrow \langle \mathtt{l}, \mathtt{l}' \rangle]\mathtt{this} = \langle \mathtt{l}, \mathtt{l} \rangle$. This is also consistent with reduction rule (R-Comp), where we require that the complete object consists of two identical lists (thus the substitution guarantees the type preservation and the progress property): in fact, we can use `this` in an object composition $\mathtt{e} \leftarrow\!\!+ \mathtt{this}$.

Concerning method invocation, we must take into consideration possible ambiguities due to method name clashes. Suppose we have the classes in Listing 4: an incomplete class *A* that requires a method *m* and defines a method *n*. An instance of *A* can be composed with an object that provides *m*, say an object of class *B* that also defines a method *n*, but with a different signature (recall the stream example in Section 3). When we invoke *m* on the composed object, we actually execute the definition of *m* in *B*; if this method then invokes *n*, the definition of *n* in *B* must be executed (executing the version in *A* would not be sound). This is consistent with the typing that has checked the invocation of *n* in *B.m* using the signature of *B.n*. On the contrary, if the definition of *n* has the same signature as in *B*, as in

```
class A abstracts ... {
    void m();
    redef boolean n(String s) {...}
}
```

then we must execute the version of *A* (according to the semantics of delegation) only if in *A n* is redefining. This method selection strategy is implemented by point 3 of Definition 5.3 by relying on the function *findredef* (Definition 5.2): given a method name, an object list and a method signature, $findredef(\mathtt{m}, \mathtt{new\ C}(\overline{\mathtt{v}}) :: \mathtt{l}, \overline{\mathtt{B}} \to \mathtt{B})$ searches for the object in the list that redefines `m` (in particular, it checks whether in the class of the head of the list `m` is redefining, i.e., $\mathtt{m} \in \mathtt{redef(C)} \wedge \overline{\mathtt{B}} \to \mathtt{B} = mtype(\mathtt{m}, \mathtt{C})$, otherwise it performs a recursive lookup in the tail of the list). The key point in *findredef* is the static annotation that guides the search in a type-safe way (as shown in Section 6). If the search succeeds, then we replace `this` with the sublist returned by *findredef* (this corresponds to the delegation mechanism of replacing `this` with the original sender); otherwise, we replace `this` with the head of the scanned list, since that method was not intended to be redefined.

It is important to notice that objects of the shape $\langle \mathtt{l}, \mathtt{l}' \rangle$ where $\mathtt{l} \neq \mathtt{l}'$ appear only as message receivers and they are produced only during method invocations; thus, the actual values produced by programs (returned by methods, passed to methods, used in object compositions, etc.) can only be of the shape $\langle \mathtt{l}, \mathtt{l} \rangle$. This is formally proved in Theorem 6.14.

5.1. Dealing with imperative features

Our language is based on FJ that abstracts the functional core of Java; then, also our approach has a functional flavor, i.e., it does not consider side effects and imperative features. In this section we want to show how the present approach can be adapted to an imperative version of FJ (e.g., using heaps and object identifiers similarly to [11]), based on our run-time representation of objects.

The key point is that, when objects are composed, the resulting object consists of a list of sub-objects; in particular, these sub-objects are not modified by the composition, which always produces a brand new object.

Consider the (R-Comp) rule in Fig. 6:

$$\langle \texttt{new C}(\overline{\texttt{v}}) :: \epsilon, \texttt{new C}(\overline{\texttt{v}}) :: \epsilon \rangle \leftarrow\!\!+ \langle \texttt{l}, \texttt{l} \rangle \longrightarrow \langle \texttt{new C}(\overline{\texttt{v}}) :: \texttt{l}, \texttt{new C}(\overline{\texttt{v}}) :: \texttt{l} \rangle$$

The new object contains the sub-objects without changing them; notice that the removal of $:: \epsilon$ from $\langle \texttt{new C}(\overline{\texttt{v}}) :: \epsilon, \texttt{new C}(\overline{\texttt{v}}) :: \epsilon \rangle$ is due only to our uniform representation for complete and incomplete objects and it has nothing to do with the state of the object itself.

Each object composition creates a brand new object and all the sub-objects are actually shared. For instance, each object composition gets a new object identifier in an imperative model. The code employing object composition in Listing 1 clarifies this point: the (same) complete object `deleg` is used for completing all the three incomplete objects. Actually we could have also written the code as follows:

```
SaveActionDelegate deleg = new SaveActionDelegate();
Action saveMenu = new MenuItem("save") <- deleg;
myFrame.addToMenu(saveMenu);
Action saveButton = new Button("save") <- deleg;
myFrame.addToToolbar(saveButton);
Action saveKeyb = new KeyboardAccel("Ctrl+S") <- deleg;
myFrame.setKeybAcc(saveKeyb);
```

Thus, modifying the internal state of `deleg` will assure that all the actions are updated too. For instance, if the `SaveAction-Delegate` had logging features, we could enable them and disable them during the execution of our program, and all the actions resulting from the object compositions will use logging consistently. Furthermore, this mechanism will assure that there will not be problems when an object is pointed to by references in different parts of the program.

This shows that objects are not only instances of classes (possibly incomplete classes), but they are also prototypes that can be used, via the object composition, to create new objects at run time, while ensuring statically that the composition is type safe. We then can use incomplete and complete objects as our re-usable building blocks to assemble at run time, on the fly, brand new objects.

Finally, not modifying incomplete objects directly also makes them more re-usable especially in cases when object composition may not have to be permanent: the behavior of an object may need to evolve many times during the execution of a program and the missing methods provided during an object composition might need to be changed, e.g., because the state of the incomplete object has changed. Since the original incomplete object is not modified, then it can be re-used smoothly in many object compositions during the evolution of a program.

5.2. Examples of reduction

In this section we present some examples of reductions, based on the code of Section 3; these will help understanding the operational semantics, and, in particular, how the run-time behaves when dealing with composed objects and required methods (as in Section 3 we will use basic types, and **void**).

We recall that an incomplete object can redefine a method of the complete one in an object composition and that the new version of the method provided by the incomplete object is called *redefining*, while the old one in the complete object is called *redefined*; the redefining method body can refer to the redefined one via the special variable `next`.

5.2.1. Graphical widgets

Let us first consider the code of Listing 1, and the method invocation (for simplicity we do not consider constructor arguments, and since annotations are not interesting in this example, we will omit them to keep the presentation simple):

$$(\texttt{new } \mathit{MenuItem}() \leftarrow\!\!+ \mathit{deleg}).\mathit{run}()$$

where $\mathit{deleg} = \texttt{new } \mathit{SaveActionDelegate}()$. By using (R-New) and (R-Comp) we get (we do not write the list tail $:: \epsilon$)

$$\langle \texttt{new } \mathit{MenuItem}() :: \mathit{deleg}, \texttt{new } \mathit{MenuItem}() :: \mathit{deleg} \rangle.\mathit{run}()$$

Now, since `run` is an abstract method in `MenuItem`, then we apply (R-DInvk) (we abbreviate $\texttt{l} = \texttt{new } \mathit{MenuItem}() :: \mathit{deleg}$):

$$\langle \mathit{deleg}, \texttt{l} \rangle.\mathit{run}()$$

The method `run` is defined in `SaveActionDelegate` and it is not a redefining method, therefore we can apply (R-Invk), thus we select the body of `run` in `SaveActionDelegate`. The only interesting instruction in the body

```
class Inc1 abstracts ... {
    void m() { n(10); }
    void n(int i);
}

class Inc2 abstracts ... {
    void m() { n(10); }
    redef void n(int i) {
        next.n(i);
        ...
    }
}

class Inc3 abstracts ... {
    void m() { n(true); }
    redef void n(boolean b) {
        next.n(b);
        ...
    }
}

class Inc4 abstracts ... {
    redef void m() { next.m(10); }
    void n(int i);
}

class C1 {
    void m() { ... }
    void n() { m(); }
}

class C2 {
    void m(boolean b) { ... }
    void n() { m(true); }
}
```

Fig. 7. Some incomplete and complete classes.

of `run` is `this.`*enable*(**false**). In order to execute it, we must apply the substitution `this` $\Leftarrow$ $\langle$*deleg*, `l`$\rangle$ (defined in Definition 5.3). In particular, we are in the case 3 of Definition 5.3, where the type annotation is **boolean** $\rightarrow$ **void**. Now, *findredef*(*enable*, `new` *MenuItem*() :: *deleg*, **boolean** $\rightarrow$ **void**) returns `new` *MenuItem*() :: *deleg*, since *enable* is redefining in *MenuItem* and the method signature matches. Thus, after the substitution we get:

$\langle$`new` *MenuItem*() :: *deleg*, `new` *MenuItem*() :: *deleg*$\rangle$.*enable*(**false**)

Now, we can apply (R-RInvk); in particular, the substitution for `next` is `next` $\leftarrow$ $\langle$*deleg*, `l`$\rangle$. Thus, for the redefining method `enable` the semantics of delegation was adopted (and `next` is correctly bound).

5.2.2. Streams

Now, let us consider the stream example (Listing 3), in particular, the invocation of method `read` on the composed object

Stream myStream = **new** *CompressStream*() <- (**new** *BufferedStream*() <- **new** *FileStream*(`"foo.txt"`));

Since `read` is a redefining method in `CompressStream`, we use (R-RInvk), and `next` is bound to

$\langle$`new` *BufferedStream*() :: `new` *FileStream*(), `new` *CompressStream*() :: `new` *BufferedStream*() :: `new` *FileStream*()$\rangle$

The semantics is call-by-value, thus, we first execute `read` in `BufferedStream`. The substitution of `this` in the expression `this.`*readBuffer*()$^{\textbf{void}\rightarrow\textbf{byte}[]}$ is crucial; when we call *findredef*(*readBuffer*, `new` *CompressStream*() :: `new` *BufferedStream*() :: `new` *FileStream*(), **void** $\rightarrow$ **byte**[]) we use the "otherwise" case of Definition 5.2-2, since *readBuffer* is not redefining in *CompressStream*, and on the recursive invocation, *findredef* will correctly return `new` *BufferedStream*() :: `new` *FileStream*(), so that, inside `read` in `BufferedStream`, we correctly use `readBuffer` in `BufferedStream`.

If we had another object in front of the object composition, say `MyStream` where **void** *readBuffer*(**int**, **byte**[]) is redefining, *findredef* would still correctly return `new` *BufferedStream*() :: `new` *FileStream*(): even though `readBuffer` is redefining in `MyStream` still its signature would not match the type annotation **void** $\rightarrow$ **byte**[].

5.2.3. Other examples

We conclude this section by considering some possible object composition configurations and the corresponding method lookup executions. We consider the incomplete classes in Fig. 7 (we omit the base classes, not relevant in this example), and we assume that we already instantiated some corresponding objects `inc1`, …, `inc4`, `c1`, `c2`.[6] We consider some possible compositions and the semantics of the invocation of *m* on the corresponding composed object:

- (`inc1` $\leftarrow\!\!+$ `c1`).`m`(): method *n* is abstract for `inc1`, therefore the invocation of *n* inside *m* is delegated to `c1`. Then, the invocation of *m* inside *n* selects the definition of `C1` since *m* is not redefining in class `Inc1` (that is, `Inc1` does not provide a new implementation for *m*); otherwise, although the selection of *m* in `Inc1` would have been type safe, the program would loop;
- (`inc2` $\leftarrow\!\!+$ `c1`).`m`(): similarly to the previous case, the invocation of *m* inside *n* selects the definition for *m* of `C1`;
- (`inc3` $\leftarrow\!\!+$ `c1`).`m`(): this composition is rejected by the compiler since it is not well typed (the signature of the redefining method *n*, `boolean`, does not match the one for *n* in the complete object, which is `int`);

[6] We name them with the same name of their class but with a lowercase letter at the beginning.

$$\frac{\Gamma \vdash \texttt{new C}(\overline{\texttt{v}}) : \texttt{C}}{\Gamma \vdash \texttt{new C}(\overline{\texttt{v}}) :: \epsilon : \texttt{C}} \quad \text{(T-ListH)}$$

$$\frac{\Gamma \vdash \texttt{new C}(\overline{\texttt{v}}) : \texttt{C} \quad \mathit{sign}(\texttt{C}) = \langle \texttt{S}_1, \texttt{S}_2 \rangle \quad \texttt{S}_2 \neq \emptyset \quad \Gamma \vdash \texttt{l} : \texttt{T} \quad \mathit{sign}(\texttt{T}) = \langle \texttt{S}'_1, \emptyset \rangle \quad \texttt{S}_2 \subseteq \texttt{S}'_1}{\Gamma \vdash \texttt{new C}(\overline{\texttt{v}}) :: \texttt{l} : \langle \texttt{C} \rangle} \quad \text{(T-List)}$$

$$\frac{\Gamma \vdash \texttt{l} : \texttt{T} \quad \Gamma \vdash \texttt{l}' : \texttt{T}' \quad \texttt{l} \subseteq \texttt{l}'}{\Gamma \vdash \langle \texttt{l}, \texttt{l}' \rangle : \texttt{T}} \quad \text{(T-ORunTime)}$$

$$\frac{\Gamma \vdash \texttt{e} : \texttt{T} \quad \Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{T}} \quad \mathit{mtype}(\texttt{m}, \texttt{T}) = \overline{\texttt{B}} \to \texttt{B} \quad \overline{\texttt{T}} <: \overline{\texttt{B}} \quad \mathit{concrete}(\texttt{T})}{\Gamma \vdash \texttt{e.m}(\overline{\texttt{e}})^{\overline{\texttt{B}} \to \texttt{B}} : \texttt{B}} \quad \text{(TA-Invk)}$$

Fig. 8. Run-time expression typing.

- (inc4 ↚ (inc1 ↚ c2)).m(): we have the following method selection sequence [7]: Inc4.m, Inc1.m, C2.n, C2.m, in fact, even though *m* is redefining in Inc4, it does not have the same signature of the definition of *m* in C2, therefore that redefinition is ignored;
- (inc4 ↚ (inc1 ↚ c1)).m(): we have the following method selections: Inc4.m, Inc1.m, C1.n, Inc4.m and the program loops, in fact, even though *m* is not redefining in Inc1, it is redefining in Inc4 with the matching type.

The last case is an example of a loop due to the delegation, and we will get back to this scenario in Section 7.

6. Properties

This section is devoted to prove the type safety property for IFJ, which means that no message-not-understood errors can occur at run time during method invocations on composed objects. Before giving the main proofs of standard type preservation and progress theorems, we develop some required lemmas and properties.

First of all, we have to define the typing rules for annotated run-time expressions. Three additional rules are needed for typing our run-time representations of objects, that is, lists and pairs of lists. Moreover, the rule for method invocation needs to be adapted to take into consideration method call annotations. We present these new rules in Fig. 8, while all the other typing rules are the same as the ones presented in Fig. 5, where e is intended to denote a run-time expression. We note that the type of a composed object is the one of the head of the list, consistently with the typing rule for object composition. Moreover, rule (T-List) implicitly requires that $\texttt{l} \neq \epsilon$ since ϵ is not typable.

We use $|\texttt{l}|$ to denote the length of the list l and the following definition of list inclusion.

Definition 6.1 (*List Inclusion*)**.** Let $\texttt{l}_1$ and $\texttt{l}_2$ be two well-typed object lists; we say that $\texttt{l}_1$ is included in $\texttt{l}_2$, denoted by $\texttt{l}_1 \subseteq \texttt{l}_2$, if and only if

1. either $\texttt{l}_1 = \texttt{l}_2$, or
2. $\texttt{l}_2 = \texttt{l}'_1 :: \texttt{l}_1$, with $\texttt{l}'_1 \neq \epsilon$.

By the definition of annotation (Definition 5.1), it is also easy to verify the following property.

Property 6.2 (*Types are Preserved Under Annotation*)**.** *If* $\Gamma \vdash \texttt{e} : \texttt{T}$ *for some* Γ *and* T*, then* $\Gamma \vdash \mathcal{A}[\![\texttt{e}]\!]_\Gamma : \texttt{T}$.

Thus, in the following, to make the presentation of the properties simpler, we will not write the annotations explicitly when they are not significant, and we will use e to refer to annotated run-time expressions.

Property 6.3. *Let* $\texttt{l} = \texttt{new C}(\overline{\texttt{v}}) :: \texttt{l}'$ *such that* $\texttt{l}' \neq \epsilon$ *and* $\Gamma \vdash \texttt{l} : \texttt{T}$*. Then:*

1. *concrete*(T) *holds.*
2. $\Gamma \vdash \texttt{l}' : \texttt{T}'$ *for some* T′ *such that concrete*(T′) *holds.*
3. *If sign*(C) = ⟨S1, S2⟩*, for any* m *such that* $\texttt{m} : \overline{\texttt{B}} \to \texttt{B} \in \texttt{S}_2$ *then mtype*(m, T′) = $\overline{\texttt{B}} \to \texttt{B}$.

Proof. It follows directly from rule (T-List). □

Corollary 6.4. *Let* $\texttt{l} = \texttt{new C}(\overline{\texttt{v}}) :: \texttt{l}'$ *such that* $\Gamma \vdash \texttt{l} : \texttt{T}$*. Then, for any* $\texttt{l}'' \subseteq \texttt{l}'$ *such that* $\texttt{l}'' \neq \epsilon$, $\Gamma \vdash \texttt{l}'' : \texttt{T}''$ *and concrete*(T″).

Proof. By iterate applications of Property 6.3-1,2. □

Property 6.5 (*Well-Typedness of List Inclusion*)**.** 1. *If* $\Gamma \vdash \texttt{l}' : \texttt{T}'$ *and concrete*(T′) *then* $\Gamma \vdash \langle \texttt{l}, \texttt{l}' \rangle : \texttt{T}$*, for some* T *where concrete*(T)*, for any* $\texttt{l} \subseteq \texttt{l}'$.
2. *If* $\Gamma \vdash \langle \texttt{l}, \texttt{l}' \rangle : \texttt{T}$ *where concrete*(T)*, then* $\Gamma \vdash \texttt{l}' : \texttt{T}'$ *for some* T′ *where concrete*(T′).

[7] In order to refer to a specific version of a method in a class, we use the fully qualified method notation, thus, e.g., A.m denotes the definition of m in class A.

Proof. 1. By Corollary 6.4 and then by using (T-ORunTime).
2. By (T-ORunTime) we have $\Gamma \vdash \mathtt{l}' : \mathtt{T}'$ for some $\mathtt{T}'$, and $\mathtt{l} \subseteq \mathtt{l}'$. If $\mathtt{l} = \mathtt{l}'$ we have the thesis; otherwise, since $\mathtt{l} \neq \epsilon$, by Property 6.3-1, we have *concrete*($\mathtt{T}'$). □

In order to state the Substitution Lemma, both for standard class methods and for incomplete class (possibly redefining) methods, we need a uniform way to refer to the type of `this`; in fact, as shown in Section 4, in rules (T-MethodA) and (T-RMethod), Fig. 5, `this` is used with a concrete type. Thus we introduce the following notation.

Notation 6.6. *Given a class name* C*, we denote* $[\mathtt{C}] = \mathtt{C}$ *if concrete*(C), $[\mathtt{C}] = \langle \mathtt{C} \rangle$ *otherwise.*

The following lemma states the substitution property for method bodies in class definitions. Then, in the Substitution Lemma, it is sufficient to consider typed expressions which are annotated versions of source code. This lemma is the crucial point for proving the Type Preservation Theorem.

Lemma 6.7 (*Substitution Lemma*)**.** *If* $\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : [\mathtt{C}], \mathtt{next} : \mathtt{S} \vdash \mathtt{e} : \mathtt{T}$*, and*

1. $\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}}'$ *where* $\overline{\mathtt{T}}' <: \overline{\mathtt{C}}$*,*
2. $\Gamma \vdash \mathtt{v} : \mathtt{T}_1$ *where* $\mathtt{T}_1 <: [\mathtt{C}]$*,*
3. $\Gamma \vdash \mathtt{v}' : \mathtt{T}^*$ *where sign*$(\mathtt{T}^*) = \langle \mathtt{S}^*, \emptyset \rangle$ *and* $\mathtt{S} \subseteq \mathtt{S}^*$*,*

then $\Gamma \vdash [\overline{\mathtt{x}} \leftarrow \overline{\mathtt{e}}, \mathtt{this} \Leftarrow \mathtt{v}, \mathtt{next} \leftarrow \mathtt{v}']\mathtt{e} : \mathtt{T}'$ *for some* $\mathtt{T}'$ *such that, either* $\mathtt{T}' <: \mathtt{T}$ *or sign*$(\mathtt{T}') = \langle \mathtt{S}', \emptyset \rangle$ *and* $\mathtt{S} \subseteq \mathtt{S}'$*.*

Proof. By induction on the derivation of $\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : [\mathtt{C}], \mathtt{next} : \mathtt{S} \vdash \mathtt{e} : \mathtt{T}$, with a case analysis on the last applied rule.

(T-Var) If $\mathtt{x} \notin \overline{\mathtt{x}}$ and $\mathtt{x} \neq \mathtt{this}$ and $\mathtt{x} \neq \mathtt{next}$, then the conclusion is immediate since $[\overline{\mathtt{x}} \leftarrow \overline{\mathtt{e}}]\mathtt{x} = \mathtt{x}$; otherwise,
- $\mathtt{x} = \mathtt{x}_i$, $\mathtt{T} = \mathtt{C}_i$, and $[\overline{\mathtt{x}} \leftarrow \overline{\mathtt{e}}]\mathtt{x} = [\overline{\mathtt{x}} \leftarrow \overline{\mathtt{e}}]\mathtt{x}_i = \mathtt{e}_i$; then, by assumption, $\Gamma \vdash \mathtt{e}_i : \mathtt{T}'_i <: \mathtt{C}_i$.
- $\mathtt{x} = \mathtt{this}$ and $\mathtt{T} = [\mathtt{C}]$, let $\mathtt{v} = \langle \mathtt{l}, \mathtt{l}' \rangle$, then $[\mathtt{this} \Leftarrow \mathtt{v}]\mathtt{this} = \langle \mathtt{l}, \mathtt{l} \rangle$ (see the first case of Definition 5.3), and $\Gamma \vdash \langle \mathtt{l}, \mathtt{l} \rangle : \mathtt{T}_1 <: [\mathtt{C}]$.
- $\mathtt{x} = \mathtt{next}$, then $[\mathtt{next} \leftarrow \mathtt{v}']\mathtt{x} = \mathtt{v}'$ and the thesis follows from the hypothesis 3.

(T-Field) By induction hypothesis, taking into account that fields, with their types, are inherited in subclasses and all field names are assumed to be distinct.

(TA-Invk) The thesis follows by the induction hypothesis in all cases but in the most crucial one when
- $\mathtt{e} = \mathtt{this.m}(\overline{\mathtt{e}}_0)^{\overline{\mathtt{B}} \to \mathtt{B}}$,
- $\mathtt{v} = \langle \mathtt{l}, \mathtt{l}' \rangle$,
- *findredef*$(\mathtt{m}, \mathtt{l}', \overline{\mathtt{B}} \to \mathtt{B}) = \mathtt{l}_1$, where $\mathtt{l}_1 \neq \emptyset \neq \mathtt{l}$

that is, when there is an object in the list where `m` is redefining (with the same signature). By hypothesis $\Gamma \vdash \langle \mathtt{l}, \mathtt{l}' \rangle : \mathtt{T}_1 <: [\mathtt{C}]$, thus *concrete*($\mathtt{T}_1$); then, by Property 6.5-2, $\Gamma \vdash \mathtt{l}' : \mathtt{T}''$ and *concrete*($\mathtt{T}''$). By Definitions 5.3 and 5.2 we have $\mathtt{l} \subseteq \mathtt{l}_1 \subseteq \mathtt{l}'$, thus by Property 6.5-1 we have $\Gamma \vdash \langle \mathtt{l}_1, \mathtt{l}' \rangle : \mathtt{T}_*$ and *concrete*($\mathtt{T}_*$).[8] By (TA-Invk) we have

$$\frac{\Gamma \vdash \mathtt{this} : [\mathtt{C}] \quad \Gamma \vdash \overline{\mathtt{e}}_0 : \overline{\mathtt{T}}_0 \qquad \mathit{mtype}(\mathtt{m}, [\mathtt{C}]) = \overline{\mathtt{B}} \to \mathtt{B} \quad \overline{\mathtt{T}}_0 <: \overline{\mathtt{B}} \quad \mathit{concrete}([\mathtt{C}])}{\Gamma \vdash \mathtt{this.m}(\overline{\mathtt{e}}_0)^{\overline{\mathtt{B}} \to \mathtt{B}} : \mathtt{B}}$$

By definition of *findredef*, we have that $\mathit{mtype}(\mathtt{m}, \mathtt{T}_*) = \overline{\mathtt{B}} \to \mathtt{B}$; by induction hypothesis, $\overline{\mathtt{e}}^* = [\overline{\mathtt{x}} \leftarrow \overline{\mathtt{e}}, \mathtt{this} \Leftarrow \mathtt{v}]\overline{\mathtt{e}}_0$ is such that $\Gamma \vdash \overline{\mathtt{e}}^* : \overline{\mathtt{T}}^*$ for some $\overline{\mathtt{T}}^*$ such that $\overline{\mathtt{T}}^* <: \overline{\mathtt{T}}_0$ (note that `next` cannot occur in $\overline{\mathtt{e}}_0$ by Property 4.1). Thus, we can apply (TA-Invk), (using transitivity $<:$):

$$\frac{\Gamma \vdash \langle \mathtt{l}_1, \mathtt{l}' \rangle : \mathtt{T}_* \quad \Gamma \vdash \overline{\mathtt{e}}^* : \overline{\mathtt{T}}^* \qquad \mathit{mtype}(\mathtt{m}, \mathtt{T}_*) = \overline{\mathtt{B}} \to \mathtt{B} \quad \overline{\mathtt{T}}^* <: \overline{\mathtt{B}} \quad \mathit{concrete}(\mathtt{T}_*)}{\Gamma \vdash \langle \mathtt{l}_1, \mathtt{l}' \rangle.\mathtt{m}(\overline{\mathtt{e}}^*)^{\overline{\mathtt{B}} \to \mathtt{B}} : \mathtt{B}}$$

(T-New) It follows from the induction hypothesis on the arguments (where `next` cannot occur by Property 4.1).

(T-Comp) For $\mathtt{e} = \mathtt{e}' \leftarrow\!\!+ \mathtt{e}''$, the last applied rule is (T-Comp) and $\mathtt{T} = \langle \mathtt{D} \rangle$ for some incomplete class `D`:

$$\frac{\Gamma \vdash \mathtt{e}' : \mathtt{D} \quad \mathit{sign}(\mathtt{D}) = \langle \mathtt{S}_1, \mathtt{S}_2 \rangle \quad \mathtt{S}_2 \neq \emptyset \qquad \Gamma \vdash \mathtt{e}'' : \mathtt{T}'' \quad \mathit{sign}(\mathtt{T}'') = \langle \mathtt{S}'_1, \emptyset \rangle \quad \mathtt{S}_2 \subseteq \mathtt{S}'_1}{\Gamma \vdash \mathtt{e}' \leftarrow\!\!+ \mathtt{e}'' : \langle \mathtt{D} \rangle}$$

[8] Note that in this case, we cannot state that $\mathtt{T}_* <: [\mathtt{C}]$ since the classes of the objects in an object composition might be unrelated.

By the induction hypothesis on $\mathtt{e}'$ and $\mathtt{e}''$ (we denote their substituted versions with $\mathtt{e}'_*$ and $\mathtt{e}''_*$, respectively), we have $\Gamma \vdash \mathtt{e}'_* : \mathtt{D}$ (since there is no subtyping on incomplete classes), $\Gamma \vdash \mathtt{e}''_* : \mathtt{T}''_* <: \mathtt{T}''$, thus $\mathit{sign}(\mathtt{T}''_*) = \langle \mathtt{S}'', \emptyset \rangle$ such that $\mathtt{S}'_1 \subseteq \mathtt{S}''$. We get the thesis by applying (T-Comp):

$$\frac{\Gamma \vdash \mathtt{e}'_* : \mathtt{D} \quad \mathit{sign}(\mathtt{D}) = \langle \mathtt{S}_1, \mathtt{S}_2 \rangle \quad \mathtt{S}_2 \neq \emptyset \qquad \Gamma \vdash \mathtt{e}''_* : \mathtt{T}''_* \quad \mathit{sign}(\mathtt{T}''_*) = \langle \mathtt{S}'', \emptyset \rangle \quad \mathtt{S}_2 \subseteq \mathtt{S}''}{\Gamma \vdash \mathtt{e}'_* \leftarrow\!\!+ \mathtt{e}''_* : \langle \mathtt{D} \rangle} \quad \square$$

The following lemma is a standard structural property which is useful in other proofs: it states that we can add to any Γ typing assertions for new variables without changing the typing assertions that can be derived under Γ.

Lemma 6.8 (*Weakening*). *If* $\Gamma \vdash \mathtt{e} : \mathtt{T}$, *then* $\Gamma, \mathtt{x} : \mathtt{T}' \vdash \mathtt{e} : \mathtt{T}$.

Proof. By induction on the derivation of $\Gamma \vdash \mathtt{e} : \mathtt{T}$. $\square$

Lemma 6.9. *If* $\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{B}} \to \mathtt{B}$ *and* $\mathit{mbody}(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e})$, *then, for some* D, *where* $\mathtt{C} = \mathtt{D}$ *or* D *is a superclass of* C, *there exists* $\mathtt{T} <: \mathtt{B}$ *such that:*

1. *If* $\mathtt{m} \notin \mathtt{redef}(\mathtt{C})$, *then* $\overline{\mathtt{x}} : \overline{\mathtt{B}}, \mathtt{this} : [\mathtt{D}] \vdash \mathtt{e} : \mathtt{T}$,
2. *otherwise,* $\overline{\mathtt{x}} : \overline{\mathtt{B}}, \mathtt{this} : [\mathtt{D}], \mathtt{next} : \mathtt{S}_2 \vdash \mathtt{e} : \mathtt{T}$, *where* $\mathit{sign}(\mathtt{D}) = \langle \mathtt{S}_1, \mathtt{S}_2 \rangle$.

Proof. Straightforward induction on the derivation of $\mathit{mbody}(\mathtt{m}, \mathtt{C})$, using (T-Method) or (T-MethodA), depending on the method being defined in a standard or incomplete class, for point 1, and (T-RMethod) for point 2. $\square$

Theorem 6.10 (*Type Preservation*). *If* $\Gamma \vdash \mathtt{e} : \mathtt{T}$ *and* $\mathtt{e} \longrightarrow \mathtt{e}'$ *then* $\Gamma \vdash \mathtt{e}' : \mathtt{T}'$ *for some* $\mathtt{T}' <: \mathtt{T}$.

Proof. By induction on a derivation of $\mathtt{e} \longrightarrow \mathtt{e}'$, with a case analysis on the final rule. We only consider reduction rules, since the property on congruence rules follows straightforwardly by the induction hypothesis.

(R-New) By hypothesis $\Gamma \vdash \mathtt{new\ C}(\overline{\mathtt{e}}) : \mathtt{C}$, thus we can apply (T-ListH) and then (T-ORunTime) to get $\Gamma \vdash \langle \mathtt{new\ C}(\overline{\mathtt{v}}) :: \epsilon, \mathtt{new\ C}(\overline{\mathtt{v}}) :: \epsilon \rangle : \mathtt{C}$.

(R-Comp) By hypothesis $\Gamma \vdash \langle \mathtt{new\ C}(\overline{\mathtt{v}}) :: \epsilon, \mathtt{new\ C}(\overline{\mathtt{v}}) :: \epsilon \rangle \leftarrow\!\!+ \langle \mathtt{l}, \mathtt{l} \rangle : \mathtt{T}$ which can be obtained only by using (T-Comp); by (T-Comp) we get $\Gamma \vdash \langle \mathtt{new\ C}(\overline{\mathtt{v}}) :: \epsilon, \mathtt{new\ C}(\overline{\mathtt{v}}) :: \epsilon \rangle : \mathtt{D}$ (for some D), $\mathit{sign}(\mathtt{D}) = \langle \mathtt{S}_1, \mathtt{S}_2 \rangle$, $\mathtt{S}_2 \neq \emptyset$, $\Gamma \vdash \langle \mathtt{l}, \mathtt{l} \rangle : \mathtt{T}^*$, $\mathit{sign}(\mathtt{T}^*) = \langle \mathtt{S}'_1, \emptyset \rangle$ and $\mathtt{S}_2 \subseteq \mathtt{S}'_1$, and in particular $\mathtt{T} = \langle \mathtt{D} \rangle$. $\Gamma \vdash \langle \mathtt{new\ C}(\overline{\mathtt{v}}) :: \epsilon, \mathtt{new\ C}(\overline{\mathtt{v}}) :: \epsilon \rangle : \mathtt{D}$ can be obtained only by (T-ORunTime) and (T-ListH), i.e., $\Gamma \vdash \mathtt{new\ C}(\overline{\mathtt{e}}) : \mathtt{C} = \mathtt{D}$ (*) and thus $\mathtt{T} = \langle \mathtt{C} \rangle$. Now, by using (*) and the other premises of (T-Comp), we can apply (T-List) to get $\Gamma \vdash \mathtt{new\ C}(\overline{\mathtt{v}}) :: \mathtt{l} : \langle \mathtt{C} \rangle$, and then (T-ORunTime) to get $\Gamma \vdash \langle \mathtt{new\ C}(\overline{\mathtt{v}}) :: \mathtt{l}, \mathtt{new\ C}(\overline{\mathtt{v}}) :: \mathtt{l} \rangle : \langle \mathtt{C} \rangle$.

(R-Field) By hypothesis $\Gamma \vdash \langle \mathtt{new\ C}(\overline{\mathtt{v}}) :: \mathtt{l}, \mathtt{new\ C}(\overline{\mathtt{v}}) :: \mathtt{l} \rangle.\mathtt{f}_i : \mathtt{T}$. By rule (T-Field), we have $\Gamma \vdash \langle \mathtt{new\ C}(\overline{\mathtt{v}}) :: \mathtt{l}, \mathtt{new\ C}(\overline{\mathtt{v}}) :: \mathtt{l} \rangle : \mathtt{T}'$ for some $\mathtt{T}'$, such that $\mathit{concrete}(\mathtt{T}')$ and $\mathit{fields}(\mathtt{T}') = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$, thus $\mathtt{T} = \mathtt{C}_i$. Since $\mathtt{e} \longrightarrow \mathtt{f}_i$, then we have the thesis.

(R-Invk) $\mathtt{e} = (\langle \mathtt{new\ C}(\overline{\mathtt{v}}) :: \mathtt{l}, \mathtt{l}' \rangle).\mathtt{m}(\overline{\mathtt{u}})^{\overline{\mathtt{B}} \to \mathtt{B}}$, $\mathit{mbody}(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e}_0)$ and $\mathtt{m} \notin \mathit{redef}(\mathtt{C})$. By (TA-Invk), we have $\Gamma \vdash \langle \mathtt{new\ C}(\overline{\mathtt{v}}) :: \mathtt{l}, \mathtt{l}' \rangle : \mathtt{T}^*$, $\Gamma \vdash \overline{\mathtt{u}} : \overline{\mathtt{T}}$, $\mathit{mtype}(\mathtt{m}, \mathtt{T}^*) = \overline{\mathtt{B}} \to \mathtt{B}$, $\overline{\mathtt{T}} <: \overline{\mathtt{B}}$ and $\mathit{concrete}(\mathtt{T}^*)$. If $\mathtt{l} = \epsilon$, then by (T-ListH), we have $\mathtt{T}^* = \mathtt{C}$; otherwise, by (T-ORunTime) and (T-ListH) we have $\mathtt{T}^* = \langle \mathtt{C} \rangle$. By Lemmas 6.9-1 and 6.8, $\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{B}}, \mathtt{this} : [\mathtt{D}] \vdash \mathtt{e}_0 : \mathtt{B}' <: \mathtt{B}$ for some D where $\mathtt{C} = \mathtt{D}$ or D is a superclass of C. Since C and D are in subclass relation and since $\mathit{concrete}([\mathtt{D}])$, we have $\mathtt{T}^* <: [\mathtt{D}]$. By Substitution Lemma 6.7, $\Gamma \vdash [\overline{\mathtt{x}} \leftarrow \overline{\mathtt{u}}, \mathtt{this} \Leftarrow \langle \mathtt{new\ C}(\overline{\mathtt{v}}) :: \mathtt{l}, \mathtt{l}' \rangle]\mathtt{e}_0 : \mathtt{T}_0 <: \mathtt{B}' <: \mathtt{B}$.

(R-RInvk) As in the previous case, using Lemma 6.9-2 instead of Lemma 6.9-1.

(R-DInvk) Follows from Property 6.3-2 and Property 6.3-3. $\square$

Lemma 6.11. *If* $\Gamma \vdash \mathtt{new\ C}_1(\overline{\mathtt{v}}_1) :: \ldots :: \mathtt{new\ C}_n(\overline{\mathtt{v}}_n) :: \epsilon : \mathtt{T}$ *where* $\mathit{concrete}(\mathtt{T})$, *let* $\mathit{sign}(\mathtt{C}_i) = \langle \mathtt{S}^i_1, \mathtt{S}^i_2 \rangle$, $1 \leq i \leq n$, *then, for any method* m*:*

1. $\mathit{mtype}(\mathtt{m}, \mathtt{T}) = \overline{\mathtt{B}} \to \mathtt{B}$ *implies that there exists some* $\mathtt{C}_i$, $1 \leq i \leq n$, *such that* $\mathtt{m} : \overline{\mathtt{B}} \to \mathtt{B} \in \mathtt{S}^i_1$.
2. *For any* $\mathtt{C}_i$, $1 \leq i \leq n$, $\mathtt{m} : \overline{\mathtt{B}} \to \mathtt{B} \in \mathtt{S}^i_1$ *implies* $\mathit{mbody}(\mathtt{m}, \mathtt{C}_i) = (\overline{\mathtt{x}}, \mathtt{e})$.
3. *For any* $\mathtt{C}_i$, $1 \leq i \leq n$, $\mathtt{m} : \overline{\mathtt{B}} \to \mathtt{B} \in \mathtt{S}^i_2$ *and* $\mathtt{m} \in \mathit{redef}(\mathtt{C}_i)$ *imply* $\mathit{mbody}(\mathtt{m}, \mathtt{C}_i) = (\overline{\mathtt{x}}, \mathtt{e})$.

Proof. 1. By induction on n:

- If $n = 1$, since $\mathit{concrete}(\mathtt{T})$, then $\mathtt{S}^1_2 = \emptyset$ and $\mathtt{m} : \overline{\mathtt{B}} \to \mathtt{B} \in \mathtt{S}^1_1$.
- In the inductive step, if $\mathtt{m} : \overline{\mathtt{B}} \to \mathtt{B} \in \mathtt{S}^1_2$, then, by Property 6.3, $\Gamma \vdash \mathtt{new\ C}_2(\overline{\mathtt{v}}_2) :: \ldots :: \mathtt{new\ C}_n(\overline{\mathtt{v}}_n) :: \epsilon : \mathtt{T}'$, where $\mathit{concrete}(\mathtt{T}')$; therefore the proof follows by the induction hypothesis.

2. If $\mathtt{m} : \overline{\mathtt{B}} \to \mathtt{B} \in \mathtt{S}^i_1$, then, by definition, $\mathtt{m} : \overline{\mathtt{B}} \to \mathtt{B}$ is defined in $\mathtt{C}_i$ or in inherited from a superclass; then, the result follows from the definition of *mbody*.
3. As in the previous case. $\square$

Concerning the progress property, we will prove that the values that can result from applying an evaluation rule to a well-typed expression are of the shape $\langle \mathtt{l}, \mathtt{l} \rangle$, where $\mathtt{l} \neq \epsilon$, which we call *final values*, while pairs of different lists are used only as receivers of method invocations. A final value $\langle \mathtt{l}, \mathtt{l} \rangle$ denotes the fully evaluated object which is represented by the list $\mathtt{l}$.

We first recall that pairs of lists do not occur in the source code of our language, namely an evaluation rule always applies to a well-typed (annotated) source code expression.

Property 6.12 (*Source Code Reduction*)**.** *Suppose* $\mathtt{e}$ *is a closed, well-typed expression, which is the annotated version of a source code expression. Then* $\mathtt{e} \longrightarrow \mathtt{e}'$ *for some* $\mathtt{e}'$.

Proof. By induction on typing rules of Fig. 5, using (TA-Invk) in place of (T-Invk). □

Lemma 6.13 (*Method Body Reduction*)**.** *Suppose* $\mathtt{e}$ *is a well-typed expression, which is the annotated version of a source code expression. If* $\overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : [\mathtt{C}], \mathtt{next} : \mathtt{S} \vdash \mathtt{e} : \mathtt{T}$, *and*

1. $\vdash \overline{\mathtt{v}} : \overline{\mathtt{T}}'$ *where* $\overline{\mathtt{T}}' <: \overline{\mathtt{C}}$,
2. $\vdash \mathtt{v} : \mathtt{T}_1$ *where* $\mathtt{T}_1 <: [\mathtt{C}]$,
3. $\vdash \mathtt{v}' : \mathtt{T}^*$ *where* $sign(\mathtt{T}^*) = \langle \mathtt{S}^*, \emptyset \rangle$ *and* $\mathtt{S} \subseteq \mathtt{S}^*$,
4. $\overline{\mathtt{v}}$ *are final values,*

then $[\overline{\mathtt{x}} \leftarrow \overline{\mathtt{v}}, \mathtt{this} \Leftarrow \mathtt{v}, \mathtt{next} \leftarrow \mathtt{v}']\mathtt{e}$ *either is a final value or can be reduced.*

Proof. Let us denote $[\overline{\mathtt{x}} \leftarrow \overline{\mathtt{v}}, \mathtt{this} \Leftarrow \mathtt{v}, \mathtt{next} \leftarrow \mathtt{v}']$ with σ and $\sigma\,\mathtt{e}$ with $\mathtt{e}^*$. The proof proceeds by induction on the derivation $\overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : [\mathtt{C}], \mathtt{next} : \mathtt{S} \vdash \mathtt{e} : \mathtt{T}$, using the property that $\mathtt{e}^*$ is well typed (by the Substitution Lemma 6.7).

(T-Var) Either $\mathtt{e} = \mathtt{this}$, then $\mathtt{e}^*$ is a final value by Definition 5.3-1, or $\mathtt{e} = \mathtt{x}_i$, then $\mathtt{e}^* = \mathtt{v}_i$ which is final by hypothesis.
(T-Field) $\mathtt{e}^* = (\sigma\,\mathtt{e}_1).\mathtt{f}$ for some $\mathtt{e}_1$ and $\mathtt{f}$; by the induction hypothesis, either $\sigma\,\mathtt{e}_1$ reduces to $\mathtt{e}_2$, for some $\mathtt{e}_2$, or $\sigma\,\mathtt{e}_1$ is a final value; in the first case, $\mathtt{e}^*$ reduces to $\mathtt{e}_2.\mathtt{f}$ using the corresponding congruence rule; in the second case, since $(\sigma\,\mathtt{e}_1).\mathtt{f}$ is well typed, then field selection can be performed (this can be easily proved as in FJ [25]).
(TA-Invk) We consider the only interesting case when $\mathtt{e}^* = \langle \mathtt{l}, \mathtt{l}' \rangle.\mathtt{m}(\overline{\mathtt{u}})$, for some well-typed run-time object $\langle \mathtt{l}, \mathtt{l}' \rangle$, method $\mathtt{m}$, and final values $\overline{\mathtt{u}}$ (the remaining cases follow straightforwardly from the induction hypothesis, using congruence rules); since $\mathtt{e}^*$ is well typed, then, by Lemma 6.11, it can be reduced.
(T-New) $\mathtt{e}^* = \mathtt{new}\ \mathtt{C}(\sigma\,\overline{\mathtt{e}})$ for some $\overline{\mathtt{e}}$. If $\sigma\,\overline{\mathtt{e}}$ are all final values, then we can apply rule (R-New). Otherwise, by induction hypothesis, one of the subexpression reduces, thus we can apply a congruence rule.
(T-Comp) $\mathtt{e}^* = \sigma\,\mathtt{e}_1 \leftarrow\!\!+\, \sigma\,\mathtt{e}_2$ for some well-typed $\mathtt{e}_1, \mathtt{e}_2$; by the induction hypothesis, if $\sigma\,\mathtt{e}_1$ and $\sigma\,\mathtt{e}_2$ are final values then $\mathtt{e}^*$ can be reduced using (R-Comp), otherwise a congruence rule can be applied. □

Theorem 6.14 (*Progress*)**.** *Let* $\mathtt{e}$ *be a closed run-time expression. If* $\vdash \mathtt{e} : \mathtt{T}$, *for some* $\mathtt{T}$, *and* $\mathtt{e} \longrightarrow \mathtt{e}'$ *for some* $\mathtt{e}'$, *then either* $\mathtt{e}'$ *is a final value, or* $\mathtt{e}'$ *can be reduced.*

Proof. By induction on the derivation $\mathtt{e} \longrightarrow \mathtt{e}'$, with a case analysis on the final rule, taking into account that $\mathtt{e}'$ is well typed by Theorem 6.10. We only consider reduction rules: if $\mathtt{e} \longrightarrow \mathtt{e}'$ is obtained by using a congruence rule, then the thesis follows from the induction hypothesis.

(R-New), (R-Comp) Immediate: $\mathtt{e}'$ is a final value.
(R-Field) $\overline{\mathtt{v}}$ are obtained by reducing well-typed expressions, then $\overline{\mathtt{v}}$ are final values by the induction hypothesis.
(R-Invk), (R-RInvk) As in the previous case, all the values in $\mathtt{e}$ are final values, thus the thesis follows by Lemma 6.13.
(R-DInvk) Since $\mathtt{e}'$ is well typed, then $\mathtt{l} \neq \epsilon$, thus we can apply either (R-Invk) or (R-RInvk). □

Theorem 6.15 (*Type Soundness*)**.** *Let* $\mathtt{e}$ *be a closed annotated source code expression, such that* $\vdash \mathtt{e} : \mathtt{T}$. *If* $\mathtt{e} \longrightarrow^\star \mathtt{e}'$, *and* $\mathtt{e}'$ *is such that no evaluation rule applies, then* $\mathtt{e}'$ *is a final value* $\mathtt{v}$, *such that* $\vdash \mathtt{v} : \mathtt{T}' <: \mathtt{T}$.

Proof. By Lemma 6.13, Theorem 6.10 and Theorem 6.14. □

7. Adding consultation

In this section we show how the language IFJ can be extended in order to provide also "consultation", as an additional semantic feature that coexists with delegation. This extension is smooth from the technical point of view, thanks to our representation of incomplete and run-time objects. Following the aim of enhancing language flexibility, the choice of whether to use consultation or delegation is left to the invoker, not to the writer of the method. Furthermore, the extension is conservative (a program using only delegation is still a valid program in this extended language).

In our opinion, both delegation and consultation should be available in a language with object composition, where the behavior can be changed at run time with redefining methods. However, we preferred not to present these two mechanisms in the same calculus from the beginning to keep the presentation of the metatheory simple, following the style of theoretical language calculi (see, e.g., Featherweight Java itself [25] and [38,1,14,15]), thus starting from a core framework and build additional features on top of it.

We briefly discuss some motivations of the proposed extension before presenting its technical and formal treatment. In particular, at the end of this section we show how to extend the properties of Section 6. Namely, the type safety is preserved in the extended language.

$$\frac{\mathtt{e} \longrightarrow \mathtt{e}'}{\mathtt{consult(e.m(\overline{e})^{\overline{B}\to B})} \longrightarrow \mathtt{consult(e'.m(\overline{e})^{\overline{B}\to B})}} \qquad \frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i'}{\mathtt{consult(v_0.m(\overline{v}, e_i, \overline{e})^{\overline{B}\to B})} \longrightarrow \mathtt{consult(v_0.m(\overline{v}, e_i', \overline{e})^{\overline{B}\to B})}}$$

$$\mathtt{consult(\langle new\ C(\overline{v}) :: l, l'\rangle.m(\overline{u})^{\overline{B}\to B})} \longrightarrow \mathtt{\langle new\ C(\overline{v}) :: l, \epsilon\rangle.m(\overline{u})^{\overline{B}\to B}} \quad \text{(R-InvkC)}$$

Fig. 9. Additional semantics rules.

Motivations. As we discussed in previous sections, delegation with the redefining method mechanism plays the same role as dynamic binding: it ensures that, at run time, the most recent version of a method is invoked. However, there might be cases when the programmer of a class wants to be sure that a method version is not overridden, or at least, that the binding at run time for that method is static. This possibility is present in mainstream programming languages such as Java and C++. In C++, if a method is not declared as **virtual**, then a static binding mechanism is used for method invocation. This means that if a class *B* redefines a method *m* of a class *A*, then, when using a variable of type *B*, the invocation of method *m* results in selecting the redefined version; instead, when using a variable of type *A*, even when it refers to an object of class *B* at run time, the definition of *m* in *A* will be selected. Java provides a similar mechanism, but, by default, all methods can be redefined, unless they are declared as **final** (and in this case, overriding of these methods in a derived class is completely ruled out).

The ability of requiring a static binding strategy gives the programmer more control on the possible evolution of the software, since it can be used to forbid derived classes to change the behavior of early-bound method invocations. If this is important in standard static class-based inheritance, it is even more crucial in our context, where the behavior of method invocation can be modified at run time via object composition. Moreover, the delegation mechanism can bring to unexpected loops in a program, as in the last case considered in Section 5.2.[9] Such cases cannot be caught by the type system, since they do not represent a type error, but only some ill-designed code. In fact, there is no name clash, but a (possible not wanted) override of a method which is in the chain of the composed object. In our setting, a static binding strategy can be provided by the *consultation* mechanism.

The extended language. First of all, we extend the syntax of IFJ with the following expression:

$$\mathtt{consult(e.m(\overline{e}))}$$

No change to the lookup functions and to the substitution of `this` (Definition 5.3) is required and we still use our representation of run-time objects.

Since consultation influences only the semantics of method invocation, the typing rule for the new construct simply relies on the typing rule (T-Invk) (and thus it is straightforward to extend the annotation function as well):

$$\frac{\Gamma \vdash \mathtt{e.m(\overline{e})} : \mathtt{B}}{\Gamma \vdash \mathtt{consult(e.m(\overline{e}))} : \mathtt{B}} \quad \text{(T-InvkC)}$$

Instead, we have to add two congruence rules and one reduction rule to the operational semantics; these rules are defined in Fig. 9. Our form of run-time objects as lists makes it straightforward to implement consultation. In fact, the new reduction rule simply removes the `consult` keyword and reduces the expression to a standard method invocation where the second list of the run-time object is empty; from this point on, the reduction procedure will use the rules previously defined in Section 5. This results in implementing a method call using the consultation mechanism, because the second list represents the entire object composition, which is used only to implement delegation. If that list is empty, the reduction procedure performs a consultation: in fact, when the method is found in the current node of the list, rule (R-Invk) substitutes `this` using point 3 of Definition 5.3. Then,

- the substitution calls the *findredef* function, passing the second list which is empty,
- then *findredef* returns $\emptyset$ (Definition 5.2-1),
- and, thus, we substitute $\langle \mathtt{new\ C(\overline{v})} :: \mathtt{l}, \epsilon\rangle$ for `this`.

Therefore, when we forward a method invocation, we will never be able to "go back" to the original sender, which corresponds exactly to the consultation semantics. This subsumes the consultation mechanism of [10] without changing Definition 5.3.

Note that `consult` acts at a method invocation level, not at a method declaration level (like, instead, **virtual** and **final**); this is consistent with the object composition run-time sought flexibility. Namely, it is the user of a method who decides whether a method should be invoked with delegation or consultation. For instance, consider the classes of Fig. 7 in Section 5.2; if we perform `consult(inc4 <+ (inc1 <+ c1).m())`, then we have the following method selections: `Inc4.m`, `Inc1.m`, `C1.n`, `C1.m` (while with delegation the program looped).

Finally, it is easy to verify that the proposed extended language preserves type safety. To this aim, we need to add a typing rule for run-time expressions of the shape $\langle \mathtt{l}, \epsilon\rangle$:

$$\frac{\Gamma \vdash \mathtt{l} : \mathtt{T}}{\Gamma \vdash \langle \mathtt{l}, \epsilon\rangle : \mathtt{T}} \quad \text{(T-ORunTimeC)}$$

[9] Note that such situations can be experienced even with standard class inheritance, method overriding and dynamic binding.

Property 6.5 is only used in the Substitution Lemma for the case of delegation, and it is not needed in the case of consultation. Therefore, it can be extended by adding the assumption $\mathtt{l}' \neq \epsilon$, for lists of the shape $\langle \mathtt{l}, \mathtt{l}' \rangle$, in order to rule out the consultation evaluation case.

We restate this property as in the following, while its proof stays the same, taking into account that if a list can be typed then it cannot be empty.

Property 7.1 (*Well-typedness of List Inclusion*). 1. *If* $\Gamma \vdash \mathtt{l}' : \mathtt{T}'$, *concrete*$(\mathtt{T}')$, *then* $\Gamma \vdash \langle \mathtt{l}, \mathtt{l}' \rangle : \mathtt{T}$, *for some* $\mathtt{T}$ *where concrete*$(\mathtt{T})$, *for any* $\mathtt{l} \subseteq \mathtt{l}'$.
2. *If* $\Gamma \vdash \langle \mathtt{l}, \mathtt{l}' \rangle : \mathtt{T}$ *where concrete*$(\mathtt{T})$, *and* $\mathtt{l}' \neq \epsilon$, *then* $\Gamma \vdash \mathtt{l}' : \mathtt{T}'$ *for some* $\mathtt{T}'$ *where concrete*$(\mathtt{T}')$.

All the other properties of Section 6, including the final type soundness property (Theorem 6.15), still hold, based on the fact that final values must be always of the shape $\langle \mathtt{l}, \mathtt{l} \rangle$ with $\mathtt{l} \neq \epsilon$. More precisely, by Definition 5.3-1, when we eventually replace `this` with a list $\langle \mathtt{l}, \mathtt{l}' \rangle$, we discard the second list and substitute `this` by $\langle \mathtt{l}, \mathtt{l} \rangle$; thus, the case when $\mathtt{l}'$ is empty does not make any difference.

8. Related work

Concerning the theory of incomplete objects, our main inspiration comes from [7]; however, while that calculus builds on top of the lambda calculus, here we aim at investigating on how object composition can fit within the basic principles of Java-like languages. Comparisons with other related work follow.

Incomplete objects in lambda calculus. An explicit form of incomplete objects was introduced in [12], where an extension of Lambda Calculus of Objects of [20] is presented. In this work, *labeled* types are used to collect information on the mutual dependencies among methods, enabling a safe subtyping in width. Labels are also used to implement the notion of *completion*, which permits adding methods in an arbitrary order allowing the typing of methods that refer to methods not yet present in the object, thus supporting a form of incomplete objects. The context is again a lambda calculus, while in this work we are interested in incorporating object composition into Java-like languages.

Other work on delegation. In [26], delegation is presented in the model of the language *Darwin*; however, this model requires some basic notions to be modified, such as method overriding. Our language, instead, proposes a conservative extension of a Java-like language (so that existing code needs not to be changed). Furthermore, in [26] the type of the *parent* object must be a declared class and this limits the flexibility of dynamic composition, while in our approach there is no implicit parent and missing methods can be provided by any complete object, independently from its class.

On mixins. Incomplete object mechanisms were originally inspired by *mixin*-based inheritance [13]: mixins are classes parametrized over the superclass and new subclasses can be generated by applying a mixin to a class (that provides all the requirements of the mixin). However, object composition in our language takes place at run time, while mixin inheritance, although more flexible than standard class-based inheritance, is still a compile-time mechanism.

On wrappers and delegates. Incomplete objects can be seen as *wrappers* for the objects used in object composition. However, they differ from decorator-based solutions such as the language extension presented in [11]: incomplete objects provide a more general-purpose language construct and the wrappers of [11] could be actually implemented through incomplete objects. Another form of wrapping of methods is the one offered by the *delegates* of C#. Delegates are objects pointing to one method or to a set of methods, that will be executed when invoked appropriately on the delegate. Therefore, it is possible to treat methods as anonymous functions, implementing a form of reuse. Delegates can be then seen as complementary to incomplete objects, as the latter implements a different form of reuse, allowing to customize a prototype (i.e., an incomplete object) in more than one way via object composition.

On partial classes. A further construct of C# that deals with some form of incompleteness is the one of *partial classes*, that makes it possible to subdivide a class definition among two or more files. This mechanism is useful to implement a form of reuse oriented to the design of large scale projects, as a class distributed over distinct files allows more programmers to work on it, moreover it helps the addition of new code to a class without modifying the source. In addition, the mechanism of partial classes is a static one, and it does not take place at run time like our object composition. Once again, the form of reuse offered by partial classes is complementary to the one implemented by incomplete objects.

On categories. Objective-C [27] provides *categories*, a run-time mechanism for modifying existing code: the programmer can place groups of related methods into a category and can add the methods within a category to a class at run time. Thus, categories permit the programmer to add methods to an existing class without the need to recompile that class or even have access to its source code. The main difference with our incomplete object mechanism is that categories act at the class level, while our linguistic feature acts at the object level.

On traits. *Traits* [18] are composable units containing only methods, and they were proposed as an add-on to traditional class-based inheritance in order to enhance decoupling and high cohesion of code in classes, therefore with the aim of allowing a higher degree of code reuse. Incomplete objects can be seen as a tool for rapid prototyping, that is, for adding methods on the fly to already existing objects. Traits and incomplete objects share an important feature, composition, which permits composing sets of methods "at the right level", for instance not too high in a hierarchy for traits, and "when needed"

for incomplete objects. The main difference is that traits are a compile-time feature, while incomplete objects are composed at run time. An issue to pursue as a further research may be the use of incomplete objects as an exploratory tool to design traits: experiments made at run time without modifying a class hierarchy might give indications on where to put a method in a new version of the hierarchy.

On aspects. There are some relations between aspects [17] and our incomplete objects. Both are used to combine features taken from different sources. In the aspect case, the main idea is to factorize into aspects some cross-cutting functions (such as logging services or concurrency primitives) that are needed globally by a library, instead of duplicating and scattering them into the business code. In our case, we consider objects as building blocks that can be used to combine features on the fly, in order to obtain and experiment with multi-function objects whenever it is desired. In a sense, the role of incomplete objects is orthogonal to the one of aspects, because the former play a local role, while the latter a more global one.

In [3], a general model (Method Driven Model) for languages supporting object composition is proposed: this is based on the design of classes in an aspect-oriented style. The authors do not formalize their model within a calculus, but it is possible to see that the main feature of a language based on this model would be to compose dynamically the overall behavior of an object from the multiple "aspects" that abstract the variant behavior, as discussed in [4]. The main difference between their proposal and ours is that for them the run-time behavior is codified in aspects, whereas we internalize it in Java by exploiting partial classes and object composition.

On gbeta. The language *gbeta* [19] supports a mechanism called "object metamorphosis", which is a mechanism to specialize dynamically an existing object, by applying to it a class as a constraint in such a way the object becomes an instance of that class. The main difference between the gbeta specializing objects and our incomplete objects is that the former maintain the object identity, while the latter are used to create dynamically new objects which are not instances of any classes present in the program. Both proposals are proved type safe, but a more direct comparison is not straightforward, as the type system of gbeta exploits concepts such as virtual classes which are not present in a Java-like setting like ours. The language gbeta also supports dynamic class composition [30] (classes are first class values and existing classes may be composed dynamically to yield new classes), while in our language we act on object composition. It is important to remark that one of our main design decision was that our extension must integrate seamlessly in a Java-like language as a conservative extension.

On roles. *Roles* [28] are a conceptual abstraction that can be used in object-oriented systems to implement specific entities within a domain. *ObjectTeams/Java* (OT/J) [24] is a programming language that provide roles in a Java context, following the criteria defined in [34]. Although the inherent compositional nature of roles looks similar to incomplete objects, the two approaches are rather different both for features and for intention. First of all, incomplete objects are a low level linguistic feature: they represent a dynamic and object-based implementation of inheritance. Thus, once objects are composed, they cannot be de-composed (although we might consider studying such an operation and how this affects the static type system). On the contrary, roles can be attached to base objects and detached; in particular, upon removal, a role is also destroyed. This is another important difference with respect to our object composition: objects in our language keep their own identity and life cycle (and they can be used in many object composition), while roles can be attached to one base object only and they "live" only when they are part of such a base object. Role definitions also specify the class of their base objects, and this couples them to these classes (this coupling can be reduced, but not removed, by using *unbound* roles, similar to abstract classes, and subroles), while in our approach the type of objects in composition is not known in advance. Moreover, OT/J implements a "flattening" semantics concerning fields, while in our composed objects all the sub-objects have their own fields. Summarizing, roles are a higher level feature which is somehow complementary to incomplete objects, and these two linguistic constructs aim at solving different programming contexts. These differences seem inherent of the role approach and can be found in other implementations (see, e.g., [5]).

9. Conclusions

Our goal was to design a language based on a trade-off between the dynamic flexibility of object-based languages and the static type discipline of class-based languages. To this aim, we presented linguistic constructs to deal with incomplete objects and object composition in a type-safe way. In our proposal, objects are still instances of (possibly incomplete) classes and they are still disciplined by the nominal subtyping, but they are also prototypes that can be used, via the object composition, to create new objects at run time, while ensuring statically that the composition is type safe.

We introduced two different treatments of method body lookup, one delegation-based, the other consultation-based. The former was presented in [6] and the latter was introduced in [10]. The delegation mechanism unleashes the flexibility of incomplete objects (compared to the consultation-based proposal) by enabling a form of dynamic method redefinition. This poses some interesting technical challenges that we solved in a pragmatic way (with in mind the constraint of being easily implementable in a Java-like language). For instance, object composition and delegation introduce the "width subtyping versus method addition" problem that is well known in the object-based setting (see, e.g., [21]). We solve this issue by representing objects as lists of sub-objects in such a way that we can explicitly deal with the "scope" of a method invocation; we believe this solution is more implementation-oriented than the dictionaries of [32] and simpler than the one of [7].

We pointed out that delegation is more powerful than consultation because it introduces a dynamic form of method redefinition. In practice, in our calculus, the leftmost object in a composition has always the power to override the methods

of the other objects whenever the method lookup is performed by delegation. However, as discussed in Section 7, there are some situations in which redefinition is not desirable. On the other hand, no redefinition at all, as it is with consultation, may be too restrictive in some other context.

This highlights the benefits of providing both delegation and consultation by object composition in Java-like languages. In the present paper, the interaction between the two mechanisms has been interpreted in a basic way: the programmer can decide to switch from the delegation mechanism to the consultation one by using a specific keyword (the operator `consult`). As an ongoing research, we are studying intermediate and more flexible forms of method lookup, in such a way that it is possible to enforce whether a certain method (or a set of methods), in a given chain of invoked methods, is (are) overridden dynamically or not. Our formalization of objects via pairs of lists allows us to introduce new operators, for such intermediate forms of combination of consultation and delegation, in such a way that the operators are independent from each other and the semantics of the language remains compositional (as it is with the addition of consultation in Section 7).

In [8] we presented I-Java, an extension of the Java language with incomplete objects and object composition. We implemented a preprocessor that, given a program that uses our language extension, produces standard Java code (the preprocessor is available at `http://i-java.sf.net`). The implementation in Java of incomplete objects with delegation is currently under development. The I-Java implementation via a preprocessor suggests a basis for a formal embedding of our calculus into Featherweight Java [25], and this is matter of an ongoing work.

In this introductory version of our calculus, we only allow an incomplete object to be completed in one shot, by composition with a completed object. To add practicality to our approach, it would be useful for incomplete objects to be also partially completed by a single composition, such that a new incomplete object is created, to be completed by one or further compositions. An additional mechanism would be to compose two incomplete objects. On the one hand, these two extensions would not pose any particular issue from an operational point of view, on the other hand they require a careful study of a suitable form of subtyping from the point of view of the type system. In fact, combining full object composition with subtyping introduces issues similar to ones discussed in [9,7], and they are the subject of future work.

Acknowledgements

We thank the referees for their helpful comments. The suggestions helped greatly to improve the final version of the paper.

References

[1] M. Abadi, L. Cardelli, A Theory of Objects, Springer, 1996.
[2] C. Anderson, F. Barbanera, M. Dezani-Ciancaglini, S. Drossopoulou, Can addresses be types? (A case study: objects with delegation), in: WOOD'03, in: ENTCS, vol. 82(8), Elsevier, 2003, pp. 1–22.
[3] C. Babu, D. Janakiram, Method driven model: a unified model for an object composition language, ACM SIGPLAN Notices 39 (8) (2004) 61–71.
[4] C. Babu, W. Jaques, D. Janakiram, DynOCoLa: enabling dynamic composition of object behaviour, in: Proc. of Workshop on Reflection, AOP and Meta-Data for Software Evolution, RAM-SE, 2005.
[5] M. Baldoni, G. Boella, L.W.N. van der Torre, Interaction between objects in PowerJava, Journal of Object Technology 6 (2) (2007).
[6] L. Bettini, V. Bono, Type safe dynamic object delegation in class-based languages, in: Proc. of PPPJ, ACM Press, 2008, pp. 171–180.
[7] L. Bettini, V. Bono, S. Likavec, Safe and flexible objects with subtyping, Journal of Object Technology 10 (4) (2005) 5–29.
[8] L. Bettini, V. Bono, E. Turin, I-Java: an extension of Java with incomplete objects and object composition, in: A. Bergel, J. Fabry (Eds.), Proc. of Software Composition, in: LNCS, vol. 5634, Springer, 2009, pp. 27–44.
[9] L. Bettini, V. Bono, B. Venneri, Subtyping-inheritance conflicts: the mobile mixin case, in: Proc. Third IFIP International Conference on Theoretical Computer Science, TCS 2004, Kluwer Academic Publishers, 2004.
[10] L. Bettini, V. Bono, B. Venneri, Object incompleteness and dynamic composition in Java-like languages, in: TOOLS 2008, in: LNBIP, vol. 11, Springer, 2008, pp. 198–217.
[11] L. Bettini, S. Capecchi, E. Giachino, Featherweight Wrap Java: wrapping objects and methods, Journal of Object Technology 7 (2) (2008) 5–29.
[12] V. Bono, M. Bugliesi, M. Dezani-Ciancaglini, L. Liquori, A subtyping for extensible, incomplete objects, Fundamenta Informaticae 38 (4) (1999) 325–364.
[13] G. Bracha, The programming language Jigsaw: mixins, modularity and multiple inheritance. Ph.D. Thesis, University of Utah, 1992.
[14] K. Bruce, Foundations of Object-Oriented Languages – Types and Semantics, The MIT Press, 2002.
[15] G. Castagna, Object-oriented programming: a unified foundation, in: Progress in Theoretical Computer Science, Birkhauser, 1997.
[16] C. Chambers, Object-oriented multi-methods in cecil, in: Proc. of ECOOP, in: LNCS, vol. 615, Springer, 1992, pp. 33–56.
[17] D. Crawford, Communications of the ACM Archive — Special Issue on Aspect-Oriented Programming, vol. 44, ACM, New York, 2001.
[18] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, A. Black, Traits: a mechanism for fine-grained reuse, ACM Transactions on Programming Languages and Systems 28 (2) (2006) 331–388.
[19] E. Ernst, gbeta — a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance. Ph.D. Thesis, Department of Computer Science, University of Århus, Denmark, 1999. Url: http://www.daimi.au.dk/~eernst/gbeta/.
[20] K. Fisher, F. Honsell, J.C. Mitchell, A lambda-calculus of objects and method specialization, Nordic Journal of Computing 1 (1) (1994) 3–37.
[21] K. Fisher, J.C. Mitchell, A delegation-based object calculus with subtyping, in: Proc. of FCT, in: LNCS, vol. 965, Springer, 1995, pp. 42–61.
[22] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
[23] A. Goldberg, D. Robson, Smalltalk 80: The Language, Addison-Wesley, 1989.
[24] S. Herrmann, A precise model for contextual roles: the programming language ObjectTeams/Java, Applied Ontology 2 (2) (2007) 181–207.
[25] A. Igarashi, B. Pierce, P. Wadler, Featherweight Java: a minimal core calculus for Java and GJ, ACM Transactions on Programming Languages and Systems 23 (3) (2001) 396–450.
[26] G. Kniesel, Type-safe delegation for run-time component adaptation, in: Proc. of ECOOP, in: LNCS, vol. 1628, Springer, 1999, pp. 351–366.
[27] S. Kochan, Programming in Objective-C 2.0, 2nd edition, Addison-Wesley, 2008.
[28] B.B. Kristensen, K. Østerbye, Roles: conceptual abstraction theory and practical language issues, Theory and Practice of Object Sytems 2 (3) (1996) 143–160.
[29] H. Lieberman, Using prototypical objects to implement shared behavior in object oriented systems, ACM SIGPLAN Notices 21 (11) (1986) 214–223.

[30] A.B. Nielsen, E. Ernst, Optimizing dynamic class composition in a statically typed language, in: R. Paige, B. Meyer (Eds.), Proc. of TOOLS, in: LNBIP, vol. 11, Springer, 2008, pp. 161–177.
[31] B.C. Pierce, Types and Programming Languages, The MIT Press, Cambridge, MA, 2002.
[32] J. Riecke, C. Stone, Privacy via subsumption, Information and Computation 172 (2002) 2–28.
[33] A.J. Riel, Object-Oriented Design Heuristics, Addison-Wesley, 1996.
[34] F. Steimann, On the representation of roles in object-oriented and conceptual modelling, Data Knowledge Engineering 35 (1) (2000) 83–106.
[35] A. Taivalsaari, On the notion of inheritance, ACM Computing Surveys 28 (3) (1996) 438–479.
[36] D. Ungar, R.B. Smith, Self: the power of simplicity, ACM SIGPLAN Notices 22 (12) (1987) 227–242.
[37] J. Viega, B. Tutt, R. Behrends, Automated delegation is a viable alternative to multiple inheritance in class based languages. Technical Report CS-98-03, UVa Computer Science, 1998.
[38] A. Wright, M. Felleisen, A syntactic approach to type soundness, Information and Computation 115 (1) (1994) 38–94.