An Overview of Feature Featherweight Java

Sven Apel[†], Christian Kästner[‡], and Christian Lengauer[†]

[†] Department of Informatics and Mathematics, University of Passau, Germany
{apel,lengauer}@uni-passau.de

[‡] School of Computer Science, University of Magdeburg, Germany
ckaestne@ovgu.de

Technical Report, Number MIP-0802
Department of Informatics and Mathematics
University of Passau, Germany
April 2008

An Overview of Feature Featherweight Java

Sven Apel†, Christian Kästner‡, and Christian Lengauer†

† Department of Informatics and Mathematics, University of Passau, Germany
{apel,lengauer}@uni-passau.de
‡ School of Computer Science, University of Magdeburg, Germany
ckaestne@ovgu.de

Abstract. *Feature-oriented programming (FOP)* is a paradigm that incorporates programming language technology, program generation techniques, and stepwise refinement. In their GPCE'07 paper, Thaker et al. suggest the development of a type system for FOP in order to guarantee safe feature composition. We present such a type system along with a calculus for a simple feature-oriented, Java-like language, called *Feature Featherweight Java (FFJ)*. Furthermore, we explore several extensions of FFJ and how they affect type soundness.

1 Introduction

Feature-oriented programming (FOP) aims at the modularization of software systems in terms of features. A *feature* implements a stakeholder's requirement and is typically an increment in program functionality [32,8]. Different variants of a software system are distinguished in terms of their individual features [21]. Contemporary feature-oriented programming languages and tools such as AHEAD [8], FSTComposer [7], and FeatureC++ [5] provide varying mechanisms that support the specification and composition of features properly. A key idea is that a feature, when added to a software system, introduces new structures, such as classes and methods, and refines existing ones, such as extending method bodies by overriding.

Stepwise refinement (SWR) is a related software development paradigm that aligns well with FOP [36]. In SWR, one adds detail to a program incrementally using refinements in order to satisfy a program specification. In terms of FOP, the individual refinements implement features.

In prior work, features have been modeled as functions and feature composition as function composition [8,25]. The function model tames feature composition in that it disallows features from affecting program structures that have been added by subsequent development steps, i.e., by features applied subsequently. This restriction is supposed to decrease the potential interactions between different program parts (i.e., features) and to avoid inadvertent interactions between present features and program elements that have been introduced in later development steps [25,27].

In their GPCE'07 paper, Thaker et al. raised the question of how the correctness of feature-oriented programs can be checked [34]. The problem is that feature-oriented languages and tools involve usually a code generation step in that they transform code into a lower-level representation. For example, the AHEAD Tool Suite transforms feature-oriented Jak code into object-oriented Java code by translating refinements of classes into subclasses [8]. Other languages and tools work similarly [5,4,7]. A problem of these languages and tools is that errors can be detected only at compilation time, not at composition time. While the compiler may detect errors caused by improper feature composition, it cannot recognize the actual cause of their occurrences. The reason is that information about features and their composition is lost during translation to the lower-level representation. For example, a feature may refer to a class that is not present because the feature the class belongs to is not present in a program variant, or a feature may affect a program element that is being introduced in a subsequent development step, which violates the principle of SWR.

Consequently, Thaker et al. suggested the development of a type system for feature-oriented languages and tools that can be used to check for the above errors at composition time. We present such a formal type system along with a soundness proof. To this end, we develop a calculus for a simple feature-oriented language on top of Featherweight Java (FJ) [18], called Feature Featherweight Java (FFJ). The syntax and semantics of FFJ conform with common feature-oriented languages. The type system not only incorporates language constructs for feature composition, but it also guarantees that the principle of SWR is not violated.

FFJ is interesting insofar as it is concerned partly with the programming language level (it provides language constructs for class, method, and constructor refinement on top of FJ) and partly with the composition engine at the meta-level (it relies on information about features that is collected outside the program text during composition), which is different from FJ. To our knowledge, FFJ is the first that aims at and incorporates both levels (base-level and the meta-level). Prior work on mixins, traits, and virtual classes concentrated on the language level, which is discussed in Section 6.

2 An Overview of FFJ

Before we go into detail, we give an informal overview of FFJ. FFJ builds on FJ. FJ is a language that models a minimal subset of Java. For example, FJ provides basic constructs like classes, fields, methods, and inheritance, but it does not support interfaces, exceptions, access modifiers, overloading, etc., even not assignment. In fact, FJ represents the functional core of Java and, therefore, every FJ program is also a regular Java program, but not vice versa. The designers of FJ concentrated on the module and type system of Java in order to simplify formal reasoning about Java-like languages and programs [18].

FFJ extends FJ by new language constructs for feature composition and by according evaluation and type rules. Although it is an extension of FJ, FFJ's key innovations can be used with other languages, e.g., C#.

As with other feature-oriented languages [8,5,7], the notion of a feature does not appear in the language syntax. That is, the programmer does not explicitly state in the program text that a class or method belongs to a feature. Merely, features are represented by containment hierarchies that are directories that aggregate the code artifacts that belong to a feature [8]. This is necessary since a feature may contain, beside code, also further supporting documents, e.g., documentation, test cases, design documents. By superimposing containment hierarchies, the code artifacts of different features are merged [8,7].

In FFJ, a programmer can add new classes to a program via the introduction of a new feature, which is trivial since only a new class file with a distinct name has to be supplied by the feature's containment hierarchy. Furthermore, using a feature, one can extend an existing class by a class refinement. A class refinement is declared like a class but preceded by the keyword refines. For example, refines class A refers to a class refinement that extends the class A. During composition, classes and refinements with the same name are merged, i.e., their members are merged. Finally, a class refinement may refine an existing method. This is handled similarly to overriding of a superclass' method by a subclass' method in Java.

The distinction between code artifacts (classes and class refinements) and features (containment hierarchies) requires a special treatment in FFJ's semantics and type system, which is different from previous approaches (see Section 6). Consequently, we use in the type system information about features that has been collected by the composition engine and that does not appear in the program text. For example, the name of a class refinement is made up of the name of the class that is refined and the feature the refinement belongs to. In order to check whether a feature's code does not refer to code from features added subsequently, FFJ uses information about the features' composition order.

We begin with a description of basic FFJ, that provides language constructs for feature composition, and proceed with four extensions for SWR. The extensions are largely orthogonal and can be combined individually and in any order with basic FFJ.

2.1 Basic FFJ

Figure 1 depicts a simple FFJ program that implements an expression evaluator, which is a solution to the infamous "expression problem" [35].[1] It consists of three features. The feature *Add* is the base feature that supports only addition of simple expressions; it introduces two classes Expr and Add (Lines 1–3, 4–7). The feature *Sub* adds support for substraction by introducing a class Sub (Lines 8–11). The feature *Eval* adds to each class a method eval for expression evaluation (Lines 12–15, 16–19, 20–23).

This simple example illustrates the main capabilities of FFJ. Like FJ, an FFJ program consists of a set of classes that, in turn, contain a single constructor

[1] Although not part of FJ and FFJ, we use basic data types, constants, and operators in our examples for sake of comprehensibility.

```
class Expr extends Object {        // Feature Add ...
  Expr() { super(); }
};
class Add extends Expr {
  int a; int b;
  Add(int a, int b) { super(); this.a=a; this.b=b; }
}
```

```
class Sub extends Expr {           // Feature Sub ...
  int a; int b;
  Sub(int a, int b) { super(); this.a=a; this.b=b; }
}
```

```
refines class Expr {               // Feature Eval ...
  refines Expr() { original(); }
  int eval() { return 0; }
}
refines class Add {
  refines Add(int a, int b) { original(a,b); }
  refines int eval() { return this.a+this.b; }
}
refines class Sub {
  refines Sub(int a, int b) { original(a,b); }
  refines int eval() { return this.a−this.b; }
}
```

Fig. 1. A solution to the "expression problem" in FFJ.

each, as well as methods and fields. For example, the class Add contains two fields and a constructor. Unlike FJ, an FFJ program may contain class refinements each of which contain a constructor refinement, a set of methods and fields that are added to the class that is refined, and a set of method refinements that refine the methods of the class that is refined (a.k.a. the base class). A base class along with its refinements has the semantics of a compound class that contains all fields and methods of its constituents, i.e., class refinements add new members. Constructor refinements extend the base constructor and method refinements replace base methods, i.e., class refinements change and extend existing members.

A feature may contain several classes and class refinements, but we impose some restrictions. First, a feature is not allowed to introduce a class that is already present in a program it is composed with. This includes also that a feature must not introduce two classes with the same name. The reason is that classes must be unambiguously identifiable. For example, the feature *Eval* is not allowed to add any further class with the name Expr. Second, a feature is not allowed two apply two refinements to the same class. The reason is that, otherwise, the order of applying class refinements cannot be determined precisely. For example, *Eval* cannot refine the class Add twice. Finally, a feature is not allowed to introduce a class together with a refinement of this class. This is to keep the changes a feature can make as simple as possible. For example, the feature *Sub* cannot introduce a class Sub along with a refinement of Sub.

Like in FJ, each class must declare exactly one superclass, which may be Object. In contrast, a class refinement does not declare (additional) superclasses. Later on, we will extend FFJ such that class refinements declare further superclasses to a base class.

Typically, with a sequence of features, a programmer can apply several refinements to a class, which is called a *refinement chain*. A refinement that is applied before another refinement in the chain is called its *predecessor*. Conversely, a class refinement that is applied after another refinement is called its *successor*. For example, when selecting the features *Add* and *Eval*, the refinement chain of Add consists of a base class and a refinement applied by *Eval*; further refinements may follow. The order of refinements in a refinement chain is determined by the selection of features and their composition order.

Figure 2 depicts the refinement and inheritance relationships of our expression example. Names of refinements are generated taking the name of the base class, followed by an '@', and the name of the feature that the refinement belongs to.

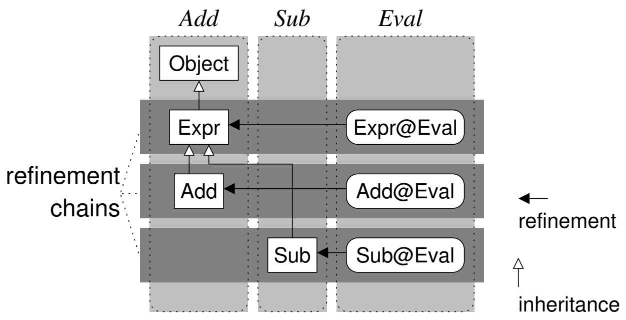

Fig. 2. Refinement and inheritance relationships of the expression example.

Fields are unique within the scope of a class and its inheritance hierarchy and refinement chain. That is, a refinement or subclass is not allowed to add a field that is already defined. For example, the feature *Eval* is not allowed to add a further field a or b to the class Add. With methods this is different. A property that FFJ has inherited from FJ is that subclasses may override methods of superclasses. Similarly to FJ, FFJ does not allow the programmer to use super inside a method body, as it would be possible in Java. That is, method overriding in FFJ means essentially method replacement.

Methods in FFJ (and FJ) are similar to Java methods except that a method body is an expression (preceded by return) and not a sequence of statements. This is due to the functional nature of FFJ (and FJ). Furthermore, overloading of methods (methods with equal names and different argument types) is not allowed in FJ and FFJ.

Unlike classes, class refinements are not allowed to define methods that have already been defined before in the refinement chain. That is, class refinements cannot override methods. This is to avoid inadvertent replacement. But, instead, a class refinement may declare a method refinement. A method refinement is like a method declaration but preceded with the keyword refines. This enables the type checker to recognize the difference between method refinement and inadvertent overriding/replacement and, possibly, to warn the programmer. For example, the feature *Eval* adds a method to the class Expr (Line 14). This way, also the subclasses of Expr, Add and Sub, have this method. Furthermore, *Eval* applies two method refinements (Lines $18, 22$) to the eval methods specifying that Add and Sub are being inherited from Expr. Note that refinements may also refine methods that have been introduced by any superclass or by other refinements that belong to previous development steps. The difference between method refinement and method overriding in subclasses becomes more useful in an extension of basic FFJ that allows a method refinement to reuse the body of a refined method (see Section 2.2).

As shown in Figure 2, refinement chains grow from left to right and inheritance hierarchies from top to bottom. When looking up a method body, FFJ traverses the combined inheritance and refinement hierarchy of the object the method belongs to and selects the right-most and bottom-most method body of a method declaration or method refinement that is compatible. That is, first, FFJ looks for a method declaration or method refinement in the refinement chain of the object's class, starting with the last refinement back to the class declaration itself. The first body of a matching method declaration or method refinement is returned. If the method is found neither in the class' refinement chain nor in its declaration, the methods in the superclass (and then the superclass' superclass, etc.) are searched, each again from latest refinement to the class declaration itself.

For example, looking up the method (new Add(3,4)).eval(), the inheritance hierarchy of Add is traversed and, eventually, eval defined in Expr is selected. Then, all refinements and superclasses of Add are traversed and the method refinement of eval defined in the class refinement of Add in the feature *Eval* is returned finally, which is bottom-most and right-most.

Finally, each class must declare exactly one constructor that is used solely to initialize the class' fields. Similarly, a class refinement must declare exactly one constructor refinement that initializes the class refinement's fields. A constructor expects values for all fields that have been declared by its class and the class' superclasses. The values for the superclass are passed via super. Similarly, a constructor refinement expects values for all fields that have been declared by previous refinements in the refinement chain and the base class. The values for the predecessor in the refinement chain are passed via original. For example, the constructor of the class Add expects two integers that are assigned to its fields a and b via this.a=a and this.b=b (Line 6), and it invokes super without arguments since the superclass Expr does not have any fields (Line 6); the constructor refinement of Add in the feature *Eval* expects two integers for its base class Add that are passed via original; it does not add any fields to Add so that the constuctor refinement still expects only two integers (Line 17). Note that this approach to constructor refinement fixes the order in a refinement chain. The reason is that each refinement has to "know" who its predecessor is, as it has to pass the predecessor's constructor arguments via original. We use this mechanism in basic FFJ for simplicity. In the next section, we discuss an extension that is more flexible in this regard.

2.2 Extensions for SWR

On top of basic FFJ, we introduce several extensions that model certain aspects of FOP and SWR.

Method Extension

A principle that has become best practice in FOP is that the replacement of existing methods is inelegant programming style [34]. In FFJ, method refinements may override methods and effectively replace them (same for method overriding in subclasses). This prevents programmers from extending existent functionality and leads to code replication in that programmers repeat code of extended methods that cannot be reused. In order to foster extension [34], we allow programmers to invoke the refined method from the method refinement, which is done using the keyword original. If original is not invoked, an error is reported.[2] This helps further to avoid inadvertent replacement, as has been demonstrated in several case studies [34].

For example, a feature might refine the eval methods of our expression evaluator in order to log their invocation. Using original, the method refinement of eval can extend the existing method instead of replacing it:[3]

```
refines class Add {                    // Feature Logging ...
  refines int eval() { return new Log().write(original()); }
} ...
```

Default Values

A class refinement in FFJ may add new fields to a class and the according constructor refinement extends the class' constructor initializing these fields. The problem of this simple mechanism is that a class cannot be used anymore by client classes that have been added before the class refinement in question. This is because the class refinement extends the constructor's signature and, for

[2] Alternatively, the error could be softened into a warning.

[3] The method write of Log expects an integer, logs the values, and returns the integer unchanged.

a client class that has been introduced before, knowing about these new fields in the first place is unlikely. Moreover, passing the values violates the principle of SWR, as a constructor refinement and fields are referenced that have been added subsequently (see also our 'backward references' extension).

Suppose a refinement of Expr that adds a new field and that refines the base class constructor accordingly:

```
refines class Expr {
  int id;
  refines Expr(int id) { original(); this.id=id; }
}
```

Applying this refinement breaks the constructor of Add; the constructor's super call receives an empty list of arguments, whereas the refined constructor of Expr expects an integer.

There are three options for solving this problem: (1) the constructor of Add can be modified expecting a value for id, (2) the constructor of Add can be refined expecting a value for id that is passed to Expr via super, or (3) the field id can be initialized with some sort of a default value. The first two options necessitate, for each refinement of a class, a modification or refinement of all its client classes (i.e., the classes that use the base class and its refinements in question). Moreover, with the first option, Add is aware of id that has been introduced subsequently, which violates the principle of SWR. Therefore, we choose the third option: providing default values for uninitialized fields.

When instantiating a class, a programmer does not need to pass values for all arguments of the class' constructor but only for some of them (i.e., for a subsequence of the argument list). The remaining arguments are filled with default values supplied by the type system. This is similar to C++, where fields that are allocated in the static data segment are initialized with their default constructors. An alternative would be to let the programmer specify the default values in the constructor declaration, much like in C++. We refrain from this alternative in our formalization as it complicates the calculus unnecessarily. As we will see later on, default values can be generated completely automatically, without asking the programmer to supply them.

Superclass Declaration

In basic FFJ, class refinements may add new field and method declarations and refine existing methods. A practice that has proved useful in SWR is that subsequent features may also alter the inheritance hierarchy [31]. Therefore, in an extension of FFJ, we let each class refinement declare a superclass, much like a class declaration. For example, we can refine the class Add in order to inherit also from Comparable:

```
class Comparable extends Object {
  Comparable() { super(); }
  boolean equals(Comparable c) { return true; }
}
refines class Add extends Comparable {
  refines Add(int a, int b) { super(); original(a, b); }
  refines boolean equals(Comparable c) {
    return ((Add)c).a == this.a && ((Add)c).b == this.b;
  }
}
```

Note that Add inherits now from both Expr and Comparable. In order to pass arguments properly, the constructor of Add's refinement uses super for passing arguments to the superclass and original for passing arguments to the base class.

Effectively, a class that is merged with its class refinements inherits from multiple classes, which is a kind of multiple inheritance. However, our intension is not meant to solve the tricky problems of multiple inheritance, e.g., the diamond problem [33], so we impose some restrictions. First, a class refinement is only allowed to declare a superclass that has not been declared before in the refinement chain, except for Object. Second, all further superclasses of this superclass must not be declared before. Third, the superclass (incl. all its superclasses) must not introduce a field or method that has been introduced before in the refinement chain; this is to avoid name clashes. The rules for method overriding and refinement are similar to basic FFJ. Finally, the method body look up mechanism changes. Now, looking up the refinement chain also requires to look in a refinement's superclasses for a method body.

Backward References

This extension allows the type system to check whether all classes, class refinements, methods, method refinements, and fields contain only references to features that have been added before, which we call *backward references*. In contrast, the type checker rejects programs containing *forward references*. This is in line with the principle of SWR disallowing code of previous development steps to affect code of subsequent development steps. Adhering to this principle decreases potential interactions in a software system and improves software comprehension [36,25].

For example, in Figure 1, the base class Add may contain a reference to the class Expr but must not contain a reference to the class Sub and its members because they are being introduced subsequently. We can enforce this property by checking superclass and field declarations, as well as bodies and signatures of methods and method refinements for the direction of their type or member references.

3 The Basic FFJ Calculus

FJ is a calculus that models a minimal subset of Java. FFJ extends FJ for FOP and SWR. Due to the lack of space we cannot give a formal description of FJ. Instead, we explain the relationship between FJ and FFJ when exploring the details of the FFJ calculus. For a better understanding of the changes and extensions that FFJ makes to FJ, in the colored version of the paper, we highlight modified rules with shaded yellow boxes and new rules with shaded purple boxes.

3.1 Syntax

$\mathsf{CD} ::=$ *class declarations:*
  $\mathsf{class\ C\ extends\ C\ \{\ \overline{C}\ \overline{f};\ KD\ \overline{MD}\ \}}$

$\mathsf{CR} ::=$ *class refinements:*
  $\mathsf{refines\ class\ C\ \{\ \overline{C}\ \overline{f};\ KR\ \overline{MD}\ \overline{MR}\ \}}$

$\mathsf{KD} ::=$ *constructor declarations:*
  $\mathsf{C(\overline{D}\ \overline{g},\ \overline{C}\ \overline{f})\ \{\ super(\overline{g});\ this.\overline{f}=\overline{f};\ \}}$

$\mathsf{KR} ::=$ *constructor refinements:*
  $\mathsf{refines\ C(\overline{E}\ \overline{h},\ \overline{C}\ \overline{f})\ \{\ original(\overline{h});\ this.\overline{f}=\overline{f};\ \}}$

$\mathsf{MD} ::=$ *method declarations:*
  $\mathsf{C\ m(\overline{C}\ \overline{x})\ \{\ return\ t;\ \}}$

$\mathsf{MR} ::=$ *method refinements:*
  $\mathsf{refines\ C\ m(\overline{C}\ \overline{x})\ \{\ return\ t;\ \}}$

$\mathsf{t} ::=$	*terms:*
x	*variable*
$\mathsf{t.f}$	*field access*
$\mathsf{t.m(\overline{t})}$	*method invocation*
$\mathsf{new\ C(\overline{t})}$	*object creation*
$\mathsf{(C)\,t}$	*cast*

$\mathsf{v} ::=$	*values:*
$\mathsf{new\ C(\overline{v})}$	*object creation*

Fig. 3. Syntax of basic FFJ.

In Figure 3, we depict the syntax of FFJ, which is a straightforward extension of the syntax of FJ [18]. An FFJ program consists of a set of class and refinement declarations as well as an expression. A class declaration CD contains a list $\overline{\mathsf{C}}\ \overline{\mathsf{f}}$ of fields,[4] a constructor declaration KD, and list $\overline{\mathsf{MD}}$ of method declarations. A class refinement CR contains a list $\overline{\mathsf{C}}\ \overline{\mathsf{f}}$ of fields, a constructor refinement KR, a list $\overline{\mathsf{MD}}$ of method declarations, and a list $\overline{\mathsf{MR}}$ of method refinements. The declaration of a class refinement is preceded by the keyword $\mathsf{refines}$. Method and constructor declarations are taken from FJ without change: A method m expects arguments $\overline{\mathsf{C}}\ \overline{\mathsf{x}}$, contains a body $\mathsf{return\ t}$, and returns a result of type C. A constructor expects two lists $\overline{\mathsf{D}}\ \overline{\mathsf{g}}$ and $\overline{\mathsf{C}}\ \overline{\mathsf{f}}$ of arguments for the fields of the superclass (passed via $\mathsf{super(\overline{g})}$) and for the fields of its own class (initialized via $\mathsf{this.\overline{f}=\overline{f}}$). A constructor refinement KR expects arguments for the predecessor refinement (via $\mathsf{original(\overline{h})}$) and for its own fields ($\mathsf{this.\overline{f}=\overline{f}}$). A method refinement is much like a method declaration; constructor and method refinements begin with the keyword $\mathsf{refines}$. The remaining syntax rules for terms t and values v are straightforward and taken from FJ without change.

Class names (metavariables A–E) are simple identifiers. A refinement name (metavariables R–T) consists of the name of the class C it refines and the name of

[4] We abbreviate lists in the obvious way: $\overline{\mathsf{C}}\ \overline{\mathsf{f}}$ is shorthand for $\mathsf{C}_1\ \mathsf{f}_1, \ldots, \mathsf{C}_n\ \mathsf{f}_n$; $\overline{\mathsf{C}}\ \overline{\mathsf{f}};$ is shorthand for $\mathsf{C}_1\ \mathsf{f}_1; \ldots; \mathsf{C}_n\ \mathsf{f}_n;$ and $\mathsf{this.\overline{f}=\overline{f};}$ is shorthand for $\mathsf{this.f_1=f_1; \ldots; this.f_n=f_n;}$.

the feature F it belongs to. Declarations of classes and refinements can be looked up via the class table CT. As in FJ, we impose some sanity conditions on the class table: (1) $CT(\mathsf{C}) = \mathsf{class\ C}\ldots$ for every $\mathsf{C} \in dom(CT)$; (2) $\mathsf{Object} \notin dom(CT)$; (3) for every class name C (except Object) appearing anywhere in CT, we have $\mathsf{C} \in dom(CT)$; and (4) there are no cycles (incl. self-cycles) in the inheritance relation. The conditions for class refinements are analogous.

The composite name of a refinement allows the class table to identify the according declaration. There is also a refinement table RT that maps refinement declarations to refinement names (e.g., $RT(\mathsf{refines\ class\ Expr}\ \ldots) = \mathsf{Expr@Eval}$). For every refinement name there is precisely one refinement declaration.

3.2 Subtyping and Refinement

Navigating the refinement chain

$$\frac{\mathsf{S\ is\ the\ successor\ of\ R}}{succ(\mathsf{R}) = \mathsf{S}} \qquad \frac{succ(\mathsf{S}) = \mathsf{R}}{pred(\mathsf{R}) = \mathsf{S}} \qquad \frac{\mathsf{S} \prec: \mathsf{R} \qquad succ(\mathsf{S}) = \mathsf{Object}}{last(\mathsf{R}) = \mathsf{S}}$$

Refinement relation $\boxed{\mathsf{S} \prec: \mathsf{R}}$

$$\frac{succ(\mathsf{R}) = \mathsf{S}}{\mathsf{S} \prec: \mathsf{R}} \qquad \frac{\mathsf{T} \prec: \mathsf{S} \qquad \mathsf{S} \prec: \mathsf{R}}{\mathsf{T} \prec: \mathsf{R}}$$

Subtyping $\boxed{\mathsf{C} <: \mathsf{D}}$

$$\mathsf{C} <: \mathsf{C} \qquad \frac{\mathsf{C} <: \mathsf{D} \qquad \mathsf{D} <: \mathsf{E}}{\mathsf{C} <: \mathsf{E}} \qquad \frac{CT(\mathsf{C}) = \mathsf{class\ C\ extends\ D}\ \{\ldots\}}{\mathsf{C} <: \mathsf{D}}$$

Fig. 4. Subtyping and refinement in basic FFJ.

In Figure 4, we show the refinement and subtyping relations of FFJ. There are three auxiliary functions that return the next refinement ($succ$), the previous refinement ($pred$), and the most specific refinement ($last$) in a refinement chain. These definitions rely on information collected by the composition engine, e.g., the features' composition order. For simplicity, the three functions may be used with classes and class refinements, and they return Object if there is no class refinement that matches.

Furthermore, there is a refinement relation $\prec:$ that states that a refinement is an immediate or distant successor of another refinement or class, i.e., it is further to the right in the refinement chain. Finally, there is a subtype relation $<:$ identical to the one of FJ. That is, in basic FFJ, class, method, and constructor refinement do not affect the subtype relation.

Field lookup $\boxed{\mathit{fields}(\mathsf{C}) \;=\; \overline{\mathsf{C}}\;\overline{\mathsf{f}}}$

$$\mathit{fields}(\mathsf{Object}) \;=\; \bullet \qquad \frac{\mathit{CT}(\mathsf{C}) = \mathsf{class\ C\ extends\ D\ \{\ \overline{C}\ \overline{f};\ KD\ \overline{MD}\ \}}}{\mathit{fields}(\mathsf{C}) \;=\; \mathit{fields}(\mathsf{D}),\ \overline{\mathsf{C}}\ \overline{\mathsf{f}},\ \mathit{fields}(\mathit{succ}(\mathsf{C}))}$$

$$\frac{\mathit{CT}(\mathsf{R}) \;=\; \mathsf{refines\ class\ C\ \{\ \overline{C}\ \overline{f};\ KR\ \overline{MD}\ \overline{MR}\ \}}}{\mathit{fields}(\mathsf{C}) \;=\; \overline{\mathsf{C}}\ \overline{\mathsf{f}},\ \mathit{fields}(\mathit{succ}(\mathsf{C}))}$$

Reverse field lookup $\boxed{\mathit{rfields}(\mathsf{R}) \;=\; \overline{\mathsf{C}}\;\overline{\mathsf{f}}}$

$$\mathit{rfields}(\mathsf{Object}) \;=\; \bullet \qquad \frac{\mathit{CT}(\mathsf{C}) \;=\; \mathsf{class\ C\ extends\ D\ \{\ \overline{C}\ \overline{f};\ KD\ \overline{MD}\ \}}}{\mathit{rfields}(\mathsf{C}) \;=\; \mathit{fields}(\mathsf{D}),\ \overline{\mathsf{C}}\ \overline{\mathsf{f}}}$$

$$\frac{\mathit{CT}(\mathsf{R}) = \mathsf{refines\ class\ C\ \{\ \overline{C}\ \overline{f};\ KR\ \overline{MD}\ \overline{MR}\ \}}}{\mathit{rfields}(\mathsf{R}) \;=\; \mathit{rfields}(\mathit{pred}(\mathsf{R})),\ \overline{\mathsf{C}}\ \overline{\mathsf{f}}}$$

Method type lookup $\boxed{\mathit{mtype}(\mathsf{m},\mathsf{C}) \;=\; \overline{\mathsf{C}}\to\mathsf{C}}$

$$\frac{\mathit{CT}(\mathsf{C}) \;=\; \mathsf{class\ C\ extends\ D\ \{\ \overline{C}\ \overline{f};\ KD\ \overline{MD}\ \}} \qquad \mathsf{B\ m(\overline{B}\ \overline{x})\ \{\ return\ t;\ \}} \in \overline{\mathsf{MD}}}{\mathit{mtype}(\mathsf{m},\mathsf{C}) \;=\; \overline{\mathsf{B}}\to\mathsf{B}}$$

$$\frac{\begin{array}{c}\mathit{CT}(\mathsf{C}) \;=\; \mathsf{class\ C\ extends\ D\ \{\ \overline{C}\ \overline{f};\ KD\ \overline{MD}\ \}} \\ \mathsf{m}\text{ is not defined in }\overline{\mathsf{MD}} \qquad \mathit{mtype}(\mathsf{m},\mathit{succ}(\mathsf{C}))\end{array}}{\mathit{mtype}(\mathsf{m},\mathsf{C}) \;=\; \mathit{mtype}(\mathsf{m},\mathit{succ}(\mathsf{C}))}$$

$$\frac{\begin{array}{c}\mathit{CT}(\mathsf{C}) \;=\; \mathsf{class\ C\ extends\ D\ \{\ \overline{C}\ \overline{f};\ KD\ \overline{MD}\ \}} \\ \mathsf{m}\text{ is not defined in }\overline{\mathsf{MD}} \qquad \neg\mathit{mtype}(\mathsf{m},\mathit{succ}(\mathsf{C}))\end{array}}{\mathit{mtype}(\mathsf{m},\mathsf{C}) \;=\; \mathit{mtype}(\mathsf{m},\mathsf{D})}$$

$$\frac{\mathit{CT}(\mathsf{R}) \;=\; \mathsf{refines\ class\ C\{\ \overline{C}\ \overline{f};\ KR\ \overline{MD}\ \overline{MR}\ \}} \qquad \mathsf{B\ m(\overline{B}\ \overline{x})\ \{\ return\ t;\ \}} \in \overline{\mathsf{MD}}}{\mathit{mtype}(\mathsf{m},\mathsf{R}) \;=\; \overline{\mathsf{B}}\to\mathsf{B}}$$

$$\frac{\mathit{CT}(\mathsf{R}) \;=\; \mathsf{refines\ class\ C\ \{\ \overline{C}\ \overline{f};\ KR\ \overline{MD}\ \overline{MR}\ \}} \qquad \mathsf{m}\text{ is not defined in }\overline{\mathsf{MD}}}{\mathit{mtype}(\mathsf{m},\mathsf{R}) \;=\; \mathit{mtype}(\mathsf{m},\mathit{succ}(\mathsf{R}))}$$

Fig. 5. Auxiliary definitions of basic FFJ.

Reverse method type lookup $\boxed{\mathit{rmtype}(\mathsf{m},\mathsf{C}) \;=\; \overline{\mathsf{C}}\rightarrow\mathsf{C}}$

$$\frac{\mathit{CT}(\mathsf{C}) \;=\; \mathsf{class\ C\ extends\ D}\ \{\ \overline{\mathsf{C}}\ \overline{\mathsf{f}};\ \mathsf{KD}\ \overline{\mathsf{MD}}\ \} \qquad \mathsf{B\ m}(\overline{\mathsf{B}}\ \overline{\mathsf{x}})\ \{\ \mathsf{return\ t;}\ \} \in \overline{\mathsf{MD}}}{\mathit{rmtype}(\mathsf{m},\mathsf{C}) \;=\; \overline{\mathsf{B}}\rightarrow\mathsf{B}}$$

$$\frac{\mathit{CT}(\mathsf{R}) \;=\; \mathsf{refines\ class\ C}\ \{\ \overline{\mathsf{C}}\ \overline{\mathsf{f}};\ \mathsf{KR}\ \overline{\mathsf{MD}}\ \overline{\mathsf{MR}}\ \} \qquad \mathsf{B\ m}(\overline{\mathsf{B}}\ \overline{\mathsf{x}})\ \{\ \mathsf{return\ t;}\ \} \in \overline{\mathsf{MD}}}{\mathit{rmtype}(\mathsf{m},\mathsf{R}) \;=\; \overline{\mathsf{B}}\rightarrow\mathsf{B}}$$

$$\frac{\mathit{CT}(\mathsf{R}) \;=\; \mathsf{refines\ class\ C}\ \{\ \overline{\mathsf{C}}\ \overline{\mathsf{f}};\ \mathsf{KR}\ \overline{\mathsf{MD}}\ \overline{\mathsf{MR}}\ \} \qquad \mathsf{m}\ \text{is not defined in}\ \overline{\mathsf{MD}}}{\mathit{rmtype}(\mathsf{m},\mathsf{R}) \;=\; \mathit{rmtype}(\mathsf{m},\mathit{pred}(\mathsf{R}))}$$

Method body lookup $\boxed{\mathit{mbody}(\mathsf{m},\mathsf{C}) \;=\; (\overline{\mathsf{x}},\mathsf{t})}$

$$\frac{\begin{array}{c}\mathit{CT}(\mathsf{C}) \;=\; \mathsf{class\ C\ extends\ D}\ \{\ \overline{\mathsf{C}}\ \overline{\mathsf{f}};\ \mathsf{KD}\ \overline{\mathsf{MD}}\ \} \\ \mathsf{B\ m}(\overline{\mathsf{B}}\ \overline{\mathsf{x}})\ \{\ \mathsf{return\ t;}\ \} \in \overline{\mathsf{MD}} \qquad \neg\mathit{mbody}(\mathsf{m},\mathit{succ}(\mathsf{C}))\end{array}}{\mathit{mbody}(\mathsf{m},\mathsf{C}) \;=\; (\overline{\mathsf{x}},\mathsf{t})}$$

$$\frac{\begin{array}{c}\mathit{CT}(\mathsf{C}) \;=\; \mathsf{class\ C\ extends\ D}\ \{\ \overline{\mathsf{C}}\ \overline{\mathsf{f}};\ \mathsf{KD}\ \overline{\mathsf{MD}}\ \} \\ \mathsf{m}\ \text{is not defined in}\ \overline{\mathsf{MD}} \qquad \neg\mathit{mbody}(\mathsf{m},\mathit{succ}(\mathsf{C}))\end{array}}{\mathit{mbody}(\mathsf{m},\mathsf{C}) \;=\; \mathit{mbody}(\mathsf{m},\mathsf{D})}$$

$$\mathit{mbody}(\mathsf{m},\ \mathsf{C}) = \mathit{mbody}(\mathsf{m},\ \mathit{succ}(\mathsf{C}))$$

$$\frac{\begin{array}{c}\mathit{CT}(\mathsf{R}) \;=\; \mathsf{refines\ class\ C}\ \{\ \overline{\mathsf{C}}\ \overline{\mathsf{f}};\ \mathsf{KR}\ \overline{\mathsf{MD}}\ \overline{\mathsf{MR}}\ \} \qquad \neg\mathit{mbody}(\mathsf{m},\mathit{succ}(\mathsf{R})) \\ \mathsf{B\ m}(\overline{\mathsf{B}}\ \overline{\mathsf{x}})\ \{\ \mathsf{return\ t;}\ \} \in \overline{\mathsf{MD}}\ \ \text{or}\ \ \mathsf{refines\ B\ m}(\overline{\mathsf{B}}\ \overline{\mathsf{x}})\ \{\ \mathsf{return\ t;}\ \} \in \overline{\mathsf{MR}}\end{array}}{\mathit{mbody}(\mathsf{m},\mathsf{R}) \;=\; (\overline{\mathsf{x}},\mathsf{t})}$$

$$\frac{\mathit{CT}(\mathsf{R}) \;=\; \mathsf{refines\ class\ C}\ \{\ \overline{\mathsf{C}}\ \overline{\mathsf{f}};\ \mathsf{KR}\ \overline{\mathsf{MD}}\ \overline{\mathsf{MR}}\ \}}{\mathit{mbody}(\mathsf{m},\mathsf{R}) \;=\; \mathit{mbody}(\mathsf{m},\mathit{succ}(\mathsf{R}))}$$

Valid method overriding $\boxed{\mathit{override}(\mathsf{m},\mathsf{D},\overline{\mathsf{C}}\rightarrow\mathsf{C}_0)}$

$$\frac{\mathit{mtype}(\mathsf{m},\mathsf{D}) \;=\; \overline{\mathsf{D}}\rightarrow\mathsf{D}_0\ \ \text{implies}\ \ \overline{\mathsf{C}} \;=\; \overline{\mathsf{D}}\ \ \text{and}\ \ \mathsf{C}_0 \;=\; \mathsf{D}_0}{\mathit{override}(\mathsf{m},\mathsf{D},\overline{\mathsf{C}}\rightarrow\mathsf{C}_0)}$$

Valid method introduction $\boxed{\mathit{introduce}(\mathsf{m},\mathsf{C})}$

$$\frac{\neg\mathit{mtype}(\mathsf{m},\mathit{succ}(\mathsf{C}))}{\mathit{introduce}(\mathsf{m},\mathsf{C})}$$

Valid method refinement $\boxed{\mathit{extend}(\mathsf{m},\mathsf{R},\overline{\mathsf{C}}\rightarrow\mathsf{C}_0)}$

$$\frac{\mathit{rmtype}(\mathsf{m},\mathit{pred}(\mathsf{R})) \;=\; \overline{\mathsf{B}}\rightarrow\mathsf{B}_0\ \ \text{implies}\ \ \overline{\mathsf{C}} \;=\; \overline{\mathsf{B}}\ \ \text{and}\ \ \mathsf{C}_0 \;=\; \mathsf{B}_0}{\mathit{extend}(\mathsf{m},\mathsf{R},\overline{\mathsf{C}}\rightarrow\mathsf{C}_0)}$$

Fig. 6. Auxiliary definitions of basic FFJ (continued).

3.3 Auxiliary Definitions

For FFJ, we have modified some auxiliary definitions of FJ, and we have added some, as shown in Figures 5 and 6. The function *fields* returns the fields of a class including the fields of its subclasses and, in extension to FJ, the fields added by its refinements. The function *rfields* is similar except that the refinement chain is searched from right to left. This is useful to determine the fields that have been introduced before a given refinement, e.g., in the well-formedness rule of class refinements (Figure 9). The function *mtype* returns the type of a method. In contrast to FJ, in FFJ first the refinement chain is searched from left to right and, if an appropriate method is not found, the search is continued in the according superclass. The function *rmtype* is used to look for a method type from right to left in a refinement chain. This is needed when checking whether a method refinement really refines a method, e.g., in the auxiliary function *extend*. Function *mbody* looks up the most specific and most refined method body. That is, it returns the method body that is right-most and bottom-most in the combined refinement and inheritance hierarchy of an object, as explained in Section 2.1. The function *override* establishes whether a method of a superclass is appropriately overridden in a subclass, i.e., whether their signatures match. The function *introduce* establishes whether a method introduced by a class refinement has not been introduced before in its refinement chain. The function *extends* establishes whether a method refinement refines a method properly, i.e., whether an according method has been introduced before and their signatures match.

3.4 Evaluation

With the extended/modified auxiliary functions, the evaluation rules of FFJ, shown in Figure 7, are entirely the same as in FJ. In rule E-ProjNew, *fields* looks up all fields of a class, including the fields of its refinements. The fact that each class refinement must refine the constructor makes sure that the number of supplied values in a class instantiation is equal to the number of fields that *fields* returns. With the 'default values' extension, this will be different (see Section 4.2). In rule E-InvkNew, *mbody* returns the appropriate method body also considering method refinements, so nothing changes compared to FJ. The rule E-CastNew is not changed in FFJ since the subtype relation is equal to FJ. The remaining congruence rules are straightforward and equal in FJ and FFJ.

3.5 Typing

Figures 8 and 9 displays the type rules of FFJ. We took the rules for term typing from FJ (Figure 8). This was possible since we changed the auxiliary functions incorporating class, method, and constructor refinement. However, we modified and extended the well-formedness rules (Figure 9). The two rules for well-formed methods enforce that the type of the method body's term is a subtype of the declared return type, that a method of a superclass is being overridden appropriately, and that no subsequent refinement introduces a method with the same

$$\frac{\mathit{fields}(\mathsf{C}) \;=\; \overline{\mathsf{C}}\;\overline{\mathsf{f}}}{(\mathsf{new}\ \mathsf{C}(\overline{\mathsf{v}})).\mathsf{f}_i \;\longrightarrow\; \mathsf{v}_i} \qquad \text{(E-ProjNew)}$$

$$\frac{\mathit{mbody}(\mathsf{m},\mathsf{C}) \;=\; (\overline{\mathsf{x}},\mathsf{t}_0)}{(\mathsf{new}\ \mathsf{C}(\overline{\mathsf{v}})).\mathsf{m}(\overline{\mathsf{u}}) \;\longrightarrow\; [\overline{\mathsf{x}} \mapsto \overline{\mathsf{u}}, \mathsf{this} \mapsto \mathsf{new}\ \mathsf{C}(\overline{\mathsf{v}})]\,\mathsf{t}_0} \qquad \text{(E-ProjInvk)}$$

$$\frac{\mathsf{C} <: \mathsf{D}}{(\mathsf{D})(\mathsf{new}\ \mathsf{C}(\overline{\mathsf{v}})) \;\longrightarrow\; \mathsf{new}\ \mathsf{C}(\overline{\mathsf{v}})} \qquad \text{(E-CastNew)}$$

$$\frac{\mathsf{t}_0 \;\longrightarrow\; \mathsf{t}_0'}{\mathsf{t}_0.\mathsf{f} \;\longrightarrow\; \mathsf{t}_0'.\mathsf{f}} \qquad \text{(E-Field)}$$

$$\frac{\mathsf{t}_0 \;\longrightarrow\; \mathsf{t}_0'}{\mathsf{t}_0.\mathsf{m}(\overline{\mathsf{t}}) \;\longrightarrow\; \mathsf{t}_0'.\mathsf{m}(\overline{\mathsf{t}})} \qquad \text{(E-InvkRecv)}$$

$$\frac{\mathsf{t}_i \;\longrightarrow\; \mathsf{t}_i'}{\mathsf{v}_0.\mathsf{m}(\overline{\mathsf{v}},\ \mathsf{t}_i,\ \overline{\mathsf{t}}) \;\longrightarrow\; \mathsf{v}_0.\mathsf{m}(\overline{\mathsf{v}},\ \mathsf{t}_i',\ \overline{\mathsf{t}})} \qquad \text{(E-InvkArg)}$$

$$\frac{\mathsf{t}_i \;\longrightarrow\; \mathsf{t}_i'}{\mathsf{new}\ \mathsf{C}(\overline{\mathsf{v}},\ \mathsf{t}_i,\ \overline{\mathsf{t}}) \;\longrightarrow\; \mathsf{new}\ \mathsf{C}(\overline{\mathsf{v}},\ \mathsf{t}_i',\ \overline{\mathsf{t}})} \qquad \text{(E-NewArg)}$$

$$\frac{\mathsf{t}_0 \;\longrightarrow\; \mathsf{t}_0'}{(\mathsf{C})\mathsf{t}_0.\mathsf{f} \;\longrightarrow\; (\mathsf{C})\mathsf{t}_0'.\mathsf{f}} \qquad \text{(E-Cast)}$$

Fig. 7. Evaluation of basic FFJ.

Term typing $\boxed{\Gamma \vdash \mathsf{t} : \mathsf{C}}$

$$\frac{\mathsf{x} : \mathsf{C} \;\in\; \Gamma}{\Gamma \;\vdash\; \mathsf{x} : \mathsf{C}} \qquad \text{(T-Var)}$$

$$\frac{\Gamma \;\vdash\; \mathsf{t}_0 : \mathsf{C}_0 \qquad \mathit{fields}(\mathsf{C}_0) \;=\; \overline{\mathsf{C}}\;\overline{\mathsf{f}}}{\Gamma \;\vdash\; \mathsf{t}_0.\mathsf{f}_i : \mathsf{C}_i} \qquad \text{(T-Field)}$$

$$\frac{\Gamma \;\vdash\; \mathsf{t}_0 : \mathsf{C}_0 \qquad \mathit{mtype}(\mathsf{m},\mathsf{C}_0) \;=\; \overline{\mathsf{D}}\to\mathsf{C} \qquad \Gamma \;\vdash\; \overline{\mathsf{t}} : \overline{\mathsf{C}} \qquad \overline{\mathsf{C}} <: \overline{\mathsf{D}}}{\Gamma \;\vdash\; \mathsf{t}_0.\mathsf{m}(\overline{\mathsf{t}}) : \mathsf{C}} \qquad \text{(T-Invk)}$$

$$\frac{\mathit{fields}(\mathsf{C}) \;=\; \overline{\mathsf{D}}\;\overline{\mathsf{f}} \qquad \Gamma \;\vdash\; \overline{\mathsf{t}} : \overline{\mathsf{C}} \qquad \overline{\mathsf{C}} <: \overline{\mathsf{D}}}{\Gamma \;\vdash\; \mathsf{new}\ \mathsf{C}(\overline{\mathsf{t}}) : \mathsf{C}} \qquad \text{(T-New)}$$

$$\frac{\Gamma \;\vdash\; \mathsf{t}_0 : \mathsf{D} \qquad \mathsf{D} <: \mathsf{C}}{\Gamma \;\vdash\; (\mathsf{C})\mathsf{t}_0 : \mathsf{C}} \qquad \text{(T-UCast)}$$

$$\frac{\Gamma \;\vdash\; \mathsf{t}_0 : \mathsf{D} \qquad \mathsf{C} <: \mathsf{D} \qquad \mathsf{C} \neq \mathsf{D}}{\Gamma \;\vdash\; (\mathsf{C})\mathsf{t}_0 : \mathsf{C}} \qquad \text{(T-DCast)}$$

$$\frac{\Gamma \;\vdash\; \mathsf{t}_0 : \mathsf{D} \qquad \mathsf{C} \not<: \mathsf{D} \qquad \mathsf{D} \not<: \mathsf{C} \qquad \textit{stupid warning}}{\Gamma \;\vdash\; (\mathsf{C})\mathsf{t}_0 : \mathsf{C}} \qquad \text{(T-SCast)}$$

Fig. 8. Typing in basic FFJ.

Method typing $\boxed{\mathsf{MD\ OK\ in\ C/R}}$

$$\frac{\begin{array}{c}\bar{\mathsf{x}} : \bar{\mathsf{C}}, \mathsf{this} : \mathsf{C} \vdash \mathsf{t}_0 : \mathsf{E}_0 \qquad \mathsf{E}_0 <: \mathsf{C}_0 \qquad \mathit{CT}(\mathsf{C}) = \mathsf{class\ C\ extends\ D}\ \{\ldots\} \\ \mathit{override}(\mathsf{m}, \mathsf{D}, \bar{\mathsf{C}} \to \mathsf{C}_0) \qquad \mathit{introduce}(\mathsf{m}, \mathsf{C})\end{array}}{\mathsf{C}_0\ \mathsf{m}(\bar{\mathsf{C}}\ \bar{\mathsf{x}})\ \{\ \mathsf{return}\ \mathsf{t}_0;\ \}\ \mathsf{OK\ in\ C}}$$

$$\frac{\begin{array}{c}\bar{\mathsf{x}} : \bar{\mathsf{C}}, \mathsf{this} : \mathsf{C} \vdash \mathsf{t}_0 : \mathsf{E}_0 \qquad \mathsf{E}_0 <: \mathsf{C}_0 \qquad \mathit{CT}(\mathsf{R}) = \mathsf{refines\ class\ C}\ \{\ldots\} \\ \mathit{introduce}(\mathsf{m}, \mathsf{R})\end{array}}{\mathsf{C}_0\ \mathsf{m}(\bar{\mathsf{C}}\ \bar{\mathsf{x}})\ \{\ \mathsf{return}\ \mathsf{t}_0;\ \}\ \mathsf{OK\ in\ R}}$$

Method refinement typing $\boxed{\mathsf{MR\ OK\ in\ R}}$

$$\frac{\begin{array}{c}\bar{\mathsf{x}} : \bar{\mathsf{C}}, \mathsf{this} : \mathsf{C} \vdash \mathsf{t}_0 : \mathsf{E}_0 \qquad \mathsf{E}_0 <: \mathsf{C}_0 \qquad \mathit{CT}(\mathsf{R}) = \mathsf{refines\ class\ C}\ \{\ldots \overline{\mathsf{MD}} \ldots\} \\ \mathsf{m}\ \text{not defined in}\ \overline{\mathsf{MD}} \qquad \mathit{extend}(\mathsf{m}, \mathsf{R}, \bar{\mathsf{C}} \to \mathsf{C}_0)\end{array}}{\mathsf{refines}\ \mathsf{C}_0\ \mathsf{m}(\bar{\mathsf{C}}\ \bar{\mathsf{x}})\ \{\ \mathsf{return}\ \mathsf{t}_0;\ \}\ \mathsf{OK\ in\ R}}$$

Class typing $\boxed{\mathsf{C\ OK}}$

$$\frac{\mathsf{KD} = \mathsf{C}(\bar{\mathsf{D}}\ \bar{\mathsf{g}}, \bar{\mathsf{C}}\ \bar{\mathsf{f}})\ \{\ \mathsf{super}(\bar{\mathsf{g}});\ \mathsf{this}.\bar{\mathsf{f}}{=}\bar{\mathsf{f}};\ \} \qquad \mathit{fields}(\mathsf{D}) = \bar{\mathsf{D}}\ \bar{\mathsf{g}} \qquad \overline{\mathsf{MD}}\ \mathsf{OK\ in\ C}}{\mathsf{class\ C\ extends\ D}\ \{\ \bar{\mathsf{C}}\ \bar{\mathsf{f}};\ \mathsf{KD}\ \overline{\mathsf{MD}}\ \}\ \mathsf{OK}}$$

Class refinement typing $\boxed{\mathsf{R\ OK}}$

$$\frac{\begin{array}{c}\mathsf{KR} = \mathsf{C}(\bar{\mathsf{E}}\ \bar{\mathsf{h}}, \bar{\mathsf{C}}\ \bar{\mathsf{f}})\ \{\ \mathsf{original}(\bar{\mathsf{h}});\ \mathsf{this}.\bar{\mathsf{f}}{=}\bar{\mathsf{f}};\ \} \qquad \mathit{rfields}(\mathit{pred}(\mathsf{R})) = \bar{\mathsf{E}}\ \bar{\mathsf{h}} \\ \overline{\mathsf{MD}}\ \mathsf{OK\ in\ R} \qquad \overline{\mathsf{MR}}\ \mathsf{OK\ in\ R} \qquad \mathsf{R} \prec: \mathsf{C} \qquad \mathit{CT}(\mathsf{C}) = \mathsf{class\ C}\ \ldots \\ \mathsf{C} \neq \mathsf{Object} \qquad \mathit{RT}(\mathsf{refines\ class\ C}\ \{\ \bar{\mathsf{C}}\ \bar{\mathsf{f}};\ \mathsf{KR}\ \overline{\mathsf{MD}}\ \overline{\mathsf{MD}}\ \}) = \mathsf{R}\end{array}}{\mathsf{refines\ class\ C}\ \{\ \bar{\mathsf{C}}\ \bar{\mathsf{f}};\ \mathsf{KR}\ \overline{\mathsf{MD}}\ \overline{\mathsf{MR}}\ \}\ \mathsf{OK}}$$

Fig. 9. Typing in basic FFJ (continued).

name (note that overloading is not allowed in FJ and FFJ). That is, *override* considers methods that are overridden by the given method and *introduce* considers methods that are introduced later and establishes whether they replace the given method or not. The well-formedness rule of method refinement enforces, beside the standard properties, that an according method is being introduced before (and not in the same class refinement) and that the signatures of the two methods match. The well-formedness rule for classes is similar to FJ. It enforces the well-formedness of the constructor, the fields, and the methods. The well-formedness rule of class refinement enforces, in addition, that there is a class to be refined, that all method refinements are well-formed, and that appropriate values are passed for the fields of the refinement and its predecessor. It relies on the composition engine in that it refers to the refinement table RT for the definition of a class refinement, given its name.

3.6 Type Soundness

We formulate the type soundness of FFJ via the standard theorems Preservation and Progress [37]. The proof is the same as the one of FJ [18], except for some minor modifications. This is the case because the changes and extensions basic FFJ makes to FJ are largely concerned with the method and field lookup and do not interfere too much with the evaluation order and type system. In some of our extensions this will be different, as we will explain in the next section. See Appendix B.1 for the complete proof and further explanations.

4 Extensions for SWR

In this section, we integrate each extension individually into FFJ, obtaining:
- FFJ$_{ME}$—Method extension
- FFJ$_{DV}$—Default values
- FFJ$_{SD}$—Superclass declaration
- FFJ$_{BR}$—Backward references

For each extension we show how the syntax, evaluation, and type rules change and if and how the type soundness proof is affected. As the extensions are largely orthogonal, they can be combined freely to obtain a consistent and type-sound variant of FFJ. Nevertheless, we provide a full version of FFJ, that contains all extensions, along with a type soundness proof in the Appendix C.

4.1 FFJ$_{ME}$—Method Extension

In basic FFJ, a method refinement replaces the body of the method that is refined. In FFJ$_{ME}$, method bodies may invoke original; otherwise a method is not well-formed. The keyword original refers to the method that is refined.

To obtain FFJ$_{ME}$, we make the following changes to FFJ. First, we extend the syntax such that original may occur in terms:

$$\begin{array}{lrr} \mathsf{t} ::= & & \textit{terms:} \\ & \ldots & \textit{basic } \text{FFJ } \textit{terms} \\ & \mathsf{original}(\overline{\mathsf{t}}) & \textit{original invocation} \end{array}$$

Second, we modify *mbody* such that it substitutes every occurrence of original with the method body that is being refined (arguments are renamed); if the refined body contains original in turn, the process is repeated. The evaluation rule E-ProjInvk (Figure 7) is divided into two new rules as follows:

$$\frac{\mathit{mbody}(\mathsf{m},\mathsf{C}) = (\overline{\mathsf{x}},\mathsf{t}_0) \qquad \mathit{succ}(\mathsf{C}) = \mathsf{Object}}{(\mathsf{new}\ \mathsf{C}(\overline{\mathsf{v}})).\mathsf{m}(\overline{\mathsf{u}}) \ \longrightarrow\ [\overline{\mathsf{x}} \mapsto \overline{\mathsf{u}}, \mathsf{this} \mapsto \mathsf{new}\ \mathsf{C}(\overline{\mathsf{v}})]\,\mathsf{t}_0} \qquad \text{(E-ProjInvk1)}$$

$$\frac{\mathit{mbody}(\mathsf{m},\mathsf{C}) = (\overline{\mathsf{x}},\mathsf{t}_0) \qquad \mathit{last}(\mathsf{C}) = \mathsf{R}}{(\mathsf{new}\ \mathsf{C}(\overline{\mathsf{v}})).\mathsf{m}(\overline{\mathsf{u}}) \ \longrightarrow\ [\overline{\mathsf{x}} \mapsto \overline{\mathsf{u}}, \mathsf{this} \mapsto \mathsf{new}\ \mathsf{C}(\overline{\mathsf{v}})]\ \mathit{eval}(\mathsf{m},\mathsf{R},\mathsf{t}_0)} \qquad \text{(E-ProjInvk2)}$$

The latter rule uses an auxiliary function *eval* that performs the actual substitution (see Appendix A.1).

Third, we have to add a premise to the well-formedness rule of method refinements of Figure 9 to let the type system make sure that every body of a well-formed method refinement contains a reference to original:

Method refinement typing $\boxed{\mathsf{MR\ OK\ in\ R}}$

$$\frac{\begin{array}{c}\overline{\mathsf{x}} : \overline{\mathsf{C}}, \mathsf{this} : \mathsf{C}\ \vdash\ \mathsf{t}_0 : \mathsf{E}_0 \qquad \mathsf{E}_0 <: \mathsf{C}_0 \qquad \mathit{CT}(\mathsf{R})\ =\ \mathsf{refines\ class}\ \{\ \ldots\}\\ \mathit{extend}(\mathsf{m},\mathsf{R},\overline{\mathsf{C}} \to \mathsf{C}_0) \qquad \mathsf{t}_0\ \text{contains}\ \mathsf{original}\end{array}}{\mathsf{refines}\ \mathsf{C}_0\ \mathsf{m}(\overline{\mathsf{C}}\ \overline{\mathsf{x}})\ \{\ \mathsf{return}\ \mathsf{t}_0;\ \}\ \mathsf{OK\ in\ R}}$$

Analogously, we have to add a premise to the two well-formedness rules of method typing in order to reject method declarations whose bodies contain original (see Appendix C).

The introduction of original to method bodies does not interfere with the evaluation order and the type system. Function *eval* substitutes all occurrences of original with the method bodies that are refined, which effectively allows a refinement to extend a method body. As a consequence, method bodies in FFJ$_{ME}$ are indistinguishable from the ones in basic FFJ. Evaluation and typing in FFJ$_{ME}$ can proceed similarly to basic FFJ. Consequently, FFJ$_{ME}$ is type-sound (see Appendix B.2 for more details).

4.2 FFJ$_{DV}$—Default Values

In order to include default values, we make some changes to FFJ, obtaining FFJ$_{DV}$. First, we allow instantiations of classes to supply only a subsequence of arguments to the constructor. Since the type checker cannot always recognize which formal arguments are meant, such a subsequence must match a prefix of the sequence of expected arguments, which is similar to user-defined default values in C++.

With this mechanism, classes can be instantiated without knowledge of refinements that subsequently add new fields. In order to assign proper values to the fields that have not been initialized, we use default values generated by the FFJ$_{DV}$ calculus. Default values are in some sense the neutral elements of a given type. We use generated default values instead of null or predefined values to keep the calculus simple. Both latter options would be possible but are beyond the scope of the paper.

Furthermore, only bottom-level classes of the class hierarchy can be instantiated using default values. The reason is that otherwise mapping arguments to fields is difficult. See Section 5 for a discussion.

In FFJ$_{DV}$, default values can be computed solely on the basis of a class definition (incl. its refinements). That is, the auxiliary function $\textit{default}(\mathsf{C})$ computes the default value of class C without relying on extra information supplied by the programmer. For a simple class without fields, the default value is just the empty instantiation, e.g., $\textit{default}(\mathsf{Object}) = \mathsf{new\ Object()}$ or $\textit{default}(\mathsf{Expr}) = \mathsf{new\ Expr()}$. The default value of a more complex class is computed recursively:

$$\textit{default}(\mathsf{Object}) = \mathsf{new\ Object()}$$

$$\frac{CT(\mathsf{C}) = \mathsf{class\ C\ extends\ D}\ \{\ \overline{\mathsf{C}}\ \overline{\mathsf{f}}\ \ldots\}}{\textit{default}(\mathsf{C}) = \mathsf{new\ C}(\textit{default}(\mathsf{C}_1), \ldots, \textit{default}(\mathsf{C}_n))}$$

Using default values, we divide the evaluation rule E-ProjNew for projection of Figure 7 into E-ProjNew1 and E-ProjNew2:

$$\frac{\textit{fields}(\mathsf{C}) = \overline{\mathsf{C}}\ \overline{\mathsf{f}} \quad |\overline{\mathsf{v}}| \geq i}{(\mathsf{new\ C}(\overline{\mathsf{v}})).\mathsf{f}_i \longrightarrow \mathsf{v}_i} \qquad \text{(E-ProjNew1)}$$

$$\frac{\textit{fields}(\mathsf{C}) = \overline{\mathsf{C}}\ \overline{\mathsf{f}} \quad |\overline{\mathsf{v}}| < i}{(\mathsf{new\ C}(\overline{\mathsf{v}})).\mathsf{f}_i \longrightarrow \textit{default}(\mathsf{C}_i)} \qquad \text{(E-ProjNew2)}$$

If the sequence of supplied values contains the value of the projected field f_i (E-ProjNew1), nothing changes compared to basic FFJ. On the other hand, if the sequence of supplied values does not contain the value of the projected field f_i (E-ProjNew2), a default value v is supplied.

Furthermore, we have to add a new type rule which specifies the type of a default value:

$$\vdash \textit{default}(\mathsf{C}) : \mathsf{C} \qquad \text{(T-Default)}$$

A default value of a class belongs always to the class' type. This is a straightforward consequence of the semantics of default values.

Finally, we have to update the type rule T-New (Figure 8) in order to allow a smaller number of values to be supplied than the number of fields a class actually contains (incl. its refinements and superclasses):

$$\frac{\mathit{fields}(\mathsf{C}) \; = \; \overline{\mathsf{D}}\;\overline{\mathsf{f}}, \overline{\mathsf{E}}\;\overline{\mathsf{h}} \qquad \Gamma \vdash \overline{\mathsf{t}} : \overline{\mathsf{C}} \qquad \overline{\mathsf{C}} <: \overline{\mathsf{D}}}{\Gamma \vdash \mathsf{new\ C}(\overline{\mathsf{t}}) : \mathsf{C}} \quad \text{(T-New)}$$

The type rules that make sure that the number of arguments of a constructor match the number of fields ($\mathsf{C\ OK}$ and $\mathsf{R\ OK}$, shown in Figure 9), do not need to change since only the number of value varies and not the number of formal constructor arguments.

The modified evaluation and type rules induce some changes in basic FFJ's type soundness proof in order to carry over to FFJ$_{DV}$. Essentially, the cases of instantiations of classes and of projections of fields change such that also subsequences of arguments for a constructor are accepted. In Appendix B.3, we explain how the proof changes and show that FFJ$_{DV}$ is type-sound.

4.3 FFJ$_{SD}$—Superclass Declaration

With this extension, each class refinement declares a superclass, possibly Object. Effectively, a class has multiple superclasses, declared by itself and its refinements. We have to change FFJ in several ways to take multiple superclasses into account, obtaining FFJ$_{SD}$.

First, we modify the syntax rules of FFJ such that a class refinement declares a superclass and a constructor refinement passes the values intended for its superclass via super:

$$\begin{array}{ll} \mathsf{CR} ::= & \textit{class refinements:} \\ \quad \mathsf{refines\ class\ C\ extends\ D\ \{\ \overline{C}\ \overline{f};\ KR\ \overline{MD}\ \overline{MR}\ \}} & \\[1ex] \mathsf{KR} ::= & \textit{constructor refinements:} \\ \quad \mathsf{refines\ C(\overline{D}\ \overline{g},\ \overline{E}\ \overline{h},\ \overline{C}\ \overline{f})\ \{\ super(g);\ original(\overline{h});\ this.\overline{f}{=}\overline{f};\ \}} & \end{array}$$

Second, we extend the subtype relation in order to consider also the superclasses of a class that have been declared by its refinements:

$$\frac{CT(\mathsf{R}) \; = \; \mathsf{refines\ class\ C\ extends\ D\ \{\ \ldots\}} \qquad \mathsf{R} \prec: \mathsf{C}}{\mathsf{C} <: \mathsf{E}}$$

Third, we have to modify and extend some auxiliary functions. Now, the function *fields* also collects the fields of the superclasses declared by the class refinements:

$$\frac{CT(\mathsf{R}) \; = \; \mathsf{refines\ class\ C\ extends\ D\ \{\ \overline{C}\ \overline{f};\ KR\ \overline{MD}\ \overline{MR}\ \}}}{\mathit{fields}(\mathsf{C}) \; = \; \mathit{fields}(\mathsf{D}),\ \overline{\mathsf{C}}\ \overline{\mathsf{f}},\ \mathit{fields}(\mathit{succ}(\mathsf{C}))}$$

Two new rules for method type and body lookup incorporate also the superclasses of class refinements:

$$\frac{\begin{array}{c} CT(\mathsf{R}) \;=\; \textsf{refines class C extends D } \{ \; \overline{\mathsf{C}}\;\overline{\mathsf{f}}; \; \mathsf{KR} \; \overline{\mathsf{MD}} \; \overline{\mathsf{MR}} \; \} \\ \mathsf{m} \text{ is not defined in } \overline{\mathsf{MD}} \qquad \neg \mathit{mtype}(\mathsf{m}, \mathit{succ}(\mathsf{R})) \end{array}}{\mathit{mtype}(\mathsf{m}, \mathsf{R}) \;=\; \mathit{mtype}(\mathsf{m}, \mathsf{D})}$$

$$\frac{\begin{array}{c} CT(\mathsf{R}) \;=\; \textsf{refines class C extends D } \{ \; \overline{\mathsf{C}}\;\overline{\mathsf{f}}; \; \mathsf{KR} \; \overline{\mathsf{MD}} \; \overline{\mathsf{MR}} \; \} \\ \mathsf{m} \text{ is not defined in } \overline{\mathsf{MD}} \text{ or } \overline{\mathsf{MR}} \qquad \neg \mathit{mbody}(\mathsf{m}, \mathit{succ}(\mathsf{R})) \end{array}}{\mathit{mbody}(\mathsf{m}, \mathsf{R}) \;=\; \mathit{mbody}(\mathsf{m}, \mathsf{D})}$$

The premises $\neg mtype(\ldots)$ and $\neg mbody(\ldots)$ are necessary to make sure that superclasses are only looked up in the case that there are no matching methods in subsequent class refinements. The remaining rules of the auxiliary functions are simply updated to be compatible with the new syntax of class refinements.

Fourth, the well-formedness rules for methods and class refinements change. Method declarations in class refinements must override methods of the class refinement's superclasses properly:

$$\frac{\begin{array}{c} \overline{\mathsf{x}} : \overline{\mathsf{C}}, \mathsf{this} : \mathsf{C} \;\vdash\; \mathsf{t}_0 : \mathsf{E}_0 \quad \mathsf{E}_0 <: \mathsf{C}_0 \quad CT(\mathsf{R}) \;=\; \textsf{refines class C extends D } \{ \ldots \} \\ \mathit{override}(\mathsf{m}, \mathsf{D}, \overline{\mathsf{C}} \to \mathsf{C}_0) \qquad \mathit{introduce}(\mathsf{m}, \mathsf{R}) \end{array}}{\mathsf{C}_0 \; \mathsf{m}(\overline{\mathsf{C}} \; \overline{\mathsf{x}}) \; \{ \; \textsf{return } \mathsf{t}_0; \; \} \textsf{ OK in R}}$$

The well-formedness rule of class refinement also enforces that the arguments for the superclass' constructor are passed properly by a constructor refinement:

$$\frac{\begin{array}{c} \mathsf{KR} \;=\; \mathsf{C}(\overline{\mathsf{D}} \; \overline{\mathsf{g}}, \; \overline{\mathsf{E}} \; \overline{\mathsf{h}}, \; \overline{\mathsf{C}} \; \overline{\mathsf{f}}) \; \{ \; \mathsf{super}(\overline{\mathsf{g}}); \; \mathsf{original}(\overline{\mathsf{h}}); \; \mathsf{this}.\overline{\mathsf{f}}{=}\overline{\mathsf{f}}; \; \} \\ \mathit{fields}(\mathsf{D}) \;=\; \overline{\mathsf{D}} \; \overline{\mathsf{g}} \qquad \mathit{rfields}(\mathit{pred}(\mathsf{R})) \;=\; \overline{\mathsf{E}} \; \overline{\mathsf{h}} \qquad \overline{\mathsf{MD}} \textsf{ OK in R} \qquad \overline{\mathsf{MR}} \textsf{ OK in R} \\ \mathsf{R} \;\prec:\; \mathsf{C} \qquad CT(\mathsf{C}) \;\neq\; \bullet \qquad \mathsf{C} \;\neq\; \mathsf{Object} \qquad \mathit{inherit}(\mathsf{C}, \mathsf{R}) \\ RT(\textsf{refines class C extends D } \{ \; \overline{\mathsf{C}} \; \overline{\mathsf{f}}; \; \mathsf{KR} \; \overline{\mathsf{MD}} \; \overline{\mathsf{MD}} \; \}) \;=\; \mathsf{R} \end{array}}{\textsf{refines class C extends D } \{ \; \overline{\mathsf{C}} \; \overline{\mathsf{f}}; \; \mathsf{KR} \; \overline{\mathsf{MD}} \; \overline{\mathsf{MR}} \; \} \textsf{ OK}}$$

Note that, using the auxiliary function *inherit*, the rule checks whether a class refinement does not declare a superclass that has been declared before in the refinement chain (see Appendix A.2). The remaining type and evaluation rules are simply updated considering the new syntax of class refinements.

Finally, the type soundness proof changes minimally in that, in the cases of casts, also multiple superclasses are considered. In Appendix B.4, we explain how the proof changes and show that FFJ$_{SD}$ is type-sound.

4.4 FFJ$_{BR}$—Backward References

In order to disallow forward references in FFJ, we have to modify the well-formedness rules for methods, method refinements, classes, and class refinements, obtaining FFJ$_{BR}$. To this end, we introduce a predicate *backward* that establishes whether a reference to a member or class is a backward reference from a given class or refinement. Function *backward* expects either a pair of a list of classes and a class/refinement or a pair of a term and a class/refinement. The former case is simple, as the compiler simply checks whether a class has been introduced before another class or refinement (i.e., by a previous feature). In the latter case, the given term is traversed and each subterm is checked for backward references to classes (in casts and class instantiations) and to members (in field accesses and method invocations). See Appendix A.3 for the definition of *backward*.

We use *backward* in the well-formedness rules of FFJ_{BR} for methods, method refinements, classes, and class refinements. For brevity, we give here only the rules for methods and classes:

Method typing $\boxed{\mathsf{MD\ OK\ in\ C}}$

$$\frac{\begin{array}{c}\overline{\mathsf{x}} : \overline{\mathsf{C}}, \mathsf{this} : \mathsf{C} \vdash \mathsf{t}_0 : \mathsf{E}_0 \quad \mathsf{E}_0 <: \mathsf{C}_0 \quad CT(\mathsf{C}) = \mathsf{class\ C\ extends\ D}\ \{\ \ldots\ \} \\ override(\mathsf{m}, \mathsf{D}, \overline{\mathsf{C}} \rightarrow \mathsf{C}_0) \qquad introduce(\mathsf{m}, \mathsf{C}) \\ backward(\overline{\mathsf{C}}, \mathsf{C}) \qquad backward(\mathsf{C}_0, \mathsf{C}) \qquad backward(\mathsf{t}_0, \mathsf{C})\end{array}}{\mathsf{C}_0\ \mathsf{m}(\overline{\mathsf{C}}\ \overline{\mathsf{x}})\ \{\ \mathsf{return}\ \mathsf{t}_0;\ \}\ \mathsf{OK\ in\ C}}$$

Class typing $\boxed{\mathsf{C\ OK}}$

$$\frac{\begin{array}{c}\mathsf{KD} = \mathsf{C}(\overline{\mathsf{D}}\ \overline{\mathsf{g}}, \overline{\mathsf{C}}\ \overline{\mathsf{f}})\ \{\ \mathsf{super}(\overline{\mathsf{g}});\ \mathsf{this}.\overline{\mathsf{f}}{=}\overline{\mathsf{f}};\ \} \quad fields(\mathsf{D}) = \overline{\mathsf{D}}\ \overline{\mathsf{g}} \quad \overline{\mathsf{MD}}\ \mathsf{OK\ in\ C} \\ backward(\overline{\mathsf{C}}, \mathsf{C}) \qquad backward(\overline{\mathsf{D}}, \mathsf{C}) \qquad backward(\mathsf{D}, \mathsf{C})\end{array}}{\mathsf{class\ C\ extends\ D}\ \{\ \overline{\mathsf{C}}\ \overline{\mathsf{f}};\ \mathsf{KD}\ \overline{\mathsf{MD}}\ \}\ \mathsf{OK}}$$

The remaining well-formedness rules are updated analogously (see Appendix C).

The modified well-formedness rules of FFJ_{BR} do not interfere with the type soundness proof of basic FFJ (Appendix B.1). This is easy to see since the well-formedness rules of FFJ_{BR} reject some programs that are well-formed in FFJ. That is, the set of well-formed FFJ_{BR} programs is a subset of the set of well-formed FFJ programs. Consequently, the type soundness theorem also holds for FFJ_{BR}, i.e., FFJ_{BR} is type-sound.

4.5 Type Soundness of FFJ with all Extensions

We have shown that FFJ_{ME}, FFJ_{DV}, FFJ_{SD} and FFJ_{BR} are type-sound. As these extensions are largely orthogonal, it is easy to show that FFJ with all extensions together is type-sound as well. For the complete syntax, evaluation, and typing rules as well as the type soundness proof we refer the interested reader to Appendix C.

5 Discussion

FFJ and its extensions model several mechanisms of contemporary feature-oriented languages and tools. In Table 1, we compare FFJ with a selection, namely Java, FSTComposer [7] and two versions of Jak: an earlier version [8], which we call Jak_1, and an extended version [34], which we call Jak_2.

	FJ	FFJ	FFJ_{ME}	FFJ_{DV}	FFJ_{SD}	FFJ_{BR}
Java	$\surd$					
Jak_1	$\surd$	$\surd$	$\surd$[a]	$\surd$[b]		
Jak_2	$\surd$	$\surd$	$\surd$	$\surd$[b]		$\surd$[d]
FSTComposer	$\surd$	$\surd$	$\surd$	$\surd$[b]	$\surd$[c]	

[a] While Jak_1 supports method extensions, in some versions it does not inform the programmer that a refinement replaces a method.
[b] Jak and FSTComposer support user-defined default values (not generated values).
[c] In FSTComposer, a class refinement may declare new interfaces but not superclasses.
[d] Jak_2 does not cover all checks of FFJ_{BR} for disallowing forward references, e.g., method signatures are not checked.

Table 1. Overview of which mechanisms are supported by which calculus and which language or tool.

It is important to note that the purpose of FFJ and its extensions is to reason about properties of feature-oriented languages and tools, like type soundness, formally. The formalizations model only a small subset of their real-world counterparts, though an important one with respect to the type system. But, as we have shown, in some cases FFJ is more consequent than contemporary feature-oriented languages, e.g., FFJ disallows forward reference to types referred to from method signatures, which is not enforced by Jak [34].

Nevertheless, FFJ is far from being a full programming language, and so it is not surprising that some extensions do not exert their full power. For example, default values in FFJ_{DV} are used to model what is necessary in order not to violate the principle of SWR. But, in the scope of FFJ, default values are only of limited use. Having multiple independent refinements of a class, it is difficult to assign new values to fields that have been initialized with a default value during compilation. For example, if we add two independent refinements to a class A, each adding a new field, it is difficult to assign values via setB and setC for both of them:

```
class A { E a; }
refines class A { E b; A setB(E b) { return new A(this.a,b); }
refines class A { E c; A setC(E c) { return new A(this.a,c); }
```

The reason is that, doing so, both refinements would have to know about each other in order to supply also a value for the field of the other refinement, e.g.:

```
refines class A { E b; A setB(C b) { return new A(this.a,b,this.c); }
refines class A { E c; A setC(E c) { return new A(this.a,this.b,c); }
```

In this case, both refinements know about each other, which violates the principle of SWR. Letting only the second refinement know about the first fixes the composition order. The first refinement must exist so that the second can be applied, although both are semantically independent.

However, this is only a problem of FFJ and not of feature-oriented languages in general, and it occurs because of the lack of assignment. FJ and FFJ omit assignments in order to simplify the formal system. Real languages and tools are more powerful. Another example is that, when instantiating a class with a subsequence of arguments, the subsequence must match a prefix of the constructor's argument list. This impairs the refinements of classes that have subclasses. Therefore, we require that only bottom-level classes are instantiated with default values (see Section 4.2). Again, this limitation appears only in the formalism and vanishes in a real programming language or tool.

6 Related Work

FFJ is inspired by several feature-oriented languages and tools, most notably AHEAD [8], FeatureC++ [5], FSTComposer [7], and Prehofer's feature-oriented Java extension [32]. A key aim of these languages is to separate the implementation of software artifacts, e.g., classes and methods, from the definition of features. That is, classes and refinements are not annotated or declared to belong to a feature. There is no statement in the program text that defines explicitly a connection between code and features. Instead, the mapping of software artifacts to features is established via containment hierarchies, as explained in Section 2. The advantage of this approach is that a feature's implementation can include, beside classes in form of Java files, also other supporting documents, e.g., documentation in form of HTML files, grammar specification in form of JavaCC files, or build scripts and deployment descriptors in form of XML files [8]. To this end, feature composition merges classes not only with their refinements but also other artifacts such as HTML or XML files with their respective refinements [2,7].

Jiazzi and C# 3.0 are two languages that are commonly not associated with FOP but that provide very similar mechanisms. With Jiazzi, a programmer can aggregate several classes in a component and compose them in a feature-oriented fashion [26]. The mapping between code and components is described externally by means of a separate linker language. In C#, there is the possibility to specify a class refinement, which is called a partial class. Aggregating a set of (partial) classes in a file system directory is very similar to feature-oriented languages, in which a feature's constituting artifacts are aggregated in a containment hierarchy. A difference is that also the class that is refined must be declared as partial, which has to be anticipated by the programmer.

Another class of programming languages that provide mechanisms for the definition and extension of classes and class hierarchies includes, e.g., *ContextL* [16], *Scala* [30], *Classbox/J* [9], and *Jx* [28]. The difference to feature-oriented languages is that they provide explicit language constructs for aggregating the classes that belong to a feature, e.g., family classes, classboxes, or layers. This implies that noncode software artifacts cannot be included in a feature [6].

Similarly, related work on a formalization of the key concepts underlying FOP has not separated features from code. Especially, calculi for mixins [14,10,1,20], traits [24], family polymorphism and virtual classes [19,13,17,11], dependent types [30,29], dependent classes [15], and nested inheritance [28] either support only the refinement of single classes or expect the classes that form a semantically coherent unit (i.e., that belong to a feature) to be located in a physical module that is defined in the host programming language. For example, a virtual class is by definition an inner class of the enclosing object, or a classbox is a package that aggregates a set of related classes. Thus, FFJ differs from previous approaches in that it relies on contextual information that has been collected by the composition engine, e.g., the features' composition order or the mapping of classes and refinements to features.

A different line of research aims at the language-independent reasoning about features [8,25,7,23]. gDeep is most related to FFJ since it provides a type system for feature-oriented languages that is language-independent [3]. The idea is that the recursive process of merging software artifacts and their refinements, when composing hierarchically structured features, is very similar for different host languages, e.g., for Java, C#, and XML. gDeep describes formally how feature composition is performed and what type constraints have to be satisfied. In contrast, FFJ does not aspire to be language-independent, although the key concepts can certainly be used with different languages. The advantage of FFJ is that its type system can be used to check whether terms of the host language (Java/FJ) violate the principles of FOP and SWR, e.g., whether methods refer to classes that have been added subsequently. Due to its language independence, gDeep does not have enough information to perform such checks; however, both FFJ and gDeep could be integrated.

Czarnecki et al. have presented an automatic verification procedure for ensuring that no ill-structured UML model template instances will be generated from a valid feature selection [12]. They use OCL constraints to express and implement a type system for model composition. In this sense, their aim is very similar to FFJ, but limited to UML diagram artifacts.

Kästner et al. have implemented a tool, called CIDE, that allows a developer to refactor a legacy software system into features [22,23]. In contrast to the feature-oriented languages and tools we have discussed so far, the link between code and features is established via annotations. Hence, the code of features is not separated physically. If a user selects a set of features, all code that is annotated with features that are not present in the selection is removed. A set of type rules ensures that only well-typed programs can be generated, e.g., if a method declaration is removed, the remaining code must not contain calls to that method anymore. This guaranty of well-formedness is very similar to FFJ. In some sense, our approach and the approach of CIDE are two sides of the same coin: one aims at feature composition and the other at feature decomposition.

7 Conclusion

FOP is a paradigm that incorporates programming language technology, program generation techniques, and SWR. The question of what a type system for FOP should look like has not been answered before [34]. We have presented a type system for FOP on top of a simple, feature-oriented language, called FFJ. The type system can be used to check before compilation whether a given composition of features is safe. FFJ is interesting insofar as it incorporates reasoning at the programming language level and the composition engine at the meta level, which is different from previous work. We have been able to show that FFJ's type system is sound.

Furthermore, we have explored several variations of FFJ for SWR and have shown that they are type-sound as well. FFJ is a promising start in experimenting with further extensions such as separate compilation, method signature extension, field overriding, feature interfaces, optional method refinements, and many more.

References

1. D. Ancona, G. Lagorio, and E. Zucca. Jam—Designing a Java Extension with Mixins. *ACM Trans. Programming Languages and Systems*, 25(5):641–712, 2003.
2. F. Anfurrutia, O. Díaz, and S. Trujillo. On Refining XML Artifacts. In *Proc. Int'l. Conf. Web Engineering*, volume 4607 of *LNCS*, pages 473–478. Springer-Verlag, 2007.
3. S. Apel and D. Hutchins. An Overview of the gDeep Calculus. Technical Report MIP-0712, Department of Informatics and Mathematics, University of Passau, 2007.
4. S. Apel, C. Kästner, T. Leich, and G. Saake. Aspect Refinement - Unifying AOP and Stepwise Refinement. *Journal of Object Technology – Special Issue: TOOLS EUROPE'07*, 2007.
5. S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proc. Int'l. Conf. Generative Programming and Component Engineering*, volume 3676 of *LNCS*, pages 125–140. Springer-Verlag, 2005.
6. S. Apel, T. Leich, and G. Saake. Aspectual Feature Modules. *IEEE Trans. Software Engineering*, 34(2):162–180, 2008.
7. S. Apel and C. Lengauer. Superimposition: A Language-Independent Approach to Software Composition. In *Proc. Int'l. Symp. Software Composition*, volume 4954 of *LNCS*, pages 20–35. Springer-Verlag, 2008.
8. D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Software Engineering*, 30(6):355–371, 2004.
9. A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/J: Controlling the Scope of Change in Java. In *Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications*, pages 177–189. ACM Press, 2005.
10. V. Bono, A. Patel, and V. Shmatikov. A Core Calculus of Classes and Mixins. In *Proc. Europ. Conf. Object-Oriented Programming*, volume 1628 of *LNCS*, pages 43–66. Springer-Verlag, 1999.

11. D. Clarke, S. Drossopoulou, J. Noble, and T. Wrigstad. Tribe: A Simple Virtual Class Calculus. In *Proc. Int'l. Conf. Aspect-Oriented Software Development*, pages 121–134. ACM Press, 2007.
12. K. Czarnecki and K. Pietroszek. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In *Proc. Int'l. Conf. Generative Programming and Component Engineering*, pages 211–220. ACM Press, 2006.
13. E. Ernst, K. Ostermann, and W. Cook. A Virtual Class Calculus. In *Proc. Int'l. Symp. Principles of Programming Languages*, pages 270–282. ACM Press, 2006.
14. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and Mixins. In *Proc. Int'l. Symp. Principles of Programming Languages*, pages 171–183. ACM Press, 1998.
15. V. Gasiunas, M. Mezini, and K. Ostermann. Dependent Classes. In *Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications*, pages 133–152. ACM Press, 2007.
16. R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-Oriented Programming. *J. Object Technology*, 7(3):125–151, 2008.
17. D. Hutchins. Eliminating Distinctions of Class: Using Prototypes to Model Virtual Classes. In *Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–19. ACM Press, 2006.
18. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Programming Languages and Systems*, 23(3):396–450, 2001.
19. A. Igarashi, C. Saito, and M. Viroli. Lightweight Family Polymorphism. In *Proc. Asian Symp. Programming Languages and Systems*, volume 3780 of *LNCS*, pages 161–177. Springer-Verlag, 2005.
20. T. Kamina and T. Tamai. McJava – A Design and Implementation of Java with Mixin-Types. In *Proc. Asian Symp. Programming Languages and Systems*, volume 3302 of *LNCS*, pages 398–414. Springer-Verlag, 2004.
21. K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.
22. C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proc. Int'l. Conf. Software Engineering*. ACM Press, 2008.
23. C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory. Language-Independent Safe Decomposition of Legacy Applications into Features. Technical Report 02/2008, School of Computer Science, University of Magdeburg, 2008.
24. L. Liquori and A. Spiwack. FeatherTrait: A Modest Extension of Featherweight Java. *ACM Trans. Programming Languages and Systems*, 30(2):1–32, 2008.
25. R. Lopez-Herrejon, D. Batory, and C. Lengauer. A Disciplined Approach to Aspect Composition. In *Proc. Int'l. Symp. Partial Evaluation and Semantics-Based Program Manipulation*, pages 68–77. ACM Press, 2006.
26. S. McDirmid, M. Flatt, and W. Hsieh. Jiazzi: New-Age Components for Old-Fashioned Java. In *Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications*, pages 211–222. ACM Press, 2001.
27. N. McEachen and R. T. Alexander. Distributing Classes with Woven Concerns: An Exploration of Potential Fault Scenarios. In *Proc. Int'l. Conf. Aspect-Oriented Software Development*, pages 192–200. ACM Press, 2005.
28. N. Nystrom, S. Chong, and A. Myers. Scalable Extensibility via Nested Inheritance. In *Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications*, pages 99–115. ACM Press, 2004.

29. M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A Nominal Theory of Objects with Dependent Types. In *Proc. Europ. Conf. Object-Oriented Programming*, volume 2743 of *LNCS*, pages 201–224. Springer-Verlag, 2003.
30. M. Odersky and M. Zenger. Scalable Component Abstractions. In *Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications*, pages 41–57. ACM Press, 2005.
31. K. Ostermann. Nominal and Structural Subtyping in Component-Based Programming. *J. Object Technology*, 7(1):121–145, 2008.
32. C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. Europ. Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 419–443. Springer-Verlag, 1997.
33. A. Taivalsaari. On the Notion of Inheritance. *ACM Comp. Surv.*, 28(3):438–479, 1996.
34. S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. In *Proc. Int'l. Conf. Generative Programming and Component Engineering*, pages 95–104. ACM Press, 2007.
35. M. Torgersen. The Expression Problem Revisited. In *Proc. Europ. Conf. Object-Oriented Programming*, volume 3086 of *LNCS*, pages 123–143. Springer-Verlag, 2004.
36. N. Wirth. Program Development by Stepwise Refinement. *Comm. ACM*, 14(4):221–227, 1971.
37. A. K. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. *Inf. Comput.*, 115(1):38–94, 1994.

A Auxiliary Functions

A.1 Evaluation of Method Refinements

The auxiliary function *eval* substitutes every occurrence of $\mathsf{original}$ with the body of the method that is refined. The arguments of the methods are renamed accordingly:

$$\frac{\begin{array}{c} CT(\mathsf{C}) \ = \ \mathsf{class\ C\ extends\ D}\ \{\ \ldots \overline{\mathsf{MD}}\ \} \\ \mathit{pred}(\mathsf{R}) \ = \ \mathsf{C} \qquad \mathsf{C_0\ m(\overline{C}\ \overline{x})}\ \{\ \mathsf{return\ t_0}\ \}\ \in \overline{\mathsf{MD}} \end{array}}{\mathit{eval}(\mathsf{m},\mathsf{R},\mathsf{t}) \ \longrightarrow \ [\mathsf{original}(\overline{\mathsf{y}}) \mapsto [\overline{\mathsf{x}} \mapsto \overline{\mathsf{y}}]\,\mathsf{t_0}]\,\mathsf{t}}$$

$$\frac{\begin{array}{c} CT(\mathsf{S}) \ = \ \mathsf{refines\ class\ C}\ \{\ \ldots \overline{\mathsf{MD}}\ \} \\ \mathit{pred}(\mathsf{R}) \ = \ \mathsf{S} \qquad \mathsf{C_0\ m(\overline{C}\ \overline{x})}\ \{\ \mathsf{return\ t_0}\ \}\ \in \overline{\mathsf{MD}} \end{array}}{\mathit{eval}(\mathsf{m},\mathsf{R},\mathsf{t}) \ \longrightarrow \ [\mathsf{original}(\overline{\mathsf{y}}) \mapsto [\overline{\mathsf{x}} \mapsto \overline{\mathsf{y}}]\,\mathsf{t_0}]\,\mathsf{t}}$$

$$\frac{\begin{array}{c} CT(\mathsf{S}) \ = \ \mathsf{refines\ class\ C}\ \{\ \ldots \overline{\mathsf{MR}}\ \} \\ \mathit{pred}(\mathsf{R}) \ = \ \mathsf{S} \qquad \mathsf{refines\ C_0\ m(\overline{C}\ \overline{x})}\ \{\ \mathsf{return\ t_0}\ \}\ \in \overline{\mathsf{MR}} \end{array}}{\mathit{eval}(\mathsf{m},\mathsf{R},\mathsf{t}) \ \longrightarrow \ \mathit{eval}(\mathsf{m},\mathsf{S},[\mathsf{original}(\overline{\mathsf{y}}) \mapsto [\overline{\mathsf{x}} \mapsto \overline{\mathsf{y}}]\,\mathsf{t_0}]\,\mathsf{t})}$$

$$\frac{\begin{array}{c} CT(\mathsf{S}) \ = \ \mathsf{refines\ class\ C}\ \{\ \ldots \overline{\mathsf{MD}}\ \overline{\mathsf{MR}}\ \} \\ \mathit{pred}(\mathsf{R}) \ = \ \mathsf{S} \qquad \mathsf{m}\text{ is not defined in }\overline{\mathsf{MR}}\text{ or }\overline{\mathsf{MD}} \end{array}}{\mathit{eval}(\mathsf{m},\mathsf{R},\mathsf{t}) \ \longrightarrow \ \mathit{eval}(\mathsf{m},\mathsf{S},\mathsf{t})}$$

A.2 Valid Superclass Declaration

The auxiliary function *inherit* returns whether a refinement declares no superclass that has been declared before in the refinement chain:

$$\mathit{inherit}(\mathsf{C}, \mathsf{C})$$

$$\frac{\mathit{super}(\mathsf{C}) \cap \mathit{super}(\mathsf{R}) = \{\mathsf{Object}\} \qquad \mathit{inherit}(\mathsf{C}, \mathit{pred}(\mathsf{R}))}{\mathit{inherit}(\mathsf{C}, \mathsf{R})}$$

It relies on the function *super* that returns all superclasses of a class or class refinement:

$$\mathit{super}(\mathsf{Object}) = \emptyset$$

$$\frac{CT(C) = \mathsf{class\ C\ extends\ D}\ \{\ \ldots\ \}}{\mathit{super}(\mathsf{C}) = \{\mathsf{D}\} \cup \mathit{super}(\mathsf{D}) \cup \mathit{super}(\mathit{succ}(\mathsf{C}))}$$

$$\frac{CT(R) = \mathsf{refines\ class\ C\ extends\ D}\ \{\ \ldots\ \}}{\mathit{super}(\mathsf{R}) = \{\mathsf{D}\} \cup \mathit{super}(\mathsf{D}) \cup \mathit{super}(\mathit{succ}(\mathsf{R}))}$$

A.3 Backward References

The function *backward* expects as arguments a class or refinement and another class or refinement and returns wheter the former has been introduced previously (we show here only the cases for classes):

$$\frac{\mathsf{D}\text{ has been introduced before }\mathsf{C}}{\mathit{backward}(\mathsf{D}, \mathsf{C})}$$

or a list of classes or refinements and a class or refinement:

$$\frac{\mathit{backward}(\mathsf{D}_1, \mathsf{C}) \ldots \mathit{backward}(\mathsf{D}_n, \mathsf{C})}{\mathit{backward}(\overline{\mathsf{D}}, \mathsf{C})}$$

Furthermore, *backward* may be used with a list of terms and a class or refinement:

$$\frac{\mathit{backward}(\mathsf{t}_1, \mathsf{C}) \ldots \mathit{backward}(\mathsf{t}_n, \mathsf{C})}{\mathit{backward}(\overline{\mathsf{t}}, \mathsf{C})}$$

or a single term and a class or refinement, in which the term may have six different shapes:

$$\mathit{backward}(\mathsf{x}, \mathsf{C}) \qquad \frac{\mathit{backward}(\overline{\mathsf{t}}, \mathsf{C})}{\mathit{backward}(\mathsf{original}(\overline{\mathsf{t}}), \mathsf{C})}$$

$$\frac{\Gamma \vdash \mathsf{t} : \mathsf{D} \quad RT(\mathsf{D}, \mathsf{f}) = \mathsf{R} \quad \mathit{backward}(\mathsf{R}, \mathsf{C}) \quad \mathit{backward}(\mathsf{t}, \mathsf{C})}{\mathit{backward}(\mathsf{t.f}, \mathsf{C})}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{t} : \mathsf{D} \quad RT(\mathsf{D}, \mathsf{m}) = \mathsf{R} \\ \mathit{backward}(\mathsf{R}, \mathsf{C}) \quad \mathit{backward}(\mathsf{t}, \mathsf{C}) \quad \mathit{backward}(\overline{\mathsf{t}}, \mathsf{C})\end{array}}{\mathit{backward}(\mathsf{t.m}(\overline{\mathsf{t}}), \mathsf{C})}$$

$$\frac{\mathit{backward}(\mathsf{D}, \mathsf{C}) \quad \mathit{backward}(\overline{\mathsf{t}}, \mathsf{C})}{\mathit{backward}(\mathsf{new\ D}(\overline{\mathsf{t}}), \mathsf{C})} \qquad \frac{\mathit{backward}(\mathsf{D}, \mathsf{C}) \quad \mathit{backward}(\mathsf{t}, \mathsf{C})}{\mathit{backward}((\mathsf{D})\mathsf{t}, \mathsf{C})}$$

Note that in two of the above rules the refinement table RT is used to determine the refinement that adds a field or method to a class. For that, the definition of RT is extended: $RT(\mathsf{D}, \mathsf{f})$ determines the refinement that adds the field f to class D; $RT(\mathsf{D}, \mathsf{m})$ determines the refinement that adds the method f to class D.

B Type Soundness

B.1 Type Soundness Proof of Basic FFJ

Theorem B1 (Preservation): If $\Gamma \vdash \mathsf{t} : \mathsf{C}$ and $\mathsf{t} \longrightarrow \mathsf{t}'$, then $\Gamma \vdash \mathsf{t}'$: C' for some $\mathsf{C}' <: \mathsf{C}$.

Before giving the main proof, we develop some required lemmas.

Lemma B1: If $mtype(\mathsf{m}, \mathsf{D}) = \overline{\mathsf{C}} \rightarrow \mathsf{C}_0$, then $mtype(\mathsf{m}, \mathsf{C}) = \overline{\mathsf{C}} \rightarrow \mathsf{C}_0$ for all $\mathsf{C} <: \mathsf{D}$.

Proof. Straightforward induction on the derivation of $\mathsf{C} <: \mathsf{D}$. Note that, whether m is defined in $CT(\mathsf{C})$ or not, $mtype(\mathsf{m}, \mathsf{C})$ should be the same as $mtype(\mathsf{m}, \mathsf{E})$ where either $CT(\mathsf{C}) = \mathsf{class\ C\ extends\ E}\ \{\ \ldots\}$ or $succ(\mathsf{C}) = \mathsf{E}$. That is, overriding or refining a method with an refinement preserves the type of the method. □

Lemma B2 (Term substitution preserves typing): If $\Gamma, \overline{\mathsf{x}} : \overline{\mathsf{B}} \vdash \mathsf{t} : \mathsf{D}$ and $\Gamma, \overline{\mathsf{s}} : \overline{\mathsf{A}}$, where $\overline{\mathsf{A}} <: \overline{\mathsf{B}}$, then $\Gamma \vdash [\overline{\mathsf{x}} \mapsto \overline{\mathsf{s}}]\,\mathsf{t} : \mathsf{C}$ for some $\mathsf{C} <: \mathsf{D}$.

Proof. By induction on the derivation of $\Gamma, \overline{\mathsf{x}} : \overline{\mathsf{B}} \vdash \mathsf{t} : \mathsf{D}$.

Case (T-Var): $\mathsf{t} = \mathsf{x} \quad \mathsf{x} : \mathsf{D} \in \Gamma$

If $\mathsf{x} \notin \overline{\mathsf{x}}$, then the result is trivial since $[\overline{\mathsf{x}} \mapsto \overline{\mathsf{s}}]\,\mathsf{x} = \mathsf{x}$. On the other hand, if $\mathsf{x} = \mathsf{x}_i$ and $\mathsf{D} = \mathsf{B}_i$, then, since $[\overline{\mathsf{x}} \mapsto \overline{\mathsf{s}}]\,\mathsf{x} = \mathsf{s}_i$, letting $\mathsf{C} = \mathsf{A}_i$ finishes the case.

Case (T-Field): $\mathsf{t} = \mathsf{t}_0.\mathsf{f}_i \quad \Gamma, \overline{\mathsf{x}} : \overline{\mathsf{B}} \vdash \mathsf{t}_0 : \mathsf{D}_0 \quad fields(\mathsf{D}_0) = \overline{\mathsf{C}}\ \overline{\mathsf{f}} \quad \mathsf{D} = \mathsf{C}_i$

By the induction hypothesis, there is some C_0 such that $\Gamma \vdash [\overline{\mathsf{x}} \mapsto \overline{\mathsf{s}}]\,\mathsf{t}_0 : \mathsf{C}_0$ and $\mathsf{C}_0 <: \mathsf{D}_0$. It is easy to check that $fields(\mathsf{C}_0) = (fields(\mathsf{D}_0), \overline{\mathsf{D}}\ \overline{\mathsf{g}})$ for some $\overline{\mathsf{D}}\ \overline{\mathsf{g}}$. Therefore, by T-Field, $\Gamma \vdash ([\overline{\mathsf{x}} \mapsto \overline{\mathsf{s}}]\,\mathsf{t}_0).\mathsf{f}_i : \mathsf{C}_i$. The fact that a class' refinements can add new fields does not affect this case. $\overline{\mathsf{D}}\ \overline{\mathsf{g}}$ contains the fields that C_0 adds and the fields that the refinements of C_0 add.

Case (T-Invk): $\mathsf{t} = \mathsf{t}_0.\mathsf{m}(\overline{\mathsf{t}}) \quad \Gamma, \overline{\mathsf{x}} : \overline{\mathsf{B}} \vdash \mathsf{t}_0 : \mathsf{D}_0 \quad mtype(\mathsf{m}, \mathsf{D}_0) = \overline{\mathsf{E}} \rightarrow \mathsf{D}$
$\Gamma, \overline{\mathsf{x}} : \overline{\mathsf{B}} \vdash \overline{\mathsf{t}} : \overline{\mathsf{D}} \quad \overline{\mathsf{D}} <: \overline{\mathsf{E}}$

By the induction hypothesis, there are some C_0 and $\overline{\mathsf{C}}$ such that:

$$\Gamma \vdash [\overline{\mathsf{x}} \mapsto \overline{\mathsf{s}}]\,\mathsf{t}_0 : \mathsf{C}_0 \quad \mathsf{C}_0 <: \mathsf{D}_0 \quad \Gamma \vdash [\overline{\mathsf{x}} \mapsto \overline{\mathsf{s}}]\,\overline{\mathsf{t}} : \overline{\mathsf{C}} \quad \overline{\mathsf{C}} <: \overline{\mathsf{D}}.$$

By Lemma B1, it follows $mtype(\mathsf{m}, \mathsf{C}_0) = \overline{\mathsf{E}} \rightarrow \mathsf{D}$. Moreover, $\overline{\mathsf{C}} <: \overline{\mathsf{E}}$ by the transitivity of <:. Therefore, by T-Invk, $\Gamma \vdash [\overline{\mathsf{x}} \mapsto \overline{\mathsf{s}}]\,\mathsf{t}_0.\mathsf{m}([\overline{\mathsf{x}} \mapsto \overline{\mathsf{s}}]\ \overline{\mathsf{t}}) : \mathsf{D}$. Since a refinement can override a method but not change the type (no overloading), this case does not change with FFJ.

Case (T-New): $\mathsf{t} = \mathsf{new\ D}(\overline{\mathsf{t}}) \quad fields(\mathsf{D}) = \overline{\mathsf{D}}\ \overline{\mathsf{f}} \quad \Gamma, \overline{\mathsf{x}} : \overline{\mathsf{B}} \vdash \overline{\mathsf{t}} : \overline{\mathsf{C}} \quad \overline{\mathsf{C}} <: \overline{\mathsf{D}}$

By the induction hypothesis, $\Gamma \vdash [\bar{\mathsf{x}} \mapsto \bar{\mathsf{s}}]\,\bar{\mathsf{t}} : \bar{\mathsf{E}}$ for some $\bar{\mathsf{E}}$ with $\bar{\mathsf{E}} <: \bar{\mathsf{C}}$. We have $\bar{\mathsf{E}} <: \bar{\mathsf{D}}$ by the transitivity of <:. Therefore, by the rule T-New, $\Gamma \vdash \mathsf{new\ D}([\bar{\mathsf{x}} \mapsto \bar{\mathsf{s}}]\ \bar{\mathsf{t}}) : \mathsf{D}$. Since each refinement must extend the constructor to initialize the fields it adds, this case remains unchanged. That is, the number of arguments ($\bar{\mathsf{t}}$) equals the number of fields ($\bar{\mathsf{f}}$) the function *fields* returns.

Case (T-UCast): $\mathsf{t} = (\mathsf{D})\mathsf{t}_0 \quad \Gamma, \bar{\mathsf{x}} : \bar{\mathsf{B}} \vdash \mathsf{t}_0 : \mathsf{C} \quad \mathsf{C} <: \mathsf{D}$

By the induction hypothesis, there is some E such that $\Gamma \vdash [\bar{\mathsf{x}} \mapsto \bar{\mathsf{s}}]\,\mathsf{t}_0 : \mathsf{E}$ and $\mathsf{E} <: \mathsf{C}$. We have $\mathsf{E} <: \mathsf{D}$ by the transitivity of the subtype relation <:, which yields $\Gamma \vdash (\mathsf{D})(\,[\bar{\mathsf{x}} \mapsto \bar{\mathsf{s}}]\,\mathsf{t}_0) : \mathsf{D}$ by T-UCast.

Case (T-DCast): $\mathsf{t} = (\mathsf{D})\mathsf{t}_0 \quad \Gamma, \bar{\mathsf{x}} : \bar{\mathsf{B}} \vdash \mathsf{t}_0 : \mathsf{C} \quad \mathsf{D} <: \mathsf{C} \quad \mathsf{D} \neq \mathsf{C}$

By the induction hypothesis, there is some E such that $\Gamma \vdash [\bar{\mathsf{x}} \mapsto \bar{\mathsf{s}}]\,\mathsf{t}_0 : \mathsf{E}$ and $\mathsf{E} <: \mathsf{C}$. If $\mathsf{E} <: \mathsf{D}$ or $\mathsf{D} <: \mathsf{E}$, then $\Gamma \vdash (\mathsf{D})([\bar{\mathsf{x}} \mapsto \bar{\mathsf{s}}]\ \mathsf{t}_0) : \mathsf{D}$ by T-UCast or T-DCast, respectively. If both $\mathsf{D} \not<: \mathsf{E}$ and $\mathsf{E} \not<: \mathsf{D}$, then $\Gamma \vdash (\mathsf{D})([\bar{\mathsf{x}} \mapsto \bar{\mathsf{s}}]\ \mathsf{t}_0) : \mathsf{D}$ (with a *stupid warning*) by T-SCast.

Case (T-SCast): $\mathsf{t} = (\mathsf{D})\mathsf{t}_0 \quad \Gamma, \bar{\mathsf{x}} : \bar{\mathsf{B}} \vdash \mathsf{t}_0 : \mathsf{C} \quad \mathsf{D} \not<: \mathsf{C} \quad \mathsf{C} \not<: \mathsf{D}$

By the induction hypothesis, there is some E such that $\Gamma \vdash [\bar{\mathsf{x}} \mapsto \bar{\mathsf{s}}]\,\mathsf{t}_0 : \mathsf{E}$ and $\mathsf{E} <: \mathsf{C}$. This means that $\mathsf{E} \not<: \mathsf{D}$ (in basic FFJ, each class has just one superclass. It follows that, if both $\mathsf{E} <: \mathsf{C}$ and $\mathsf{E} <: \mathsf{D}$, then either $\mathsf{C} <: \mathsf{D}$ or $\mathsf{D} <: \mathsf{C}$) So $\Gamma \vdash (\mathsf{D})(\,[\bar{\mathsf{x}} \mapsto \bar{\mathsf{s}}]\,\mathsf{t}_0) : \mathsf{D}$ (with a *stupid warning*), by T-SCast. □

Lemma B3 (Weakening): If $\Gamma \vdash \mathsf{t} : \mathsf{C}$, then $\Gamma, \mathsf{x} : \mathsf{D} \vdash \mathsf{t} : \mathsf{C}$

Proof. Straightforward induction. Nothing changes in FFJ compared to FJ. □

Lemma B4: If $\mathit{mtype}(\mathsf{m}, \mathsf{C}_0) = \bar{\mathsf{D}} \to \mathsf{D}$, and $\mathit{mbody}(\mathsf{m}, \mathsf{C}_0) = (\bar{\mathsf{x}}, \mathsf{t})$, then for some D_0 and some $\mathsf{C} <: \mathsf{D}$ we have $\mathsf{C}_0 <: \mathsf{D}_0$ and $\bar{\mathsf{x}} : \bar{\mathsf{D}}, \mathsf{this} : \mathsf{D}_0 \vdash \mathsf{t} : \mathsf{C}$.

Proof. By induction on the derivation of $\mathit{mbody}(\mathsf{m}, \mathsf{C}_0)$. The base case (in which m is defined in C_0) is easy since m is defined in $CT(\mathsf{C}_0)$ and the well-formedness of the class table implies that we must have derived $\bar{\mathsf{x}} : \bar{\mathsf{D}}, \mathsf{this} : \mathsf{C}_0 \vdash \mathsf{t} : \mathsf{C}$ by the well-formedness rules of method declarations and refinements. The induction step is also straightforward. This lemma holds for FFJ since a method refinement does not change the argument and result types of a method and this points always to the class that is refined. □

Proof of Theorem B1 (Preservation). By induction on a derivation of $\mathsf{t} \longrightarrow \mathsf{t}'$, with a case analysis on the final rule.

Case (E-ProjNew): $\mathsf{t} = \mathsf{new\ C}_0(\bar{\mathsf{v}}).\mathsf{f}_i \quad \mathsf{t}' = \mathsf{v}_i \quad \mathit{fields}(\mathsf{C}_0) = \bar{\mathsf{D}}\ \bar{\mathsf{f}}$

From the shape of t, we see that the final rule in the derivation of $\Gamma \vdash \mathsf{t} : \mathsf{C}$ must be T-Field, with premise $\Gamma \vdash \mathsf{new\ C}_0(\bar{\mathsf{v}}) : \mathsf{D}_0$, for some D_0, and that $\mathsf{C} = \mathsf{D}_i$. Similarly, the last rule in the derivation of $\Gamma \vdash \mathsf{new\ C}_0(\bar{\mathsf{v}}) : \mathsf{D}_0$ must be T-New, with premises $\Gamma \vdash \bar{\mathsf{v}} : \bar{\mathsf{C}}$ and $\bar{\mathsf{C}} <: \bar{\mathsf{D}}$, and with $\mathsf{D}_0 = \mathsf{C}_0$. In particular, $\Gamma \vdash \mathsf{v}_i : \mathsf{C}_i$, which finishes the case, since $\mathsf{C}_i <: \mathsf{D}_i$.

Case (E-InvkNew): $\mathsf{t} = (\mathsf{new\ C}_0(\bar{\mathsf{v}})).\mathsf{m}(\bar{\mathsf{u}}) \quad \mathsf{t}' = [\bar{\mathsf{x}} \mapsto \bar{\mathsf{u}}, \mathsf{this} \mapsto \mathsf{new\ C}_0(\bar{\mathsf{v}})]\,\mathsf{t}_0$
$\mathit{mbody}(\mathsf{m}, \mathsf{C}_0) = (\bar{\mathsf{x}}, \mathsf{t}_0)$

The final rule in the derivation of $\Gamma \vdash \mathsf{t} : \mathsf{C}$ must be T-INVK and T-NEW, with premises $\Gamma \vdash \mathsf{new}\ \mathsf{C}_0(\overline{\mathsf{v}}) : \mathsf{C}_0$, $\Gamma \vdash \overline{\mathsf{u}} : \overline{\mathsf{C}}$, $\overline{\mathsf{C}} <: \overline{\mathsf{D}}$, and $mtype(\mathsf{m}, \mathsf{C}_0) = \overline{\mathsf{D}} \to \mathsf{C}$. By Lemma B4, we have $\overline{\mathsf{x}} : \overline{\mathsf{D}}, \mathsf{this} : \mathsf{D}_0 \vdash \mathsf{t} : \mathsf{B}$ for some D_0 and B, with $\mathsf{C}_0 <: \mathsf{D}_0$ and $\mathsf{B} <: \mathsf{C}$. By Lemma B3, $\Gamma, \overline{\mathsf{x}} : \overline{\mathsf{D}}, \mathsf{this} : \mathsf{D}_0 \vdash \mathsf{t}_0 : \mathsf{B}$. Then, by Lemma B2, $\Gamma\, [\overline{\mathsf{x}} \mapsto \overline{\mathsf{u}}, \mathsf{this} \mapsto \mathsf{new}\ \mathsf{C}_0(\overline{\mathsf{v}})]\, \mathsf{t}_0 : \mathsf{E}$ for some $\mathsf{E} <: \mathsf{B}$. By the transitivity of $<:$, we obtain $\mathsf{E} <: \mathsf{C}$. Letting $\mathsf{C}' = \mathsf{E}$ completes the case.

CASE (E-CASTNEW): $\mathsf{t} = (\mathsf{D})(\mathsf{new}\ \mathsf{C}_0(\overline{\mathsf{v}})) \quad \mathsf{C}_0 <: \mathsf{D} \quad \mathsf{t}' = \mathsf{new}\ \mathsf{C}_0(\overline{\mathsf{v}})$

The proof of $\Gamma \vdash (\mathsf{D})(\mathsf{new}\ \mathsf{C}_0(\overline{\mathsf{v}})) : \mathsf{C}$ must end with T-UCAST since ending with T-SCAST or T-DCAST would contradict the assumption $\mathsf{C}_0 <: \mathsf{D}$. The premises of T-UCAST, give us $\Gamma \vdash \mathsf{new}\ \mathsf{C}_0(\overline{\mathsf{v}}) : \mathsf{C}_0$ and $\mathsf{D} = \mathsf{C}$, finishing the case.

The cases for the congruence rules are easy. We show just the case E-CAST.

CASE (E-CAST): $\mathsf{t} = (\mathsf{D})\mathsf{t}_0 \quad \mathsf{t}' = (\mathsf{D})\mathsf{t}_0' \quad \mathsf{t}_0 \longrightarrow \mathsf{t}_0'$

There are three subcases according to the last typing rule used.

SUBCASE (T-UCAST): $\Gamma \vdash \mathsf{t}_0 : \mathsf{C}_0 \quad \mathsf{C}_0 <: \mathsf{D} \quad \mathsf{D} = \mathsf{C}$

By the induction hypothesis, $\Gamma \vdash \mathsf{t}_0' : \mathsf{C}_0'$ for some $\mathsf{C}_0' <: \mathsf{C}_0$. By transitivity of $<:$, $\mathsf{C}_0' <: \mathsf{C}$. Therefore, by T-UCAST $\Gamma \vdash (\mathsf{C})\mathsf{t}_0' : \mathsf{C}$ (with no additional *stupid warning*).

SUBCASE (T-DCAST): $\Gamma \vdash \mathsf{t}_0 : \mathsf{C}_0 \quad \mathsf{D} <: \mathsf{C}_0 \quad \mathsf{D} = \mathsf{C}$

By the induction hypothesis, $\Gamma \vdash \mathsf{t}_0' : \mathsf{C}_0'$ for some $\mathsf{C}_0' <: \mathsf{C}_0$. If $\mathsf{C}_0' <: \mathsf{C}$ or $\mathsf{C} <: \mathsf{C}_0'$, then $\Gamma \vdash (\mathsf{C})\mathsf{t}_0' : \mathsf{C}$ by T-UCAST or T-DCAST (without any additional *stupid warning*). On the other hand, if both $\mathsf{C}_0' \not<: \mathsf{C}$ or $\mathsf{C} \not<: \mathsf{C}_0'$, then, $\Gamma \vdash (\mathsf{C})\mathsf{t}_0' : \mathsf{C}$ with a *stupid warning* by T-SCAST.

SUBCASE (T-SCAST): $\Gamma \vdash \mathsf{t}_0 : \mathsf{C}_0 \quad \mathsf{D} \not<: \mathsf{C}_0 \quad \mathsf{C}_0 \not<: \mathsf{D} \quad \mathsf{D} = \mathsf{C}$

By the induction hypothesis, $\Gamma \vdash \mathsf{t}_0' : \mathsf{C}_0'$ for some $\mathsf{C}_0' <: \mathsf{C}_0$. Then, both $\mathsf{C}_0' \not<: \mathsf{C}$ and $\mathsf{C} \not<: \mathsf{C}_0'$ also hold. Therefore $\Gamma \vdash (\mathsf{C})\mathsf{t}_0' : \mathsf{C}$ with a *stupid warning.*

If $\mathsf{C}_0' \not<: \mathsf{C}$, then $\mathsf{C} \not<: \mathsf{C}_0'$ since $\mathsf{C} \not<: \mathsf{C}_0$ and, therefore, $\Gamma \vdash (\mathsf{C})\mathsf{t}_0' : \mathsf{C}$ with *stupid warning.* If $\mathsf{C}_0' <: \mathsf{C}$, then $\Gamma \vdash (\mathsf{C})\mathsf{t}_0' : \mathsf{C}$ by T-UCAST (with no additional *stupid warning*). This subcase is analogous to the case T-SCAST of the proof of Lemma B2.

□

THEOREM B2 (Progress): Suppose t is a well-typed term.

1. If t includes $\mathsf{new}\ \mathsf{C}_0(\overline{\mathsf{t}}).\mathsf{f}_i$ as a subterm, then $fields(\mathsf{C}_0) = \overline{\mathsf{C}}\ \overline{\mathsf{f}}$ for some $\overline{\mathsf{C}}$ and $\overline{\mathsf{f}}$.
2. If t includes $\mathsf{new}\ \mathsf{C}_0(\overline{\mathsf{t}}).\mathsf{m}(\overline{\mathsf{u}})$ as a subterm, then $mbody(\mathsf{m}, \mathsf{C}_0) = (\overline{\mathsf{x}}, \mathsf{t}_0)$ and $|\overline{\mathsf{x}}| = |\overline{\mathsf{u}}|$ for some $\overline{\mathsf{x}}$ and t_0.

Proof. If t has $\mathsf{new}\ \mathsf{C}_0(\overline{\mathsf{t}}).\mathsf{f}_i$ as a subterm, then, by well-typedness of the subterm, it is easy to check that $fields(\mathsf{C}_0)$ is well-defined and f_i appears in it. The fact that class refinements may add fields (that have not been defined already) does not change this conclusion. Similarly, if t has $\mathsf{new}\ \mathsf{C}_0(\overline{\mathsf{t}}).\mathsf{m}(\overline{\mathsf{u}})$ as a subterm, then it is also easy to show that $mbody(\mathsf{m}, \mathsf{C}_0) = (\overline{\mathsf{x}}, \mathsf{t}_0)$ and $|\overline{\mathsf{x}}| = |\overline{\mathsf{u}}|$ from the fact that $mtype(\mathsf{m}, \mathsf{C}_0) = \overline{\mathsf{C}} \to \mathsf{D}$ where $|\overline{\mathsf{x}}| = |\overline{\mathsf{C}}|$. This conclusion holds for FFJ since a method refinement must have the same signature than the method refined. □

THEOREM B3 (FFJ Type Soundness): If $\emptyset \vdash \mathsf{t} : \mathsf{C}$ and $\mathsf{t} \longrightarrow^* \mathsf{t}'$ with t' a normal form, then t' is either a value v with $\emptyset \vdash \mathsf{v} : \mathsf{D}$ and $\mathsf{D} <: \mathsf{C}$, or a term containing $(\mathsf{D})(\mathsf{new}\ \mathsf{C}(\overline{\mathsf{t}}))$ in which $\mathsf{C} <: \mathsf{D}$.

Proof. Immediate from Theorem B1 and B2. Nothing changes in the proof of Theorem B3 for FFJ compared to FJ. □

B.2 Type Soundness Proof of FFJ$_{ME}$

The case E-INVKNEW of the proof of Theorem B1 (Preservation) is the only that might be affected by the extension of FFJ$_{ME}$ since in its assumption it contains a method body, which now may contain an original. However, E-INVKNEW2, the new evaluation rule of FFJ$_{ME}$, substitutes each occurrence of original. This evaluation process results in a method body that is indistinguishable from a common FFJ (or FJ) method body. Thus, the proof proceeds with the assumptions of FFJ. That is, FFJ$_{ME}$ is type-sound.

B.3 Type Soundness Proof of FFJ$_{DV}$

Compared to basic FFJ, the case T-NEW of the proof of Lemma B2 changes as follows:

CASE (T-NEW): $\mathsf{t} = \mathsf{new}\ \mathsf{D}(\overline{\mathsf{t}})$ $\quad$ $fields(\mathsf{D}) = \overline{\mathsf{D}}\ \overline{\mathsf{f}}, \overline{\mathsf{H}}\ \overline{\mathsf{h}}$ $\quad$ $\Gamma, \overline{\mathsf{x}} : \overline{\mathsf{B}} \vdash \overline{\mathsf{t}} : \overline{\mathsf{C}}$ $\quad$ $\overline{\mathsf{C}} <: \overline{\mathsf{D}}$

By the induction hypothesis, $\Gamma \vdash [\overline{\mathsf{x}} \mapsto \overline{\mathsf{s}}]\,\overline{\mathsf{t}} : \overline{\mathsf{E}}$ for some $\overline{\mathsf{E}}$ with $\overline{\mathsf{E}} <: \overline{\mathsf{C}}$. We have $\overline{\mathsf{E}} <: \overline{\mathsf{D}}$ by the transitivity of <:. Therefore, by the FFJ$_{DV}$ version of T-NEW, $\Gamma \vdash \mathsf{new}\ \mathsf{D}([\overline{\mathsf{x}} \mapsto \overline{\mathsf{s}}]\ \overline{\mathsf{t}}) : \mathsf{D}$. The key is that only for a subset of fields ($\overline{\mathsf{D}}\ \overline{\mathsf{f}}$) values have to be provided, instead of values for all fields ($\overline{\mathsf{D}}\ \overline{\mathsf{f}}, \overline{\mathsf{H}}\ \overline{\mathsf{h}}$).

Furthermore, the case E-PROJNEW of the proof of Theorem B1 (Preservation) changes. Since we have now two cases for projection (E-PROJNEW1 and E-PROJNEW2), we need two consider both in the new proof:

CASE (E-PROJNEW1): $\mathsf{t} = \mathsf{new}\ \mathsf{C}_0(\overline{\mathsf{v}}).\mathsf{f}_i$ $\quad$ $\mathsf{t}' = \mathsf{v}_i$ $\quad$ $fields(\mathsf{C}_0) = \overline{\mathsf{D}}\ \overline{\mathsf{f}}, \overline{\mathsf{E}}\ \overline{\mathsf{h}}$
$|\overline{\mathsf{v}}| = |\overline{\mathsf{f}}|$ $\quad$ $|\overline{\mathsf{C}}| = |\overline{\mathsf{D}}|$

With default values, the number of arguments $\overline{\mathsf{v}}$ that are supplied during the instantiation of the class can be lesser than the number of fields $\overline{\mathsf{D}}\ \overline{\mathsf{f}}, \overline{\mathsf{E}}\ \overline{\mathsf{h}}$ of the class. In this case, a value v_i is supplied for the field f_i that is projected. The remaining proof is similar to FFJ without default values.

CASE (E-PROJNEW2): $\mathsf{t} = \mathsf{new}\ \mathsf{C}_0(\overline{\mathsf{v}}).\mathsf{h}_i$ $\quad$ $\mathsf{t}' = default(\mathsf{E}_i)$ $\quad$ $fields(\mathsf{C}_0) = \overline{\mathsf{D}}\,\overline{\mathsf{f}}, \overline{\mathsf{E}}\,\overline{\mathsf{h}}$
$|\overline{\mathsf{v}}| = |\overline{\mathsf{f}}|$ $\quad$ $|\overline{\mathsf{C}}| = |\overline{\mathsf{D}}|$

In this case, no value is supplied for the field h_i that is projected. Therefore, by rule E-PROJNEW2, a default value $default(\mathsf{E}_i)$ is supplied. By the typing rule T-DEFAULT $\Gamma \vdash default(\mathsf{E}_i) : E_i$, which finishes the case, since $\mathsf{E}_i <: \mathsf{E}_i$.

Including the above changes, FFJ$_{DV}$ is type-sound.

B.4 Type Soundness Proof of FFJ$_{SD}$

Compared to basic FFJ, the only cases that differ are for stupid casts since now classes may inherit from multipe superclasses. Case T-SCAST of the proof of Lemma B2 changes as follows:

CASE (T-SCAST): $\mathsf{t} = (\mathsf{D})\mathsf{t}_0 \quad \Gamma, \bar{\mathsf{x}} : \bar{\mathsf{B}} \vdash \mathsf{t}_0 : \mathsf{C} \quad \mathsf{D} \not<: \mathsf{C} \quad \mathsf{C} \not<: \mathsf{D}$

By the induction hypothesis, there is some E such that $\Gamma \vdash [\bar{\mathsf{x}} \mapsto \bar{\mathsf{s}}]\,\mathsf{t}_0 : \mathsf{E}$ and $\mathsf{E} <: \mathsf{C}$. If $\mathsf{E} \not<: \mathsf{D}$, by the transitivity of $<:$, $\mathsf{D} \not<: \mathsf{E}$ since $\mathsf{D} \not<: \mathsf{C}$ and $\Gamma \vdash (\mathsf{D})([\bar{\mathsf{x}} \mapsto \bar{\mathsf{s}}]\ \mathsf{t}_0) : \mathsf{D}$ (with a *stupid warning*) by T-SCAST. On the other hand, if $\mathsf{E} <: \mathsf{D}$, then $\Gamma \vdash (\mathsf{D})([\bar{\mathsf{x}} \mapsto \bar{\mathsf{s}}]\ \mathsf{t}_0) : \mathsf{D}$ by T-UCAST. This case is different from FJ since E can be a subclass of C and D with $\mathsf{D} \not<: \mathsf{C}$ and $\mathsf{C} \not<: \mathsf{D}$. In FFJ, there would be either $\mathsf{D} <: \mathsf{C}$ or $\mathsf{C} <: \mathsf{D}$. Thus, in FFJ, if $\mathsf{E} <: \mathsf{D}$ T-UCAST finishes the case; otherwise, like in FJ, T-SCAST finishes the case.

Subcase T-SCAST of the proof of Theorem B1 (Preservation) changes as follows:

SUBCASE (T-SCAST): $\Gamma \vdash \mathsf{t}_0 : \mathsf{C}_0 \quad \mathsf{D} \not<: \mathsf{C}_0 \quad \mathsf{C}_0 \not<: \mathsf{D} \quad \mathsf{D} = \mathsf{C}$

By the induction hypothesis, $\Gamma \vdash \mathsf{t}'_0 : \mathsf{C}'_0$ for some $\mathsf{C}'_0 <: \mathsf{C}_0$. If $\mathsf{C}'_0 \not<: \mathsf{C}$, then $\mathsf{C} \not<: \mathsf{C}'_0$ since $\mathsf{C} \not<: \mathsf{C}_0$ and, therefore, $\Gamma \vdash (\mathsf{C})\mathsf{t}'_0 : \mathsf{C}$ with *stupid warning*. If $\mathsf{C}'_0 <: \mathsf{C}$, then $\Gamma \vdash (\mathsf{C})\mathsf{t}'_0 : \mathsf{C}$ by T-UCAST (with no additional *stupid warning*). This subcase is analogous to the case T-SCAST of the proof of Lemma B2.

Including the above changes, FFJ$_{SD}$ is type-sound.

C The Complete FFJ Calculus Including all Extensions

For completeness, we provide in this section the FFJ calculus including all extensions, called full FFJ, along with the type soundness proof.

C.1 Syntax of Full FFJ

In Figure 10, we depict the syntax rules of full FFJ.

C.2 Subtyping and Refinement in Full FFJ

In Figure 11, we define the subtyping and refinement relations of full FFJ.

C.3 Auxiliary Definitions of Full FFJ

In Figures 12, 13, 14, 15, we list the auxiliary definitions used in full FFJ.

C.4 Evaluation of Full FFJ

In Figure 16, we show the evaluation rules of full FFJ.

$\mathsf{CD} ::=$	*class declarations:*
$\mathsf{class\ C\ extends\ C\ \{\ \overline{C}\ \overline{f};\ KD\ \overline{MD}\ \}}$	
$\mathsf{CR} ::=$	*class refinements:*
$\mathsf{refines\ class\ C\ extends\ D\ \{\ \overline{C}\ \overline{f};\ KR\ \overline{MD}\ \overline{MR}\ \}}$	
$\mathsf{KD} ::=$	*constructor declarations:*
$\mathsf{C(\overline{D}\ \overline{g},\ \overline{C}\ \overline{f})\ \{\ super(\overline{g});\ this.\overline{f}{=}\overline{f};\ \}}$	
$\mathsf{KR} ::=$	*constructor refinements:*
$\mathsf{refines\ C(\overline{D}\ \overline{g},\ \overline{E}\ \overline{h},\ \overline{C}\ \overline{f})\ \{\ super(\overline{g});\ original(\overline{h});\ this.\overline{f}{=}\overline{f};\ \}}$	
$\mathsf{MD} ::=$	*method declarations:*
$\mathsf{C\ m(\overline{C}\ \overline{x})\ \{\ return\ t;\ \}}$	
$\mathsf{MR} ::=$	*method refinements:*
$\mathsf{refines\ C\ m(\overline{C}\ \overline{x})\ \{\ return\ t;\ \}}$	
$\mathsf{t} ::=$	*terms:*
x	*variable*
$\mathsf{t.f}$	*field access*
$\mathsf{t.m(\overline{t})}$	*method invocation*
$\mathsf{original(\overline{t})}$	*original invocation*
$\mathsf{new\ C(\overline{t})}$	*object creation*
$\mathsf{(C)\,t}$	*cast*
$\mathsf{v} ::=$	*values:*
$\mathsf{new\ C(\overline{v})}$	*object creation*

Fig. 10. Syntax of FFJ including all extensions.

Navigating the refinement chain

$$\frac{\text{S is the successor of R}}{\mathit{succ}(\mathsf{R}) \;=\; \mathsf{S}} \qquad \frac{\mathit{succ}(\mathsf{S}) \;=\; \mathsf{R}}{\mathit{pred}(\mathsf{R}) \;=\; \mathsf{S}} \qquad \frac{\mathsf{S} \prec: \mathsf{R} \qquad \mathit{succ}(\mathsf{S}) \;=\; \mathsf{Object}}{\mathit{last}(\mathsf{R}) \;=\; \mathsf{S}}$$

Refinement relation $\boxed{\mathsf{S} \prec: \mathsf{R}}$

$$\frac{\mathit{succ}(\mathsf{R}) \;=\; \mathsf{S}}{\mathsf{S} \prec: \mathsf{R}} \qquad \frac{\mathsf{T} \prec: \mathsf{S} \qquad \mathsf{S} \prec: \mathsf{R}}{\mathsf{T} \prec: \mathsf{R}}$$

Subtyping $\boxed{\mathsf{C} <: \mathsf{D}}$

$$\mathsf{C} <: \mathsf{C} \qquad \frac{\mathsf{C} <: \mathsf{D} \qquad \mathsf{D} <: \mathsf{E}}{\mathsf{C} <: \mathsf{E}} \qquad \frac{CT(\mathsf{C}) \;=\; \mathsf{class\ C\ extends\ D\ \{\ \ldots\}}}{\mathsf{C} <: \mathsf{D}}$$

$$\frac{CT(\mathsf{R}) \;=\; \mathsf{refines\ class\ C\ extends\ D\ \{\ \ldots\}} \qquad \mathsf{R} \prec: \mathsf{C}}{\mathsf{C} <: \mathsf{E}}$$

Fig. 11. Subtyping and refinement in FFJ including all extensions.

Field lookup $\boxed{\mathit{fields}(\mathsf{C}) \;=\; \overline{\mathsf{C}}\;\overline{\mathsf{f}}}$

$$\mathit{fields}(\mathsf{Object}) \;=\; \bullet \qquad \frac{CT(\mathsf{C}) = \mathsf{class\ C\ extends\ D\ \{\ \overline{C}\ \overline{f};\ KD\ \overline{MD}\ \}}}{\mathit{fields}(\mathsf{C}) \;=\; \mathit{fields}(\mathsf{D}),\ \overline{\mathsf{C}}\ \overline{\mathsf{f}},\ \mathit{fields}(\mathit{succ}(\mathsf{C}))}$$

$$\frac{CT(\mathsf{R}) \;=\; \mathsf{refines\ class\ C\ extends\ D\ \{\ \overline{C}\ \overline{f};\ KR\ \overline{MD}\ \overline{MR}\ \}}}{\mathit{fields}(\mathsf{C}) \;=\; \mathit{fields}(\mathsf{D}),\ \overline{\mathsf{C}}\ \overline{\mathsf{f}},\ \mathit{fields}(\mathit{succ}(\mathsf{C}))}$$

Reverse field lookup $\boxed{\mathit{rfields}(\mathsf{R}) \;=\; \overline{\mathsf{C}}\;\overline{\mathsf{f}}}$

$$\mathit{rfields}(\mathsf{Object}) \;=\; \bullet \qquad \frac{CT(\mathsf{C}) \;=\; \mathsf{class\ C\ extends\ D\ \{\ \overline{C}\ \overline{f};\ KD\ \overline{MD}\ \}}}{\mathit{rfields}(\mathsf{C}) \;=\; \mathit{fields}(\mathsf{D}),\ \overline{\mathsf{C}}\ \overline{\mathsf{f}}}$$

$$\frac{CT(\mathsf{R}) = \mathsf{refines\ class\ C\ extends\ D\ \{\ \overline{C}\ \overline{f};\ KR\ \overline{MD}\ \overline{MR}\ \}}}{\mathit{rfields}(\mathsf{R}) \;=\; \mathit{rfields}(\mathit{pred}(\mathsf{R})),\ \mathit{fields}(\mathsf{D}),\ \overline{\mathsf{C}}\ \overline{\mathsf{f}}}$$

Method type lookup $\boxed{\mathit{mtype}(\mathsf{m},\mathsf{C}) \;=\; \overline{\mathsf{C}}\rightarrow\mathsf{C}}$

$$\frac{CT(\mathsf{C}) \;=\; \mathsf{class\ C\ extends\ D\ \{\ \overline{C}\ \overline{f};\ KD\ \overline{MD}\ \}} \qquad \mathsf{B\ m(\overline{B}\ \overline{x})\ \{\ return\ t;\ \}} \in \overline{\mathsf{MD}}}{\mathit{mtype}(\mathsf{m},\mathsf{C}) \;=\; \overline{\mathsf{B}}\rightarrow\mathsf{B}}$$

$$\frac{\begin{array}{c} CT(\mathsf{C}) \;=\; \mathsf{class\ C\ extends\ D\ \{\ \overline{C}\ \overline{f};\ KD\ \overline{MD}\ \}} \\ \mathsf{m}\text{ is not defined in }\overline{\mathsf{MD}} \qquad \mathit{mtype}(\mathsf{m},\mathit{succ}(\mathsf{C})) \end{array}}{\mathit{mtype}(\mathsf{m},\mathsf{C}) \;=\; \mathit{mtype}(\mathsf{m},\mathit{succ}(\mathsf{C}))}$$

$$\frac{\begin{array}{c} CT(\mathsf{C}) \;=\; \mathsf{class\ C\ extends\ D\ \{\ \overline{C}\ \overline{f};\ KD\ \overline{MD}\ \}} \\ \mathsf{m}\text{ is not defined in }\overline{\mathsf{MD}} \qquad \neg\mathit{mtype}(\mathsf{m},\mathit{succ}(\mathsf{C})) \end{array}}{\mathit{mtype}(\mathsf{m},\mathsf{C}) \;=\; \mathit{mtype}(\mathsf{m},\mathsf{D})}$$

$$\frac{CT(\mathsf{R}) \;=\; \mathsf{refines\ class\ C\ extends\ D\{\ \overline{C}\ \overline{f};\ KR\ \overline{MD}\ \overline{MR}\ \}} \qquad \mathsf{B\ m(\overline{B}\ \overline{x})\ \{\ return\ t;\ \}} \in \overline{\mathsf{MD}}}{\mathit{mtype}(\mathsf{m},\mathsf{R}) \;=\; \overline{\mathsf{B}}\rightarrow\mathsf{B}}$$

$$\frac{\begin{array}{c} CT(\mathsf{R}) \;=\; \mathsf{refines\ class\ C\ extends\ D\ \{\ \overline{C}\ \overline{f};\ KR\ \overline{MD}\ \overline{MR}\ \}} \\ \mathsf{m}\text{ is not defined in }\overline{\mathsf{MD}} \qquad \neg\mathit{mtype}(\mathsf{m},\mathit{succ}(\mathsf{R})) \end{array}}{\mathit{mtype}(\mathsf{m},\mathsf{R}) \;=\; \mathit{mtype}(\mathsf{m},\mathsf{D})}$$

$$\frac{\begin{array}{c} CT(\mathsf{R}) \;=\; \mathsf{refines\ class\ C\ extends\ D\ \{\ \overline{C}\ \overline{f};\ KR\ \overline{MD}\ \overline{MR}\ \}} \\ \mathsf{m}\text{ is not defined in }\overline{\mathsf{MD}} \qquad \mathit{mtype}(\mathsf{m},\mathit{succ}(\mathsf{R})) \end{array}}{\mathit{mtype}(\mathsf{m},\mathsf{R}) \;=\; \mathit{mtype}(\mathsf{m},\mathit{succ}(\mathsf{R}))}$$

Fig. 12. Auxiliary definitions of FFJ including all extensions.

Reverse method type lookup $\boxed{\mathit{rmtype}(\mathsf{m},\mathsf{C}) = \overline{\mathsf{C}} \to \mathsf{C}}$

$$\frac{CT(\mathsf{C}) = \mathsf{class\ C\ extends\ D\ \{\ \overline{C}\ \overline{f};\ KD\ \overline{MD}\ \}} \qquad \mathsf{B\ m(\overline{B}\ \overline{x})\ \{\ return\ t;\ \}} \in \overline{\mathsf{MD}}}{\mathit{rmtype}(\mathsf{m},\mathsf{C}) = \overline{\mathsf{B}} \to \mathsf{B}}$$

$$\frac{CT(\mathsf{R}) = \mathsf{refines\ class\ C\ extends\ D\ \{\ \overline{C}\ \overline{f};\ KR\ \overline{MD}\ \overline{MR}\ \}} \qquad \mathsf{B\ m(\overline{B}\ \overline{x})\ \{\ return\ t;\ \}} \in \overline{\mathsf{MD}}}{\mathit{rmtype}(\mathsf{m},\mathsf{R}) = \overline{\mathsf{B}} \to \mathsf{B}}$$

$$\frac{CT(\mathsf{R}) = \mathsf{refines\ class\ C\ extends\ D\{\ \overline{C}\ \overline{f};\ KR\ \overline{MD}\ \overline{MR}\ \}} \qquad \mathsf{m}\text{ is not defined in }\overline{\mathsf{MD}}}{\mathit{rmtype}(\mathsf{m},\mathsf{R}) = \mathit{rmtype}(\mathsf{m},\mathit{pred}(\mathsf{R}))}$$

Method body lookup $\boxed{\mathit{mbody}(\mathsf{m},\mathsf{C}) = (\overline{\mathsf{x}},\mathsf{t})}$

$$\frac{\begin{array}{c} CT(\mathsf{C}) = \mathsf{class\ C\ extends\ D\ \{\ \overline{C}\ \overline{f};\ KD\ \overline{MD}\ \}} \\ \mathsf{B\ m(\overline{B}\ \overline{x})\ \{\ return\ t;\ \}} \in \overline{\mathsf{MD}} \qquad \neg\mathit{mbody}(\mathsf{m},\mathit{succ}(\mathsf{C})) \end{array}}{\mathit{mbody}(\mathsf{m},\mathsf{C}) = (\overline{\mathsf{x}},\mathsf{t})}$$

$$\frac{\begin{array}{c} CT(\mathsf{C}) = \mathsf{class\ C\ extends\ D\ \{\ \overline{C}\ \overline{f};\ KD\ \overline{MD}\ \}} \\ \mathsf{m}\text{ is not defined in }\overline{\mathsf{MD}} \qquad \neg\mathit{mbody}(\mathsf{m},\mathit{succ}(\mathsf{C})) \end{array}}{\mathit{mbody}(\mathsf{m},\mathsf{C}) = \mathit{mbody}(\mathsf{m},\mathsf{D})} \qquad \mathit{mbody}(\mathsf{m},\ \mathsf{C}) = \mathit{mbody}(\mathsf{m},\ \mathit{succ}(\mathsf{C}))$$

$$\frac{\begin{array}{c} CT(\mathsf{R}) = \mathsf{refines\ class\ C\ extends\ D\ \{\ \overline{C}\ \overline{f};\ KR\ \overline{MD}\ \overline{MR}\ \}} \qquad \neg\mathit{mbody}(\mathsf{m},\mathit{succ}(\mathsf{R})) \\ \mathsf{B\ m(\overline{B}\ \overline{x})\ \{\ return\ t;\ \}} \in \overline{\mathsf{MD}}\ \text{ or }\ \mathsf{refines\ B\ m(\overline{B}\ \overline{x})\ \{\ return\ t;\ \}} \in \overline{\mathsf{MR}} \end{array}}{\mathit{mbody}(\mathsf{m},\mathsf{R}) = (\overline{\mathsf{x}},\mathsf{t})}$$

$$\frac{\begin{array}{c} CT(\mathsf{R}) = \mathsf{refines\ class\ C\ extends\ D\ \{\ \overline{C}\ \overline{f};\ KR\ \overline{MD}\ \overline{MR}\ \}} \\ \mathsf{m}\text{ is not defined in }\overline{\mathsf{MD}}\text{ or }\overline{\mathsf{MR}} \qquad \neg\mathit{mbody}(\mathsf{m},\mathit{succ}(\mathsf{R})) \end{array}}{\mathit{mbody}(\mathsf{m},\mathsf{R}) = \mathit{mbody}(\mathsf{m},\mathsf{D})}$$

$$\frac{CT(\mathsf{R}) = \mathsf{refines\ class\ C\ extends\ D\ \{\ \overline{C}\ \overline{f};\ KR\ \overline{MD}\ \overline{MR}\ \}}}{\mathit{mbody}(\mathsf{m},\mathsf{R}) = \mathit{mbody}(\mathsf{m},\mathit{succ}(\mathsf{R}))}$$

Valid method overriding $\boxed{\mathit{override}(\mathsf{m},\mathsf{D},\overline{\mathsf{C}} \to \mathsf{C}_0)}$

$$\frac{\mathit{mtype}(\mathsf{m},\mathsf{D}) = \overline{\mathsf{D}} \to \mathsf{D}_0 \text{ implies } \overline{\mathsf{C}} = \overline{\mathsf{D}} \text{ and } \mathsf{C}_0 = \mathsf{D}_0}{\mathit{override}(\mathsf{m},\mathsf{D},\overline{\mathsf{C}} \to \mathsf{C}_0)}$$

Valid method introduction $\boxed{\mathit{introduce}(\mathsf{m},\mathsf{C})}$

$$\frac{\neg\mathit{mtype}(\mathsf{m},\mathit{succ}(\mathsf{C}))}{\mathit{introduce}(\mathsf{m},\mathsf{C})}$$

Valid method refinement $\boxed{\mathit{extend}(\mathsf{m},\mathsf{R},\overline{\mathsf{C}} \to \mathsf{C}_0)}$

$$\frac{\mathit{rmtype}(\mathsf{m},\mathit{pred}(\mathsf{R})) = \overline{\mathsf{B}} \to \mathsf{B}_0 \text{ implies } \overline{\mathsf{C}} = \overline{\mathsf{B}} \text{ and } \mathsf{C}_0 = \mathsf{B}_0}{\mathit{extend}(\mathsf{m},\mathsf{R},\overline{\mathsf{C}} \to \mathsf{C}_0)}$$

Fig. 13. Auxiliary definitions of FFJ including all extensions (continued).

Method refinement evaluation $\boxed{\mathit{eval}(\mathsf{m},\mathsf{R},\mathsf{t})}$

$$\frac{\begin{array}{c} CT(\mathsf{C}) \;=\; \mathsf{class\ C\ extends\ D\ \{\ \ldots\ \overline{MD}\ \}} \\ \mathit{pred}(\mathsf{R}) \;=\; \mathsf{C} \qquad \mathsf{C_0\ m(\overline{C}\ \overline{x})\ \{\ return\ t_0\ \}} \;\in \overline{\mathsf{MD}} \end{array}}{\mathit{eval}(\mathsf{m},\mathsf{R},\mathsf{t}) \;\longrightarrow\; [\mathsf{original}(\overline{\mathsf{y}}) \mapsto [\overline{\mathsf{x}} \mapsto \overline{\mathsf{y}}]\,\mathsf{t_0}]\,\mathsf{t}}$$

$$\frac{\begin{array}{c} CT(\mathsf{S}) \;=\; \mathsf{refines\ class\ C\ extends\ D\ \{\ \ldots\ \overline{MD}\ \}} \\ \mathit{pred}(\mathsf{R}) \;=\; \mathsf{S} \qquad \mathsf{C_0\ m(\overline{C}\ \overline{x})\ \{\ return\ t_0\ \}} \;\in \overline{\mathsf{MD}} \end{array}}{\mathit{eval}(\mathsf{m},\mathsf{R},\mathsf{t}) \;\longrightarrow\; [\mathsf{original}(\overline{\mathsf{y}}) \mapsto [\overline{\mathsf{x}} \mapsto \overline{\mathsf{y}}]\,\mathsf{t_0}]\,\mathsf{t}}$$

$$\frac{\begin{array}{c} CT(\mathsf{S}) \;=\; \mathsf{refines\ class\ C\ extends\ D\ \{\ \ldots\ \overline{MR}\ \}} \\ \mathit{pred}(\mathsf{R}) \;=\; \mathsf{S} \qquad \mathsf{refines\ C_0\ m(\overline{C}\ \overline{x})\ \{\ return\ t_0\ \}} \;\in \overline{\mathsf{MR}} \end{array}}{\mathit{eval}(\mathsf{m},\mathsf{R},\mathsf{t}) \;\longrightarrow\; \mathit{eval}(\mathsf{m},\mathsf{S},[\mathsf{original}(\overline{\mathsf{y}}) \mapsto [\overline{\mathsf{x}} \mapsto \overline{\mathsf{y}}]\,\mathsf{t_0}]\,\mathsf{t})}$$

$$\frac{\begin{array}{c} CT(\mathsf{S}) \;=\; \mathsf{refines\ class\ C\ extends\ D\ \{\ \ldots\ \overline{MD}\ \overline{MR}\ \}} \\ \mathit{pred}(\mathsf{R}) \;=\; \mathsf{S} \qquad \mathsf{m}\text{ is not defined in }\overline{\mathsf{MR}}\text{ or }\overline{\mathsf{MD}} \end{array}}{\mathit{eval}(\mathsf{m},\mathsf{R},\mathsf{t}) \;\longrightarrow\; \mathit{eval}(\mathsf{m},\mathsf{S},\mathsf{t})}$$

Valid superclass declaration $\boxed{\mathit{inherit}(\mathsf{C},\mathsf{R})}$

$$\mathit{inherit}(\mathsf{C},\mathsf{C}) \qquad \frac{\mathit{super}(\mathsf{C}) \cap \mathit{super}(\mathsf{R}) = \{\mathsf{Object}\} \qquad \mathit{inherit}(\mathsf{C},\mathit{pred}(\mathsf{R}))}{\mathit{inherit}(\mathsf{C},\mathsf{R})}$$

Superclass lookup $\boxed{\mathit{super}(\mathsf{C})}$

$$\mathit{super}(\mathsf{Object}) = \emptyset \qquad \frac{CT(C) = \mathsf{class\ C\ extends\ D\ \{\ \ldots\ \}}}{\mathit{super}(\mathsf{C}) = \{\mathsf{D}\} \cup \mathit{super}(\mathsf{D}) \cup \mathit{super}(\mathit{succ}(\mathsf{C}))}$$

$$\frac{CT(R) = \mathsf{refines\ class\ C\ extends\ D\ \{\ \ldots\ \}}}{\mathit{super}(\mathsf{R}) = \{\mathsf{D}\} \cup \mathit{super}(\mathsf{D}) \cup \mathit{super}(\mathit{succ}(\mathsf{R}))}$$

Fig. 14. Auxiliary definitions of FFJ including all extensions (continued).

Backward reference $\boxed{\mathit{backward}(\mathsf{D}, \mathsf{C})}$

$$\frac{\mathsf{D}\text{ has been introduced before }\mathsf{C}}{\mathit{backward}(\mathsf{D}, \mathsf{C})}$$

$$\frac{\mathit{backward}(\mathsf{D}_1, \mathsf{C}) \ldots \mathit{backward}(\mathsf{D}_n, \mathsf{C})}{\mathit{backward}(\overline{\mathsf{D}}, \mathsf{C})} \qquad \frac{\mathit{backward}(\mathsf{t}_1, \mathsf{C}) \ldots \mathit{backward}(\mathsf{t}_n, \mathsf{C})}{\mathit{backward}(\bar{\mathsf{t}}, \mathsf{C})}$$

$$\mathit{backward}(\mathsf{x}, \mathsf{C}) \qquad \frac{\mathit{backward}(\bar{\mathsf{t}}, \mathsf{C})}{\mathit{backward}(\mathsf{original}(\bar{\mathsf{t}}), \mathsf{C})}$$

$$\frac{\mathit{\Gamma} \vdash \mathsf{t} : \mathsf{D} \quad \mathit{RT}(\mathsf{D}, \mathsf{f}) = \mathsf{R} \quad \mathit{backward}(\mathsf{R}, \mathsf{C}) \quad \mathit{backward}(\mathsf{t}, \mathsf{C})}{\mathit{backward}(\mathsf{t.f}, \mathsf{C})}$$

$$\frac{\begin{array}{c}\mathit{\Gamma} \vdash \mathsf{t} : \mathsf{D} \quad \mathit{RT}(\mathsf{D}, \mathsf{m}) = \mathsf{R} \\ \mathit{backward}(\mathsf{R}, \mathsf{C}) \quad \mathit{backward}(\mathsf{t}, \mathsf{C}) \quad \mathit{backward}(\bar{\mathsf{t}}, \mathsf{C})\end{array}}{\mathit{backward}(\mathsf{t.m}(\bar{\mathsf{t}}), \mathsf{C})}$$

$$\frac{\mathit{backward}(\mathsf{D}, \mathsf{C}) \quad \mathit{backward}(\bar{\mathsf{t}}, \mathsf{C})}{\mathit{backward}(\mathsf{new\ D}(\bar{\mathsf{t}}), \mathsf{C})} \qquad \frac{\mathit{backward}(\mathsf{D}, \mathsf{C}) \quad \mathit{backward}(\mathsf{t}, \mathsf{C})}{\mathit{backward}((\mathsf{D})\mathsf{t}, \mathsf{C})}$$

Default value $\boxed{\mathit{default}(\mathsf{C})}$

$$\mathit{default}(\mathsf{Object}) = \mathsf{new\ Object()}$$

$$\frac{\mathit{CT}(\mathsf{C}) = \mathsf{class\ C\ extends\ D}\ \{\ \overline{\mathsf{C}}\ \bar{\mathsf{f}}\ \ldots \}}{\mathit{default}(\mathsf{C}) = \mathsf{new\ C}(\mathit{default}(\mathsf{C}_1), \ldots, \mathit{default}(\mathsf{C}_n))}$$

Fig. 15. Auxiliary definitions of FFJ including all extensions (continued).

$$\frac{\mathit{fields}(\mathsf{C}) \;=\; \overline{\mathsf{C}}\;\overline{\mathsf{f}} \qquad |\overline{\mathsf{v}}| \;\geq\; i}{(\mathsf{new\ C}(\overline{\mathsf{v}})).\mathsf{f}_i \;\longrightarrow\; \mathsf{v}_i} \tag{E-ProjNew1}$$

$$\frac{\mathit{fields}(\mathsf{C}) \;=\; \overline{\mathsf{C}}\;\overline{\mathsf{f}} \qquad |\overline{\mathsf{v}}| \;<\; i}{(\mathsf{new\ C}(\overline{\mathsf{v}})).\mathsf{f}_i \;\longrightarrow\; \mathit{default}(\mathsf{C}_i)} \tag{E-ProjNew2}$$

$$\frac{\mathit{mbody}(\mathsf{m},\mathsf{C}) \;=\; (\overline{\mathsf{x}},\mathsf{t}_0) \qquad \mathit{succ}(\mathsf{C}) \;=\; \mathsf{Object}}{(\mathsf{new\ C}(\overline{\mathsf{v}})).\mathsf{m}(\overline{\mathsf{u}}) \;\longrightarrow\; [\overline{\mathsf{x}} \mapsto \overline{\mathsf{u}}, \mathsf{this} \mapsto \mathsf{new\ C}(\overline{\mathsf{v}})]\,\mathsf{t}_0} \tag{E-ProjInvk1}$$

$$\frac{\mathit{mbody}(\mathsf{m},\mathsf{C}) \;=\; (\overline{\mathsf{x}},\mathsf{t}_0) \qquad \mathit{last}(\mathsf{C}) \;=\; \mathsf{R}}{(\mathsf{new\ C}(\overline{\mathsf{v}})).\mathsf{m}(\overline{\mathsf{u}}) \;\longrightarrow\; [\overline{\mathsf{x}} \mapsto \overline{\mathsf{u}}, \mathsf{this} \mapsto \mathsf{new\ C}(\overline{\mathsf{v}})]\;\mathit{eval}(\mathsf{m},\mathsf{R},\mathsf{t}_0)} \tag{E-ProjInvk2}$$

$$\frac{\mathsf{C} <: \mathsf{D}}{(\mathsf{D})(\mathsf{new\ C}(\overline{\mathsf{v}})) \;\longrightarrow\; \mathsf{new\ C}(\overline{\mathsf{v}})} \tag{E-CastNew}$$

$$\frac{\mathsf{t}_0 \;\longrightarrow\; \mathsf{t}_0'}{\mathsf{t}_0.\mathsf{f} \;\longrightarrow\; \mathsf{t}_0'.\mathsf{f}} \tag{E-Field}$$

$$\frac{\mathsf{t}_0 \;\longrightarrow\; \mathsf{t}_0'}{\mathsf{t}_0.\mathsf{m}(\overline{\mathsf{t}}) \;\longrightarrow\; \mathsf{t}_0'.\mathsf{m}(\overline{\mathsf{t}})} \tag{E-InvkRecv}$$

$$\frac{\mathsf{t}_i \;\longrightarrow\; \mathsf{t}_i'}{\mathsf{v}_0.\mathsf{m}(\overline{\mathsf{v}},\ \mathsf{t}_i,\ \overline{\mathsf{t}}) \;\longrightarrow\; \mathsf{v}_0.\mathsf{m}(\overline{\mathsf{v}},\ \mathsf{t}_i',\ \overline{\mathsf{t}})} \tag{E-InvkArg}$$

$$\frac{\mathsf{t}_i \;\longrightarrow\; \mathsf{t}_i'}{\mathsf{new\ C}(\overline{\mathsf{v}},\ \mathsf{t}_i,\ \overline{\mathsf{t}}) \;\longrightarrow\; \mathsf{new\ C}(\overline{\mathsf{v}},\ \mathsf{t}_i',\ \overline{\mathsf{t}})} \tag{E-NewArg}$$

$$\frac{\mathsf{t}_0 \;\longrightarrow\; \mathsf{t}_0'}{(\mathsf{C})\mathsf{t}_0.\mathsf{f} \;\longrightarrow\; (\mathsf{C})\mathsf{t}_0'.\mathsf{f}} \tag{E-Cast}$$

Fig. 16. Evaluation of FFJ including all extensions.

Term typing $\boxed{\Gamma \vdash \mathsf{t} : \mathsf{C}}$

$$\frac{\mathsf{x} : \mathsf{C} \in \Gamma}{\Gamma \vdash \mathsf{x} : \mathsf{C}} \qquad \text{(T-Var)}$$

$$\frac{\Gamma \vdash \mathsf{t}_0 : \mathsf{C}_0 \qquad \mathit{fields}(\mathsf{C}_0) = \overline{\mathsf{C}}\;\overline{\mathsf{f}}}{\Gamma \vdash \mathsf{t}_0.\mathsf{f}_i : \mathsf{C}_i} \qquad \text{(T-Field)}$$

$$\frac{\Gamma \vdash \mathsf{t}_0 : \mathsf{C}_0 \qquad \mathit{mtype}(\mathsf{m}, \mathsf{C}_0) = \overline{\mathsf{D}} \to \mathsf{C} \qquad \Gamma \vdash \overline{\mathsf{t}} : \overline{\mathsf{C}} \qquad \overline{\mathsf{C}} <: \overline{\mathsf{D}}}{\Gamma \vdash \mathsf{t}_0.\mathsf{m}(\overline{\mathsf{t}}) : \mathsf{C}} \qquad \text{(T-Invk)}$$

$$\frac{\mathit{fields}(\mathsf{C}) = \overline{\mathsf{D}}\;\overline{\mathsf{f}}, \overline{\mathsf{E}}\;\overline{\mathsf{h}} \qquad \Gamma \vdash \overline{\mathsf{t}} : \overline{\mathsf{C}} \qquad \overline{\mathsf{C}} <: \overline{\mathsf{D}}}{\Gamma \vdash \mathsf{new}\ \mathsf{C}(\overline{\mathsf{t}}) : \mathsf{C}} \qquad \text{(T-New)}$$

$$\frac{\Gamma \vdash \mathsf{t}_0 : \mathsf{D} \qquad \mathsf{D} <: \mathsf{C}}{\Gamma \vdash (\mathsf{C})\mathsf{t}_0 : \mathsf{C}} \qquad \text{(T-UCast)}$$

$$\frac{\Gamma \vdash \mathsf{t}_0 : \mathsf{D} \qquad \mathsf{C} <: \mathsf{D} \qquad \mathsf{C} \neq \mathsf{D}}{\Gamma \vdash (\mathsf{C})\mathsf{t}_0 : \mathsf{C}} \qquad \text{(T-DCast)}$$

$$\frac{\Gamma \vdash \mathsf{t}_0 : \mathsf{D} \qquad \mathsf{C} \not<: \mathsf{D} \qquad \mathsf{D} \not<: \mathsf{C} \qquad \mathit{stupid\ warning}}{\Gamma \vdash (\mathsf{C})\mathsf{t}_0 : \mathsf{C}} \qquad \text{(T-SCast)}$$

$$\vdash \mathit{default}(\mathsf{C}) : \mathsf{C} \qquad \text{(T-Default)}$$

Fig. 17. Typing in FFJ including all extensions.

Method typing $\boxed{\mathsf{MD\ OK\ in\ C/R}}$

$$\frac{\begin{array}{c}\overline{\mathsf{x}} : \overline{\mathsf{C}}, \mathsf{this} : \mathsf{C} \vdash \mathsf{t}_0 : \mathsf{E}_0 \qquad \mathsf{E}_0 <: \mathsf{C}_0 \qquad CT(\mathsf{C}) = \mathsf{class\ C\ extends\ D}\ \{\ \ldots\ \} \\ \mathit{override}(\mathsf{m}, \mathsf{D}, \overline{\mathsf{C}} \to \mathsf{C}_0) \qquad \mathit{introduce}(\mathsf{m}, \mathsf{C}) \qquad \mathsf{t}_0 \text{ does not contain } \mathsf{original} \\ \mathit{backward}(\overline{\mathsf{C}}, \mathsf{C}) \qquad \mathit{backward}(\mathsf{C}_0, \mathsf{C}) \qquad \mathit{backward}(\mathsf{t}_0, \mathsf{C})\end{array}}{\mathsf{C}_0\ \mathsf{m}(\overline{\mathsf{C}}\ \overline{\mathsf{x}})\ \{\ \mathsf{return\ t}_0;\ \}\ \mathsf{OK\ in\ C}}$$

$$\frac{\begin{array}{c}\overline{\mathsf{x}} : \overline{\mathsf{C}}, \mathsf{this} : \mathsf{C} \vdash \mathsf{t}_0 : \mathsf{E}_0 \qquad \mathsf{E}_0 <: \mathsf{C}_0 \qquad CT(\mathsf{R}) = \mathsf{refines\ class\ C\ extends\ D}\ \{\ \ldots\ \} \\ \mathit{override}(\mathsf{m}, \mathsf{D}, \overline{\mathsf{C}} \to \mathsf{C}_0) \qquad \mathit{introduce}(\mathsf{m}, \mathsf{R}) \qquad \mathsf{t}_0 \text{ does not contain } \mathsf{original} \\ \mathit{backward}(\overline{\mathsf{C}}, \mathsf{C}) \qquad \mathit{backward}(\mathsf{C}_0, \mathsf{C}) \qquad \mathit{backward}(\mathsf{t}_0, \mathsf{C})\end{array}}{\mathsf{C}_0\ \mathsf{m}(\overline{\mathsf{C}}\ \overline{\mathsf{x}})\ \{\ \mathsf{return\ t}_0;\ \}\ \mathsf{OK\ in\ R}}$$

Method refinement typing $\boxed{\mathsf{MR\ OK\ in\ R}}$

$$\frac{\begin{array}{c}\overline{\mathsf{x}} : \overline{\mathsf{C}}, \mathsf{this} : \mathsf{C} \vdash \mathsf{t}_0 : \mathsf{E}_0 \quad \mathsf{E}_0 <: \mathsf{C}_0 \quad CT(\mathsf{R}) = \mathsf{refines\ class\ C\ extends\ D}\ \{\ \ldots\ \overline{\mathsf{MD}}\ \ldots\ \} \\ \mathsf{m} \text{ not defined in } \overline{\mathsf{MD}} \qquad \mathit{extend}(\mathsf{m}, \mathsf{R}, \overline{\mathsf{C}} \to \mathsf{C}_0) \qquad \mathsf{t}_0 \text{ contains } \mathsf{original} \\ \mathit{backward}(\overline{\mathsf{C}}, \mathsf{R}) \qquad \mathit{backward}(\mathsf{C}_0, \mathsf{R}) \qquad \mathit{backward}(\mathsf{t}_0, \mathsf{R})\end{array}}{\mathsf{refines\ C}_0\ \mathsf{m}(\overline{\mathsf{C}}\ \overline{\mathsf{x}})\ \{\ \mathsf{return\ t}_0;\ \}\ \mathsf{OK\ in\ R}}$$

Class typing $\boxed{\mathsf{C\ OK}}$

$$\frac{\begin{array}{c}\mathsf{KD} = \mathsf{C}(\overline{\mathsf{D}}\ \overline{\mathsf{g}}, \overline{\mathsf{C}}\ \overline{\mathsf{f}})\ \{\ \mathsf{super}(\overline{\mathsf{g}});\ \mathsf{this}.\overline{\mathsf{f}}{=}\overline{\mathsf{f}};\ \} \qquad \mathit{fields}(\mathsf{D}) = \overline{\mathsf{D}}\ \overline{\mathsf{g}} \qquad \overline{\mathsf{MD}}\ \mathsf{OK\ in\ C} \\ \mathit{backward}(\overline{\mathsf{C}}, \mathsf{C}) \qquad \mathit{backward}(\overline{\mathsf{D}}, \mathsf{C}) \qquad \mathit{backward}(\mathsf{D}, \mathsf{C})\end{array}}{\mathsf{class\ C\ extends\ D}\ \{\ \overline{\mathsf{C}}\ \overline{\mathsf{f}};\ \mathsf{KD}\ \overline{\mathsf{MD}}\ \}\ \mathsf{OK}}$$

Class refinement typing $\boxed{\mathsf{R\ OK}}$

$$\frac{\begin{array}{c}\mathsf{KR} = \mathsf{C}(\overline{\mathsf{D}}\ \overline{\mathsf{g}}, \overline{\mathsf{E}}\ \overline{\mathsf{h}}, \overline{\mathsf{C}}\ \overline{\mathsf{f}})\ \{\ \mathsf{super}(\overline{\mathsf{g}});\ \mathsf{original}(\overline{\mathsf{h}});\ \mathsf{this}.\overline{\mathsf{f}}{=}\overline{\mathsf{f}};\ \} \\ \mathit{fields}(\mathsf{D}) = \overline{\mathsf{D}}\ \overline{\mathsf{g}} \qquad \mathit{rfields}(\mathit{pred}(\mathsf{R})) = \overline{\mathsf{E}}\ \overline{\mathsf{h}} \qquad \overline{\mathsf{MD}}\ \mathsf{OK\ in\ R} \qquad \overline{\mathsf{MR}}\ \mathsf{OK\ in\ R} \\ \mathit{inherit}(\mathsf{C}, \mathsf{R}) \qquad \mathsf{R} \prec: \mathsf{C} \qquad CT(\mathsf{C}) = \mathsf{class\ C}\ \ldots \qquad \mathsf{C} \neq \mathsf{Object} \\ RT(\mathsf{refines\ class\ C\ extends\ D}\ \{\ \overline{\mathsf{C}}\ \overline{\mathsf{f}};\ \mathsf{KR}\ \overline{\mathsf{MD}}\ \overline{\mathsf{MD}}\ \}) = \mathsf{R} \\ \mathit{backward}(\overline{\mathsf{C}}, \mathsf{R}) \quad \mathit{backward}(\overline{\mathsf{D}}, \mathsf{R}) \quad \mathit{backward}(\overline{\mathsf{E}}, \mathsf{R}) \quad \mathit{backward}(\mathsf{D}, \mathsf{R})\end{array}}{\mathsf{refines\ class\ C\ extends\ D}\ \{\ \overline{\mathsf{C}}\ \overline{\mathsf{f}};\ \mathsf{KR}\ \overline{\mathsf{MD}}\ \overline{\mathsf{MR}}\ \}\ \mathsf{OK}}$$

Fig. 18. Typing in FFJ including all extensions (continued).

C.5 Typing of Full FFJ

In Figure 16, we show the type rules of full FFJ.

C.6 Type Soundness of Full FFJ

In this section, we provide the type soundness proof of full FFJ.

THEOREM C1 (Preservation): If $\Gamma \vdash \mathsf{t} : \mathsf{C}$ and $\mathsf{t} \longrightarrow \mathsf{t}'$, then $\Gamma \vdash \mathsf{t}' : \mathsf{C}'$ for some $\mathsf{C}' <: \mathsf{C}$.

Before giving the main proof, we develop some required lemmas.

LEMMA C1: If $\mathit{mtype}(\mathsf{m}, \mathsf{D}) = \overline{\mathsf{C}} \rightarrow \mathsf{C}_0$, then $\mathit{mtype}(\mathsf{m}, \mathsf{C}) = \overline{\mathsf{C}} \rightarrow \mathsf{C}_0$ for all $\mathsf{C} <: \mathsf{D}$.

Proof. Straightforward induction on the derivation of $\mathsf{C} <: \mathsf{D}$. Note that, whether m is defined in $CT(\mathsf{C})$ or not, $\mathit{mtype}(\mathsf{m}, \mathsf{C})$ should be the same as $\mathit{mtype}(\mathsf{m}, \mathsf{E})$ where either $CT(\mathsf{C}) = \mathsf{class\ C\ extends\ E}\ \{\ \ldots\}$ or $\mathit{succ}(\mathsf{C}) = \mathsf{E}$. That is, overriding or refining a method with an refinement preserves the type of the method. $\square$

LEMMA C2 (Term substitution preserves typing): If $\Gamma, \overline{\mathsf{x}} : \overline{\mathsf{B}} \vdash \mathsf{t} : \mathsf{D}$ and $\Gamma, \overline{\mathsf{s}} : \overline{\mathsf{A}}$, where $\overline{\mathsf{A}} <: \overline{\mathsf{B}}$, then $\Gamma \vdash [\overline{\mathsf{x}} \mapsto \overline{\mathsf{s}}]\,\mathsf{t} : \mathsf{C}$ for some $\mathsf{C} <: \mathsf{D}$.

Proof. By induction on the derivation of $\Gamma, \overline{\mathsf{x}} : \overline{\mathsf{B}} \vdash \mathsf{t} : \mathsf{D}$.

CASE (T-VAR): $\mathsf{t} = \mathsf{x} \quad \mathsf{x} : \mathsf{D} \in \Gamma$

If $\mathsf{x} \notin \overline{\mathsf{x}}$, then the result is trivial since $[\overline{\mathsf{x}} \mapsto \overline{\mathsf{s}}]\,\mathsf{x} = \mathsf{x}$. On the other hand, if $\mathsf{x} = \mathsf{x}_i$ and $\mathsf{D} = \mathsf{B}_i$, then, since $[\overline{\mathsf{x}} \mapsto \overline{\mathsf{s}}]\,\mathsf{x} = \mathsf{s}_i$, letting $\mathsf{C} = \mathsf{A}_i$ finishes the case.

CASE (T-FIELD): $\mathsf{t} = \mathsf{t}_0.\mathsf{f}_i \quad \Gamma, \overline{\mathsf{x}} : \overline{\mathsf{B}} \vdash \mathsf{t}_0 : \mathsf{D}_0 \quad \mathit{fields}(\mathsf{D}_0) = \overline{\mathsf{C}}\ \overline{\mathsf{f}} \quad \mathsf{D} = \mathsf{C}_i$

By the induction hypothesis, there is some C_0 such that $\Gamma \vdash [\overline{\mathsf{x}} \mapsto \overline{\mathsf{s}}]\,\mathsf{t}_0 : \mathsf{C}_0$ and $\mathsf{C}_0 <: \mathsf{D}_0$. It is easy to check that $\mathit{fields}(\mathsf{C}_0) = (\mathit{fields}(\mathsf{D}_0), \overline{\mathsf{D}}\ \overline{\mathsf{g}})$ for some $\overline{\mathsf{D}}\ \overline{\mathsf{g}}$. Therefore, by T-FIELD, $\Gamma \vdash ([\overline{\mathsf{x}} \mapsto \overline{\mathsf{s}}]\,\mathsf{t}_0).\mathsf{f}_i : \mathsf{C}_i$. The fact that a class' refinements can add new fields does not affect this case. $\overline{\mathsf{D}}\ \overline{\mathsf{g}}$ contains the fields that C_0 adds and the fields that the refinements of C_0 add.

CASE (T-INVK): $\mathsf{t} = \mathsf{t}_0.\mathsf{m}(\overline{\mathsf{t}}) \quad \Gamma, \overline{\mathsf{x}} : \overline{\mathsf{B}} \vdash \mathsf{t}_0 : \mathsf{D}_0 \quad \mathit{mtype}(\mathsf{m}, \mathsf{D}_0) = \overline{\mathsf{E}} \rightarrow \mathsf{D}$
$\Gamma, \overline{\mathsf{x}} : \overline{\mathsf{B}} \vdash \overline{\mathsf{t}} : \overline{\mathsf{D}} \quad \overline{\mathsf{D}} <: \overline{\mathsf{E}}$

By the induction hypothesis, there are some C_0 and $\overline{\mathsf{C}}$ such that:

$$\Gamma \vdash [\overline{\mathsf{x}} \mapsto \overline{\mathsf{s}}]\,\mathsf{t}_0 : \mathsf{C}_0 \quad \mathsf{C}_0 <: \mathsf{D}_0 \quad \Gamma \vdash [\overline{\mathsf{x}} \mapsto \overline{\mathsf{s}}]\,\overline{\mathsf{t}} : \overline{\mathsf{C}} \quad \overline{\mathsf{C}} <: \overline{\mathsf{D}}.$$

By Lemma C1, it follows $\mathit{mtype}(\mathsf{m}, \mathsf{C}_0) = \overline{\mathsf{E}} \rightarrow \mathsf{D}$. Moreover, $\overline{\mathsf{C}} <: \overline{\mathsf{E}}$ by the transitivity of $<:$. Therefore, by T-INVK, $\Gamma \vdash [\overline{\mathsf{x}} \mapsto \overline{\mathsf{s}}]\,\mathsf{t}_0.\mathsf{m}([\overline{\mathsf{x}} \mapsto \overline{\mathsf{s}}]\ \overline{\mathsf{t}}) : \mathsf{D}$. Since a refinement can override a method but not change the type (no overloading), this case does not change with FFJ.

CASE (T-NEW): $\mathsf{t} = \mathsf{new\ D}(\overline{\mathsf{t}}) \quad \mathit{fields}(\mathsf{D}) = \overline{\mathsf{D}}\ \overline{\mathsf{f}}, \overline{\mathsf{H}}\ \overline{\mathsf{h}} \quad \Gamma, \overline{\mathsf{x}} : \overline{\mathsf{B}} \vdash \overline{\mathsf{t}} : \overline{\mathsf{C}} \quad \overline{\mathsf{C}} <: \overline{\mathsf{D}}$

By the induction hypothesis, $\Gamma \vdash [\overline{\mathsf{x}} \mapsto \overline{\mathsf{s}}]\,\overline{\mathsf{t}} : \overline{\mathsf{E}}$ for some $\overline{\mathsf{E}}$ with $\overline{\mathsf{E}} <: \overline{\mathsf{C}}$. We have $\overline{\mathsf{E}} <: \overline{\mathsf{D}}$ by the transitivity of $<:$. Therefore, by the rule T-NEW, $\Gamma \vdash \mathsf{new\ D}([\overline{\mathsf{x}} \mapsto \overline{\mathsf{s}}]\ \overline{\mathsf{t}}) : \mathsf{D}$. The key is that only for a subset of fields $(\overline{\mathsf{D}}\ \overline{\mathsf{f}})$ values have to be provided, instead of values for all fields $(\overline{\mathsf{D}}\ \overline{\mathsf{f}}, \overline{\mathsf{H}}\ \overline{\mathsf{h}})$.

Case (T-UCast): $\mathsf{t} = (\mathsf{D})\mathsf{t}_0 \quad \Gamma, \overline{\mathsf{x}} : \overline{\mathsf{B}} \vdash \mathsf{t}_0 : \mathsf{C} \quad \mathsf{C} <: \mathsf{D}$

By the induction hypothesis, there is some E such that $\Gamma \vdash [\overline{\mathsf{x}} \mapsto \overline{\mathsf{s}}]\,\mathsf{t}_0 : \mathsf{E}$ and $\mathsf{E} <: \mathsf{C}$. We have $\mathsf{E} <: \mathsf{D}$ by the transitivity of the subtype relation $<:$, which yields $\Gamma \vdash (\mathsf{D})(\,[\overline{\mathsf{x}} \mapsto \overline{\mathsf{s}}]\,\mathsf{t}_0) : \mathsf{D}$ by T-UCast.

Case (T-DCast): $\mathsf{t} = (\mathsf{D})\mathsf{t}_0 \quad \Gamma, \overline{\mathsf{x}} : \overline{\mathsf{B}} \vdash \mathsf{t}_0 : \mathsf{C} \quad \mathsf{D} <: \mathsf{C} \quad \mathsf{D} \neq \mathsf{C}$

By the induction hypothesis, there is some E such that $\Gamma \vdash [\overline{\mathsf{x}} \mapsto \overline{\mathsf{s}}]\,\mathsf{t}_0 : \mathsf{E}$ and $\mathsf{E} <: \mathsf{C}$. If $\mathsf{E} <: \mathsf{D}$ or $\mathsf{D} <: \mathsf{E}$, then $\Gamma \vdash (\mathsf{D})([\overline{\mathsf{x}} \mapsto \overline{\mathsf{s}}]\ \mathsf{t}_0) : \mathsf{D}$ by T-UCast or T-DCast, respectively. If both $\mathsf{D} \not<: \mathsf{E}$ and $\mathsf{E} \not<: \mathsf{D}$, then $\Gamma \vdash (\mathsf{D})([\overline{\mathsf{x}} \mapsto \overline{\mathsf{s}}]\ \mathsf{t}_0) : \mathsf{D}$ (with a *stupid warning*) by T-SCast.

Case (T-SCast): $\mathsf{t} = (\mathsf{D})\mathsf{t}_0 \quad \Gamma, \overline{\mathsf{x}} : \overline{\mathsf{B}} \vdash \mathsf{t}_0 : \mathsf{C} \quad \mathsf{D} \not<: \mathsf{C} \quad \mathsf{C} \not<: \mathsf{D}$

By the induction hypothesis, there is some E such that $\Gamma \vdash [\overline{\mathsf{x}} \mapsto \overline{\mathsf{s}}]\,\mathsf{t}_0 : \mathsf{E}$ and $\mathsf{E} <: \mathsf{C}$. If $\mathsf{E} \not<: \mathsf{D}$, by the transitivity of $<:$, $\mathsf{D} \not<: \mathsf{E}$ since $\mathsf{D} \not<: \mathsf{C}$ and $\Gamma \vdash (\mathsf{D})([\overline{\mathsf{x}} \mapsto \overline{\mathsf{s}}]\ \mathsf{t}_0) : \mathsf{D}$ (with a *stupid warning*) by T-SCast. On the other hand, if $\mathsf{E} <: \mathsf{D}$, then $\Gamma \vdash (\mathsf{D})([\overline{\mathsf{x}} \mapsto \overline{\mathsf{s}}]\ \mathsf{t}_0) : \mathsf{D}$ by T-UCast. This case is different from FJ since E can be a subclass of C and D with $\mathsf{D} \not<: \mathsf{C}$ and $\mathsf{C} \not<: \mathsf{D}$. In FFJ, there would be either $\mathsf{D} <: \mathsf{C}$ or $\mathsf{C} <: \mathsf{D}$. Thus, in FFJ, if $\mathsf{E} <: \mathsf{D}$ T-UCast finishes the case; otherwise, like in FJ, T-SCast finishes the case. □

Lemma C3 (Weakening): If $\Gamma \vdash \mathsf{t} : \mathsf{C}$, then $\Gamma, \mathsf{x} : \mathsf{D} \vdash \mathsf{t} : \mathsf{C}$

Proof. Straightforward induction. Nothing changes in FFJ compared to FJ. □

Lemma C4: If $\mathit{mtype}(\mathsf{m}, \mathsf{C}_0) = \overline{\mathsf{D}} \to \mathsf{D}$, and $\mathit{mbody}(\mathsf{m}, \mathsf{C}_0) = (\overline{\mathsf{x}}, \mathsf{t})$, then for some D_0 and some $\mathsf{C} <: \mathsf{D}$ we have $\mathsf{C}_0 <: \mathsf{D}_0$ and $\overline{\mathsf{x}} : \overline{\mathsf{D}}, \mathsf{this} : \mathsf{D}_0 \vdash \mathsf{t} : \mathsf{C}$.

Proof. By induction on the derivation of $\mathit{mbody}(\mathsf{m}, \mathsf{C}_0)$. The base case (in which m is defined in C_0) is easy since m is defined in $CT(\mathsf{C}_0)$ and the well-formedness of the class table implies that we must have derived $\overline{\mathsf{x}} : \overline{\mathsf{D}}, \mathsf{this} : \mathsf{C}_0 \vdash \mathsf{t} : \mathsf{C}$ by the well-formedness rules of method declarations and refinements. The induction step is also straightforward. This lemma holds for FFJ since a method refinement does not change the argument and result types of a method and this points always to the class that is refined. □

Proof of Theorem C1 (Preservation). By induction on a derivation of $\mathsf{t} \longrightarrow \mathsf{t}'$, with a case analysis on the final rule.

Case (E-ProjNew1): $\mathsf{t} = \mathsf{new}\ \mathsf{C}_0(\overline{\mathsf{v}}).\mathsf{f}_i \quad \mathsf{t}' = \mathsf{v}_i \quad \mathit{fields}(\mathsf{C}_0) = \overline{\mathsf{D}}\ \overline{\mathsf{f}}, \overline{\mathsf{E}}\ \overline{\mathsf{h}}$
$|\overline{\mathsf{v}}| = |\overline{\mathsf{f}}| \quad |\overline{\mathsf{C}}| = |\overline{\mathsf{D}}|$

With default values, the number of arguments $\overline{\mathsf{v}}$ that are supplied during the instantiation of the class can be lesser than the number of fields $\overline{\mathsf{D}}\ \overline{\mathsf{f}}, \overline{\mathsf{E}}\ \overline{\mathsf{h}}$ of the class. In this case, a value v_i is supplied for the field f_i that is projected. The remaining proof is similar to FFJ without default values.

Case (E-ProjNew2): $\mathsf{t} = \mathsf{new}\ \mathsf{C}_0(\overline{\mathsf{v}}).\mathsf{h}_i \quad \mathsf{t}' = \mathit{default}(\mathsf{E}_i) \quad \mathit{fields}(\mathsf{C}_0) = \overline{\mathsf{D}}\ \overline{\mathsf{f}}, \overline{\mathsf{E}}\ \overline{\mathsf{h}}$
$|\overline{\mathsf{v}}| = |\overline{\mathsf{f}}| \quad |\overline{\mathsf{C}}| = |\overline{\mathsf{D}}|$

In this case, no value is supplied for the field h_i that is projected. Therefore, by rule E-PROJNEW2, a default value $\mathit{default}(\mathsf{E}_i)$ is supplied. By the typing rule T-DEFAULT $\Gamma \vdash \mathit{default}(\mathsf{E}_i) : E_i$, which finishes the case, since $\mathsf{E}_i <: \mathsf{E}_i$.

CASE (E-INVKNEW): $\mathsf{t} = (\mathsf{new}\ \mathsf{C}_0(\overline{\mathsf{v}})).\mathsf{m}(\overline{\mathsf{u}}) \quad \mathsf{t}' = [\overline{\mathsf{x}} \mapsto \overline{\mathsf{u}}, \mathsf{this} \mapsto \mathsf{new}\ \mathsf{C}_0(\overline{\mathsf{v}})]\,\mathsf{t}_0$
$\mathit{mbody}(\mathsf{m}, \mathsf{C}_0) = (\overline{\mathsf{x}}, \mathsf{t}_0)$

The final rule in the derivation of $\Gamma \vdash \mathsf{t} : \mathsf{C}$ must be T-INVK and T-NEW, with premises $\Gamma \vdash \mathsf{new}\ \mathsf{C}_0(\overline{\mathsf{v}}) : \mathsf{C}_0$, $\Gamma \vdash \overline{\mathsf{u}} : \overline{\mathsf{C}}$, $\overline{\mathsf{C}} <: \overline{\mathsf{D}}$, and $\mathit{mtype}(\mathsf{m}, \mathsf{C}_0) = \overline{\mathsf{D}} \to \mathsf{C}$. By Lemma C4, we have $\overline{\mathsf{x}} : \overline{\mathsf{D}}, \mathsf{this} : \mathsf{D}_0 \vdash \mathsf{t} : \mathsf{B}$ for some D_0 and B, with $\mathsf{C}_0 <: \mathsf{D}_0$ and $\mathsf{B} <: \mathsf{C}$. By Lemma C3, $\Gamma, \overline{\mathsf{x}} : \overline{\mathsf{D}}, \mathsf{this} : \mathsf{D}_0 \vdash \mathsf{t}_0 : \mathsf{B}$. Then, by Lemma C2, $\Gamma\,[\overline{\mathsf{x}} \mapsto \overline{\mathsf{u}}, \mathsf{this} \mapsto \mathsf{new}\ \mathsf{C}_0(\overline{\mathsf{v}})]\,\mathsf{t}_0 : \mathsf{E}$ for some $\mathsf{E} <: \mathsf{B}$. By the transitivity of $<:$, we obtain $\mathsf{E} <: \mathsf{C}$. Letting $\mathsf{C}' = \mathsf{E}$ completes the case.

Note that this case is not affected by the extension of FFJ$_{ME}$ since E-INVKNEW2 substitutes each occurrence of original. This results in a method body that is indistinguishable from a common FFJ (or FJ) method body. Thus, the proof proceed with the assumptions of basic FFJ.

CASE (E-CASTNEW): $\mathsf{t} = (\mathsf{D})(\mathsf{new}\ \mathsf{C}_0(\overline{\mathsf{v}})) \quad \mathsf{C}_0 <: \mathsf{D} \quad \mathsf{t}' = \mathsf{new}\ \mathsf{C}_0(\overline{\mathsf{v}})$

The proof of $\Gamma \vdash (\mathsf{D})(\mathsf{new}\ \mathsf{C}_0(\overline{\mathsf{v}})) : \mathsf{C}$ must end with T-UCAST since ending with T-SCAST or T-DCAST would contradict the assumption $\mathsf{C}_0 <: \mathsf{D}$. The premises of T-UCAST, give us $\Gamma \vdash \mathsf{new}\ \mathsf{C}_0(\overline{\mathsf{v}}) : \mathsf{C}_0$ and $\mathsf{D} = \mathsf{C}$, finishing the case.

The cases for the congruence rules are easy. We show just the case E-CAST.

CASE (E-CAST): $\mathsf{t} = (\mathsf{D})\mathsf{t}_0 \quad \mathsf{t}' = (\mathsf{D})\mathsf{t}_0' \quad \mathsf{t}_0 \longrightarrow \mathsf{t}_0'$

There are three subcases according to the last typing rule used.

SUBCASE (T-UCAST): $\Gamma \vdash \mathsf{t}_0 : \mathsf{C}_0 \quad \mathsf{C}_0 <: \mathsf{D} \quad \mathsf{D} = \mathsf{C}$

By the induction hypothesis, $\Gamma \vdash \mathsf{t}_0' : \mathsf{C}_0'$ for some $\mathsf{C}_0' <: \mathsf{C}_0$. By transitivity of $<:$, $\mathsf{C}_0' <: \mathsf{C}$. Therefore, by T-UCAST $\Gamma \vdash (\mathsf{C})\mathsf{t}_0' : \mathsf{C}$ (with no additional *stupid warning*).

SUBCASE (T-DCAST): $\Gamma \vdash \mathsf{t}_0 : \mathsf{C}_0 \quad \mathsf{D} <: \mathsf{C}_0 \quad \mathsf{D} = \mathsf{C}$

By the induction hypothesis, $\Gamma \vdash \mathsf{t}_0' : \mathsf{C}_0'$ for some $\mathsf{C}_0' <: \mathsf{C}_0$. If $\mathsf{C}_0' <: \mathsf{C}$ or $\mathsf{C} <: \mathsf{C}_0'$, then $\Gamma \vdash (\mathsf{C})\mathsf{t}_0' : \mathsf{C}$ by T-UCAST or T-DCAST (without any additional *stupid warning*). On the other hand, if both $\mathsf{C}_0' \not<: \mathsf{C}$ or $\mathsf{C} \not<: \mathsf{C}_0'$, then, $\Gamma \vdash (\mathsf{C})\mathsf{t}_0' : \mathsf{C}$ with a *stupid warning* by T-SCAST.

SUBCASE (T-SCAST): $\Gamma \vdash \mathsf{t}_0 : \mathsf{C}_0 \quad \mathsf{D} \not<: \mathsf{C}_0 \quad \mathsf{C}_0 \not<: \mathsf{D} \quad \mathsf{D} = \mathsf{C}$

By the induction hypothesis, $\Gamma \vdash \mathsf{t}_0' : \mathsf{C}_0'$ for some $\mathsf{C}_0' <: \mathsf{C}_0$. If $\mathsf{C}_0' \not<: \mathsf{C}$, then $\mathsf{C} \not<: \mathsf{C}_0'$ since $\mathsf{C} \not<: \mathsf{C}_0$ and, therefore, $\Gamma \vdash (\mathsf{C})\mathsf{t}_0' : \mathsf{C}$ with *stupid warning*. If $\mathsf{C}_0' <: \mathsf{C}$, then $\Gamma \vdash (\mathsf{C})\mathsf{t}_0' : \mathsf{C}$ by T-UCAST (with no additional *stupid warning*). This subcase is analogous to the case T-SCAST of the proof of Lemma C2.

□

THEOREM C2 (Progress): Suppose t is a well-typed term.

1. If t includes $\mathsf{new}\ \mathsf{C}_0(\overline{\mathsf{t}}).\mathsf{f}_i$ as a subterm, then $\mathit{fields}(\mathsf{C}_0) = \overline{\mathsf{C}}\ \overline{\mathsf{f}}$ for some $\overline{\mathsf{C}}$ and $\overline{\mathsf{f}}$.

2. If t includes $\mathsf{new}\ \mathsf{C}_0(\bar{\mathsf{t}}).\mathsf{m}(\bar{\mathsf{u}})$ as a subterm, then $\mathit{mbody}(\mathsf{m}, \mathsf{C}_0) = (\bar{\mathsf{x}}, \mathsf{t}_0)$ and $|\bar{\mathsf{x}}| = |\bar{\mathsf{u}}|$ for some $\bar{\mathsf{x}}$ and t_0.

Proof. If t has $\mathsf{new}\ \mathsf{C}_0(\bar{\mathsf{t}}).\mathsf{f}_i$ as a subterm, then, by well-typedness of the subterm, it is easy to check that $\mathit{fields}(\mathsf{C}_0)$ is well-defined and f_i appears in it. The fact that class refinements may add fields (that have not been defined already) does not change this conclusion. Similarly, if t has $\mathsf{new}\ \mathsf{C}_0(\bar{\mathsf{t}}).\mathsf{m}(\bar{\mathsf{u}})$ as a subterm, then it is also easy to show that $\mathit{mbody}(\mathsf{m}, \mathsf{C}_0) = (\bar{\mathsf{x}}, \mathsf{t}_0)$ and $|\bar{\mathsf{x}}| = |\bar{\mathsf{u}}|$ from the fact that $\mathit{mtype}(\mathsf{m}, \mathsf{C}_0) = \bar{\mathsf{C}} \rightarrow \mathsf{D}$ where $|\bar{\mathsf{x}}| = |\bar{\mathsf{C}}|$. This conclusion holds for FFJ since a method refinement must have the same signature than the method refined. □

Theorem C3 (FFJ Type Soundness): If $\emptyset \vdash \mathsf{t} : \mathsf{C}$ and $\mathsf{t} \longrightarrow^* \mathsf{t}'$ with t' a normal form, then t' is either a value v with $\emptyset \vdash \mathsf{v} : \mathsf{D}$ and $\mathsf{D} <: \mathsf{C}$, or a term containing $(\mathsf{D})(\mathsf{new}\ \mathsf{C}(\bar{\mathsf{t}}))$ in which $\mathsf{C} <: \mathsf{D}$.

Proof. Immediate from Theorem C1 and C2. Nothing changes in the proof of Theorem C3 for FFJ compared to FJ. □

Theorem C4 (Reduction Preserves Cast-Safety): If t is cast-safe in Γ and $\mathsf{t} \longrightarrow \mathsf{t}'$, then t' is cast-safe in Γ.

Proof. The proof is straightforward; nothing changes for FFJ as compared to FJ. □

Theorem C5 (Progress of Cast-Safe Programs): Suppose t is cast-safe in Γ. If t has $(\mathsf{C})\mathsf{new}\ \mathsf{C}_0(\bar{\mathsf{t}})$ as a subtem, then $\mathsf{C}_0 <: \mathsf{C}$.

Proof. The proof is straightforward; nothing changes for FFJ as compared to FJ. □

Corollary C1 (No Typecast Errors in Cast-Safe Programs): If t is cast-safe in $\emptyset$ and $\mathsf{t} \longrightarrow^* \mathsf{t}'$ wiht t' a normal form, then t' is a value.

Proof. The proof is straightforward; nothing changes for FFJ as compared to FJ. □