FeatherTrait: A Modest Extension of Featherweight Java

LUIGI LIQUORI
INRIA
and
ARNAUD SPIWACK
ENS Cachan

In the context of *statically typed, class-based languages*, we investigate classes that can be extended with *trait* composition. A trait is a collection of methods without state; it can be viewed as an *incomplete stateless class*. Traits can be composed in any order, but only make sense when imported by a class that provides state variables and additional methods to disambiguate conflicting names arising between the imported traits. We introduce FeatherTrait Java (FTJ), a conservative extension of the simple lightweight class-based calculus Featherweight Java (FJ) with *statically typed traits*. In FTJ, classes can be built using traits as basic behavioral bricks; method conflicts between imported traits must be resolved *explicitly* by the user either by (i) aliasing or excluding method names in traits, or by (ii) overriding explicitly the conflicting methods in the class or in the trait itself. We present an operational semantics with a lookup algorithm, and a sound type system that guarantees that evaluating a well-typed expression never yields a *message-not-understood* run-time error nor gets the interpreter stuck. We give examples of the increased expressive power of the trait-based inheritance model. The resulting calculus appears to be a good starting point for a rigorous mathematical analysis of typed class-based languages featuring trait-based inheritance.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*syntax, semantics*; D.3.2 [**Programming Languages**]: Language Classifications—*object-oriented languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*classes and objects*; *inheritance*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*object-oriented constructs*

General Terms: Design, Languages, Theory

Additional Key Words and Phrases: Java, inheritance, language design, language semantics

---

This work was supported by French grant CNRS *ACI Modulogic*, and by the AEOLUS FET Global Computing Proactive IST-015964, *Algorithmic Principles for Building Efficient Overlay Computers*.
Authors' addresses: L. Liquori, INRIA, 2004 Route des Lucioles, BP 93, F-06902 Sophia Antipolis, France, email: Luigi.Liquori@inria.fr; A. Spiwack, Ecole Normale Supérieure de Cachan, 61 avenue du Président Wilson, F-94235 Cachan cedex, France, email: Arnaud.Spiwack@dptinfo.ens-cachan.fr.

ACM Reference Format:

Liquori, L., Spiwack, A. 2007. Feather Trait: A modest extension of Featherweight Java. ACM Trans. Program. Lang. Syst. 30, 2, Article 11 (March 2008), 32 pages. DOI = 10.1145/1330017.1330022 http://doi.acm.org/10.1145/1330017.1330022

---

1. INTRODUCTION

"Inside every large language is a small language
struggling to get out ..." [Igarashi et al. 2001]

> "... and inside every small language is a sharp
> extension looking for better expressivity ..."

Inheritance is commonly viewed as one crucial feature of object-oriented languages. There are essentially three kinds of inheritance, namely, single inheritance, multiple inheritance (including mixin-based inheritance and trait-based inheritance), and object-based (delegation-based) inheritance.

Single Inheritance is the simplest model, adopted, for example, in Java [Sun] and C# [Microsoft]. The inheritance relation forms a tree, and a derived class can inherit methods and variables only from its parent class.

Object-Based Inheritance, also called *delegation-based* inheritance, adopted, for example, in Self [Ungar and Smith 1987] and Obliq [Cardelli 1995]. It is the most flexible inheritance model based on the idea that objects are created dynamically by modifying existing objects used as *prototypes*. An object created from a given prototype may add new methods or redefine methods supplied by the prototype; this may change the object type. Any message sent to the created object is handled directly by it if it contains the corresponding method, otherwise the message is "passed back", that is, *delegated* to the prototype. Because of the extreme dynamicity with respect to the class-based model, even in the presence of a single parent inheritance, static type-checking is very difficult [Liquori 1997, 1998; Di Gianantonio et al. 1998].

Multiple Inheritance is a richer but debated model (adopted, e.g., in C++ [Stroustrup 1997]): a derived class can inherit from many parent classes (forms an inheritance directed acyclic graph).

Compared with single inheritance, multiple inheritance adds additional run-time overhead (potentially involving dynamic binding). The literature presents a rich list of potential problems with multiple inheritance, including fork-join (diamond) inheritance, the yo-yo problem, access to overridden methods, and the complication of type-checking in the presence of parametric classes. Among this more flexible inheritance model, two concepts have been developed in the past few years

(1) *Mixins*. There are essentially two kinds of mixins, *mixin-classes* and *mixin-modules*.

A mixin-class is like a class (it contains defined methods, that is, interface-types and bodies, or deferred methods, that is, only interface-types) that can be applied to various parent classes in order to extend them with the methods contained in the mixin-class itself. Mixin-classes are named and can be applied to a parent class. The concept of mixin-classes, invented in the 90's by Bracha and Cook [Bracha and Cook 1990], has been studied in the recent years by, among others, Flatt et al. [1998], Bono et al. [1999], and in the contexts of an extension of Java, by Ancona et al. [2003], and by Allen et al. [2003].

A mixin-module (introduced to solve implementation inheritance problems) is a module which supports deferred components. Mixins are named and can be composed using an *ad hoc* algebra (e.g., `merge` and `restrict` operations). Mixin-modules were introduced by Bracha [1992], and have been studied more recently by Duggan and Sourelis [1996], Findler and Flatt [1998], Flatt and Felleisen [1998], Wells and Vestergaard [2000], Ancona and Zucca [2002b, 2002a], and Hirschowitz et al. [2004].

(2) *Traits*. Defined by Schärli et al. [2003] and Ducasse et al. [2006], these have recently emerged as a novel technique for building composable units of behaviors in a dynamically typed language *à la* Smalltalk. Intuitively, a trait is just a collection of methods, that is, behaviors without state. Derived traits can be built from an *unordered* list of parent traits, together with new method declarations. Thus, traits are (incomplete) classes without state. Traits can be composed in any order. A trait makes sense only when "imported" by a class that provides state variables and possibly some additional methods to disambiguate conflicting names arising among the imported traits. The order for importing traits in classes is irrelevant.

Historically, traits, intended as a collection of state[1] and behavior, have been originally employed in the pure object-based languages Self [Ungar and Smith 1987], or in the language Obliq [Cardelli 1995], or for the encoding of classes as records-of-premethods in the Object Calculus by Abadi and Cardelli [1996].

More recently, typed traits, intended as pure behavior without state, have been introduced by Fisher and Reppy in an object-based core calculus for the Moby programming language (of the ML [Milner et al. 1997] family) [Moby Team 2007; Fisher and Reppy 2004]. Then, traits have been immerged in Igarashi et al. [2001] Featherweight Java by Liquori and Spiwack [2004], studied by Smith and Drossopoulou [2005] in Java setting, and implemented by Odersky in the class-based language Scala [Scala Team 2007], and in the new language Fortress by Allen et al. [2007]. Here, programs are first type-checked and then executed, forgetting type information; in this way compilation ensures the absence of *message-not-understood* run-time errors, enhancing safety greatly and speeding-up the compiled code.

1.1 Contributions

FeatherTrait Java (FTJ), described in this article, conservatively extends the simple calculus of Featherweight Java (FJ) by Igarashi et al. [2001] with statically-typed traits. The main aim is to introduce the typed trait-based inheritance in a class-based calculus *à la* Java; the calculus features mutually

---

[1] This is in contrast to traits of Schärli et al. [2003] and Ducasse et al. [2006].

recursive class declarations, object creation, field access, method invocation and overriding, method recursion through `this`, subtyping and simple casting. Just as with FJ, some of the features of Java that we *do not* model include assignment, interfaces, overloading, base types (`int`, `boolean`, `String`, etc.), `null` pointers, abstract method declaration, shadowing of superclass fields by subclass fields, access control (`public`, `private`, etc), and exceptions. Since FTJ provides no operations with side effects, a method body always consists of `return` followed by an expression.

The main contributions of this article are

(1) We define the calculus FTJ, a conservative extension of FJ featuring trait inheritance. Multiple traits can be inherited by one class, and conflicts between common methods defined in two or more inherited traits must be resolved *explicitly* by the user either by (i) aliasing or excluding method names in traits, or by (ii) overriding explicitly the conflicted methods in the class that imports those traits or in the trait itself.

(2) We define a simple type system that type-checks traits when imported in classes, resulting in a sharp and lightweight extension of the type system of FJ. This can be considered as a first step in adding a powerful but safe form of trait-based inheritance to the Java language.

1.2 Outline of the Article

The article is structured as follows. In Section 2, we review the *untyped* trait-based inheritance model and we present the main ideas underlying our *typed* trait-based inheritance model for Java. In Section 3, we present the syntax of FTJ, together with some useful notational conventions. Section 4 presents the operational semantics, while Section 5 presents the type system of FTJ. Section 6 presents the main meta-theoretical results. Section 7 presents a few examples of using traits in FTJ. Section 8 discusses related work and Section 9 concludes. Appendices A and B contain the full formal system, while Appendix C contains the full proofs.

The presentation is kept as simple as possible, with a syntax and a semantics for FTJ made as close as possible to this of FJ, few definitions and few theorems. Some knowledge of the syntax, semantics and type system of FJ may be helpful in reading this article. A preliminary version of this work appeared in June 2004 as an INRIA technical report [Liquori and Spiwack 2004].

2. TRAIT-BASED INHERITANCE

We start this section with a brief presentation of the main concepts of traits, using FTJ syntax. One useful feature of *trait-based inheritance* is that when a conflict arises between traits included in the same class (e.g., a method defined in two different traits), then the conflict is signaled and it is up to the user to *explicitly* and *manually* resolve the conflict. Three simple rules can be easily implemented in the method-lookup algorithm for that purpose

(1) Methods defined in a class take precedence over methods defined in the traits imported by the class.

(2) Methods defined in a composite trait take precedence over methods defined in the imported traits.
(3) Methods defined in traits (imported by a class) take precedence over methods defined in its parent class.

The above rules are the simple recipe of the trait-based inheritance model. They greatly increase the flexibility of the calculus that uses traits.

Another property of trait-based inheritance is that a class that imports traits is semantically equivalent to a class that defines *in situ* all the methods defined in traits. This can be done via *flattening*, which immediately suggests how to build a compiler translating FTJ code into FJ code, via code duplication.

A trait can import from other traits. Hence, it *requires* methods that are not defined in the trait itself, those methods being useful in order to “complete” its behavior. In FTJ syntax

```
trait T1 {String p(){return ‘‘hello’’;}}
trait T2 {String p(){return ‘‘world’’;}}
trait T3 imports T1 T2
         {String m(){return (...this.p()...);}      p is a required method
         !String p(){return ‘‘hello world’’;}  p is an overriding method
         !String n(){return (...this.q()...);}}    q is a required method
```

Trait `T3` *imports traits* `T1` *and* `T2`*, overrides* `p`*, and* `q` *is still a required method*

Observe that a trait is by definition *potentially incomplete*, that is, it cannot be instantiated into a “runnable” object, since they have no instance variable, and it can lack some method implementations, for example,

```
trait T4 {Object p(){return (...this.r()...);}}    r is a required method
trait T5 {Object q(){return (...this.s()...);}}    s is a required method
trait T6 imports T4 T5
         {Object m(){return (...this.p()...);}     p is a required method
         !Object n(){return (...this.q()...);}}    q is a required method
```

Trait `T6` *imports traits* `T4` *and* `T5`*, and* `r` *and* `s` *are still required methods*

2.1 Conflict Resolution

When dealing with trait inheritance, conflicts can arise; for example a class `C` might import two traits `T1` and `T2` defining the same method `p` with different behavior. Conflicts between traits must be resolved *manually*, that is, there is no special or rigid discipline to learn how to use traits. Once a conflict is detected, there are essentially three ways to resolve the conflict (below, “winner” denotes the body selected by the lookup algorithm)

(1) *Overriding a New Method* `p` *Inside the Class.* A new method `p` is redefined inside the class with an new behavior. The (trait-based) lookup algorithm will hide the conflictin traits in favor of the overriding method defined in

the class. In FTJ syntax

```
class C extends Object
        imports T1 T2       each trait defines a (different) behavior for p
{...;...                                     instance vars and constructor
 D p(...){...}}                             new behavior for p, the winner
```

(2) *Aliasing the Method* `p` *in Traits and Redefining the Method in Class.* The method `p` is aliased in `T1` and `T2` with new different names. A new behavior for `p` can now be given in the class `C` (possibly re-using the aliased methods `p_of_T1` and `p_of_T2` which are no longer in conflict). In FTJ syntax

```
class C extends Object
        imports T1 with {p@p_of_T1}[2]          T1 aliases p with p_of_T1
                T2 with {p@p_of_T2}             T2 aliases p with p_of_T2
{...;...                                     instance vars and constructor
 D p(...){...}}            new winner behavior for p, it may use p_of_T1/2
```

(3) *Excluding the Method* `p` *in One of the Traits.* One method `p` in trait `T1` or `T2` is excluded. This solves the conflict in favor of one trait. In FTJ syntax

```
class C extends Object
        imports T1                            contains the winner method p
                T2 minus {p}                       method p is now hidden
{...;...}                         instance vars, constructor and methods
```

A *diamond problem* occurs in the following situation. Let `T` be a trait with a method `p`, and let `T1` and `T2` be two traits that inherit a method `p` from `T`. Then, a trait or class that imports both `T1` and `T2` would ostensibly have two definitions for the method `p`. One point of view is that this is harmless since both definitions for `p` are the same. In contrast, Snyder [1987] suggests that diamonds should be considered as conflicts. The type system of FTJ statically detects all possible diamond conflicts and considers them as legal, that is, type-safe.

3. FEATHERTRAIT JAVA

In FTJ, a program consists of a collection of class declarations, plus a collection of trait declarations and an expression to be evaluated.

3.1 Notational Conventions

—We adopt the same notational conventions and *hygiene conditions* as FJ, with the following additions: the metavariable `T` ranges over *trait names*, and `TA` ranges over *trait alterations*. `TL` (respectively `CL`) ranges over trait declarations (respectively class declarations). `TT` (respectively `CT`) ranges over *trait tables* (respectively *class tables*), where a *trait table* `TT` is a partial function from trait names to trait alterations, and a *class table* `CT` is

---

[2] In the original model of Schärli et al. [2003], `m@n` was denoted by `n->m`.

$$\begin{array}{llll}
\mathtt{CL} & ::= & \mathtt{class\ C\ extends\ C\ [imports\ \overline{TA}]\{\overline{C}\ \overline{f};K\ \overline{M}\}} & \text{Class Declarations}\\
\mathtt{TL} & ::= & \mathtt{trait\ T\ [imports\ \overline{TA}]\{\overline{M}\}} & \text{Trait Declarations}\\
\mathtt{TA} & ::= & \mathtt{T} \mid \mathtt{TA\ with\ \{m@m\}} \mid \mathtt{TA\ minus\ \{m\}} & \text{Trait Alterations}\\
\mathtt{K} & ::= & \mathtt{C(\overline{C}\ \overline{f})\{super(\overline{f});this.\overline{f}=\overline{f};\}} & \text{Constructors}\\
\mathtt{M} & ::= & \mathtt{C\ m(\overline{C}\ \overline{x})\{return\ e;\}} & \text{Methods}\\
\mathtt{e} & ::= & \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m(\overline{e})} \mid \mathtt{new\ C(\overline{e})} \mid \mathtt{(C)e} & \text{Expressions}
\end{array}$$

Fig. 1. Syntax of FTJ.

a partial function from class names to class declarations. Finally, `K` (respectively `M`) ranges over constructors (respectively methods), `f` (respectively, `m`, `n`, `p`, and `q`) ranges over field names (respectively method names), `e` (respectively `x`) ranges over expressions (respectively variables), and `A`, `B`, `C`, `D`, and `E` range over class names, and $\mathtt{M}_\perp$ (respectively, $(\overline{\mathtt{x}}, \mathtt{e})_\perp$) ranges over methods (respectively method bodies) and the special failure value *fail*.

—Sequences of field declarations, parameter names, method and trait declarations, and trait alterations (vector notation) are assumed to contain no duplicate names.

—As in FJ, we set the root class `Object` as the superclass of all classes: this class has no method nor field, and does not appear in the class table `CT`.

3.2 Syntax

The syntax of FTJ is given in Figure 1: it extends the syntax of FJ. An FTJ program is a triple (`CT`, `TT`, `e`) of a class table, a trait table, and an expression. A class

$$\mathtt{class\ C\ extends\ C\ imports}^{3}\ \overline{\mathtt{TA}}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\mathtt{K}\ \overline{\mathtt{M}}\}$$

in FTJ is composed of field declarations $\overline{\mathtt{C}}\ \overline{\mathtt{f}}$, a constructor `K`, some new or redefined methods $\overline{\mathtt{M}}$, plus a list of imported (and possibly altered) traits $\overline{\mathtt{TA}}$. A trait

$$\mathtt{trait\ T\ imports}\ \overline{\mathtt{TA}}\ \{\overline{\mathtt{M}}\}$$

is composed of a list of methods $\overline{\mathtt{M}}$ and some other traits $\overline{\mathtt{TA}}$ imported by the trait itself. All conflicts will be discovered using the trait checking rules, and resolved using the class checking rule. As noted above, the "diamond problem" is not considered as a conflict. Expressions are the usual ones of FJ.

---

[3] The keyword `imports` was preferred to the keyword `implements` (*à la* Java) because traits already implements some methods.

3.3 Trait-Based Inheritance in FTJ

We list the features of FTJ

- A method defined in a class has the same behavior as a method defined in a trait and imported by a class.
- A class (or a trait) may import many traits: the composition order of traits does not matter.
- A trait can be *altered* either by dropping a method name `m`, or by *aliasing* a method name `m` with another method name `n`.
- A *modified method lookup* is implemented to deal with traits and trait alterations.
- A method defined in a class (respectively, trait) takes precedence over, or *overrides* a method defined in a trait and imported by the class (respectively, trait).
- A *diamond schema* is accepted statically.
- A trait is type-checked only inside a class, that is, inside a complete unit of behavior (that is, inside a class).

FTJ, is, like FJ, a *functional* calculus, that is, there is no notion of state and no assignment; for example, instead of assigning a different value to a variable, we build another object completely from scratch with the new modified value in place of the old one. The possibility to type-check traits only once (see conclusions), is beyond the scope of this article.

4. OPERATIONAL SEMANTICS

The small-step operational semantics of FTJ is the same as that of FJ. The essential difference between the two is the new lookup algorithm. The reduction relation, given in Appendix A, defines the relation $\mathtt{e} \longrightarrow \mathtt{e}'$, read "expression $\mathtt{e}$ reduces to expression $\mathtt{e}'$ in one step". As in FJ, the variable `this` denotes the receiver itself (in the substitution). The first two rules (Run·Field) and (Run·Call) deal with field lookup and method call, while the last rule (Run·Cast) is a typecast. As usual, these reduction rules can be applied at any point of the computation, so the classical congruence rules apply, which we omit here, as we omit the rules of subtyping, proving judgments of the form $\mathtt{A} <: \mathtt{B}$ (see Appendix A). The functions *mbody* and *fields* are slight extensions of the corresponding functions in FJ. The *mbody* function needs to be customized in order to find method bodies defined within traits and altered traits. The *mbody* function calls another function, *tlook*, which deals with trait lookup. The *tlook* function may call another function, *altlook*, which deals with trait alterations lookup.

4.1 Lookup Algorithm

The lookup algorithm described in Appendix A takes into account the three simple method-precedence rules. Extra complications with respect to the lookup in FJ arise because of trait inheritance and because traits can be altered.

Field Lookup. It is performed as in FJ (see Appendix A).

Method Lookup. This is performed by the rules (MBdy·Cla) that searches in the current class, then (MBdy·Tr) that searches in all imported traits, and (MBdy·SCla) that searches in the direct parent class. The trait lookup function *tlook* searches the method body of `m` only if `m` is not overridden in the class; this forces the uniqueness of the search, otherwise a conflict would arise, since a method defined in the class could have overridden method `m`. In particular, the rule (MBdy·Tr) is as follows

$$\frac{\begin{array}{l}\texttt{CT(C)} = \texttt{class C extends D imports } \overline{\texttt{TA}}\ \{\overline{\texttt{C}}\ \overline{\texttt{f}};\texttt{K}\ \overline{\texttt{M}}\}\\ \texttt{m} \notin \mathit{meth}(\overline{\texttt{M}}) \quad \mathit{tlook}(\texttt{m}, \overline{\texttt{TA}}) = \texttt{B m}(\overline{\texttt{B}}\ \overline{\texttt{x}})\{\texttt{return e;}\}\end{array}}{\mathit{mbody}(\texttt{m}, \texttt{C}) = (\overline{\texttt{x}}, \texttt{e})}\ (\text{MBdy}\cdot\text{Tr})$$

Here *meth* is a function, defined in Appendix A, that collects method names.

Trait Lookup. This is performed by the function *tlook* that, intuitively, searches the body of the method `m` "traversing" a sequence of trait alterations. The two simple inference rules are as follows

$$\frac{\exists \texttt{TA} \in \overline{\texttt{TA}}.\ \mathit{altlook}(\texttt{m}, \texttt{TA}) \neq \mathit{fail}}{\mathit{tlook}(\texttt{m}, \overline{\texttt{TA}}) = \mathit{altlook}(\texttt{m}, \texttt{TA})}\ (\text{Tr}\cdot\text{Ok})$$

$$\frac{\forall \texttt{TA} \in \overline{\texttt{TA}}.\ \mathit{altlook}(\texttt{m}, \texttt{TA}) = \mathit{fail}}{\mathit{tlook}(\texttt{m}, \overline{\texttt{TA}}) = \mathit{fail}}\ (\text{Tr}\cdot\text{Ko})$$

The auxiliary function *altlook* takes into account altered traits, that is, traits with dropped methods, or with aliased methods. Finding a method that has been dropped or aliased is one of the key parts of the lookup algorithm.

Trait Alteration Lookup. The trait alteration lookup rules are detailed in Appendix A. The most interesting trait alteration rules are the following ones

$$\frac{\begin{array}{l}\texttt{TT(T)} = \texttt{trait T imports } \overline{\texttt{TA}}\ \{\overline{\texttt{M}}\}\\ \texttt{B m}(\overline{\texttt{B}}\ \overline{\texttt{x}})\{\texttt{return e;}\} \in \overline{\texttt{M}}\end{array}}{\mathit{altlook}(\texttt{m}, \texttt{T}) = \texttt{B m}(\overline{\texttt{B}}\ \overline{\texttt{x}})\{\texttt{return e;}\}}\ (\text{ATr}\cdot\text{Found})$$

$$\frac{\mathit{altlook}(\texttt{n}, \texttt{TA}) = \texttt{B n}(\overline{\texttt{B}}\ \overline{\texttt{x}})\{\texttt{return e;}\}}{\mathit{altlook}(\texttt{m}, \texttt{TA with }\{\texttt{n@m}\}) = \texttt{B m}(\overline{\texttt{B}}\ \overline{\texttt{x}})\{\texttt{return e;}\}}\ (\text{ATr}\cdot\text{Ali}_1)$$

$$\frac{\mathtt{m} \neq \mathtt{p} \quad \mathtt{m} \neq \mathtt{q} \quad \mathit{altlook}(\mathtt{m}, \mathtt{TA}) = \mathtt{M}_{\perp}}{\mathit{altlook}(\mathtt{m}, \mathtt{TA\ with}\ \{\mathtt{p@q}\}) = \mathtt{M}_{\perp}}\ (\text{ATr}{\cdot}\text{Ali}_2)$$

$$\frac{\mathtt{m} \neq \mathtt{n}}{\mathit{altlook}(\mathtt{m}, \mathtt{TA\ with}\ \{\mathtt{m@n}\}) = \mathit{fail}}\ (\text{ATr}{\cdot}\text{Ali}_3)$$

$$\frac{\mathit{altlook}(\mathtt{n}, \mathtt{TA}) = \mathit{fail}}{\mathit{altlook}(\mathtt{m}, \mathtt{TA\ with}\ \{\mathtt{n@m}\}) = \mathit{fail}}\ (\text{ATr}{\cdot}\text{Ali}_4)$$

- (ATr·Found) The function *altlook* succeeds to return the body of the method we are looking for, since that method is found in the trait.
- (ATr·Ali$_1$) When looking up a method `m` in a trait alteration where `n` is aliased to `m`, we look up for the method with the former name `n`, and then we rename. The condition $\mathtt{m} \neq \mathtt{n}$ is not required since it is enforced by the type system.
- (ATr·Ali$_2$) Recursive call. The conditions $\mathtt{m} \neq \mathtt{p}$ and $\mathtt{m} \neq \mathtt{q}$ guarantee that it is another method which is aliased.
- (ATr·Ali$_3$) Failure. The condition $\mathtt{m} \neq \mathtt{n}$ is not necessary, since it is enforced by the type system; however, it has been left to emphasize that the cases are pairwise disjoint.
- (ATr·Ali$_4$) Recursive call with failure. A failure is propagated in case the premise fails.

Diamond Inheritance. Let us extend the *meth* function to trait alterations, to obtain the set of method names available in the alteration (see Appendix A). The following definitions will be useful to type-check traits and classes (key rules (Tr·Ok) and (Cla·Ok)). Let $\overset{\text{def}}{=}$ means equal by definition.

Definition 4.1. *Method Intersection and Diamond Detection*

$$\cap\overline{\mathtt{TA}} \overset{\text{def}}{=} \{\mathtt{m} \mid \exists \mathtt{TA}_1 \neq \mathtt{TA}_2 \in \overline{\mathtt{TA}}.\ \mathtt{m} \in \mathit{meth}(\mathtt{TA}_1) \cap \mathit{meth}(\mathtt{TA}_2)\}$$

$$\diamond\overline{\mathtt{TA}} \overset{\text{def}}{=} \{\mathtt{m} \mid \exists \mathtt{n}, \mathtt{TA}_1.\ \forall \mathtt{TA}_2 \in \overline{\mathtt{TA}}.\ \mathtt{m} \in \mathit{meth}(\mathtt{TA}_2) \implies \mathtt{m}\ \text{in}\ \mathtt{TA}_2 \lhd \mathtt{n}\ \text{in}\ \mathtt{TA}_1\}.$$

Intuitively

- The set $\cap\overline{\mathtt{TA}}$ denotes methods defined in more than one trait; it is used to detect conflicts when importing traits.
- The set $\diamond\overline{\mathtt{TA}}$ denotes methods that potentially determine a diamond when dealing with trait inheritance; such methods are expected to be "nonconflicting", hence accepted by the type system. The notation $\mathtt{m}\ \text{in}\ \mathtt{TA}_2 \lhd \mathtt{n}\ \text{in}\ \mathtt{TA}_1$, read "`m` of $\mathtt{TA}_1$ behaves exactly as `n` of $\mathtt{TA}_2$", will be introduced in the next paragraph.

In a nutshell: The set $\cap\overline{\mathtt{TA}}$ detects every conflict in $\overline{\mathtt{TA}}$, while the set $\diamond\overline{\mathtt{TA}}$ detects every diamond. A class declaration

$$\mathtt{class\ C\ extends\ D\ imports}\ \overline{\mathtt{TA}}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}}; \mathtt{K}\ \overline{\mathtt{M}}\}$$

is well formed only if the imported trait alterations imported by the class `C` satisfy the constraint

$$\cap\overline{\mathtt{TA}} \setminus \diamond\overline{\mathtt{TA}} \subseteq \mathit{meth}(\overline{\mathtt{M}})$$

ensuring that every conflict is resolved, that is, every new-born conflict ($\cap\overline{\mathtt{TA}}$)[4] that is not a diamond ($\diamond\overline{\mathtt{TA}}$) is being overridden. The $\subseteq$ relation, instead of the more restrictive $=$ relation, is given in order to make FTJ a conservative extension of FJ.

Method Paths in Trait Alterations. To compute $\diamond\overline{\mathtt{TA}}$, we need a relation proving judgments of the form

$$\mathtt{m}\ \mathsf{in}\ \mathtt{TA}_1 \trianglelefteq \mathtt{n}\ \mathsf{in}\ \mathtt{TA}_2$$

The meaning of this judgment is as follows: `m` is a method provided by trait $\mathtt{TA}_1$ that behaves exactly as method `n` provided by $\mathtt{TA}_2$ through any number of trait declarations or alteration steps (paths). The most interesting rules are

$$\frac{\begin{array}{l}\mathtt{TT(T) = trait\ T\ imports}\ \overline{\mathtt{TA}}\ \{\overline{\mathtt{M}}\}\\ \mathtt{TA} \in \overline{\mathtt{TA}} \qquad \mathtt{m} \in \mathit{meth}(\mathtt{TA}) \setminus \mathit{meth}(\overline{\mathtt{M}})\end{array}}{\mathtt{m}\ \mathsf{in}\ \mathtt{T} \trianglelefteq \mathtt{m}\ \mathsf{in}\ \mathtt{TA}}\ (\text{Path}\cdot\text{Inh})$$

$$\frac{\mathtt{p}\ \mathsf{in}\ \mathtt{TA}_1 \trianglelefteq \mathtt{n}\ \mathsf{in}\ \mathtt{TA}_2}{\mathtt{m}\ \mathsf{in}\ \mathtt{TA}_1\ \mathtt{with}\ \{\mathtt{p@m}\} \trianglelefteq \mathtt{n}\ \mathsf{in}\ \mathtt{TA}_2}\ (\text{Path}\cdot\text{Ali}_1)$$

$$\frac{\mathtt{m}\ \mathsf{in}\ \mathtt{TA}_1 \trianglelefteq \mathtt{n}\ \mathsf{in}\ \mathtt{TA}_2 \qquad \mathtt{m} \neq \mathtt{p} \qquad \mathtt{m} \neq \mathtt{q}}{\mathtt{m}\ \mathsf{in}\ \mathtt{TA}_1\ \mathtt{with}\ \{\mathtt{p@q}\} \trianglelefteq \mathtt{n}\ \mathsf{in}\ \mathtt{TA}_2}\ (\text{Path}\cdot\text{Ali}_2)$$

$$\frac{\mathtt{m}\ \mathsf{in}\ \mathtt{TA}_1 \trianglelefteq \mathtt{n}\ \mathsf{in}\ \mathtt{TA}_2 \qquad \mathtt{m} \neq \mathtt{p}}{\mathtt{m}\ \mathsf{in}\ \mathtt{TA}_1\ \mathtt{minus}\ \{\mathtt{p}\} \trianglelefteq \mathtt{n}\ \mathsf{in}\ \mathtt{TA}_2}\ (\text{Path}\cdot\text{Exl})$$

- (Path·Inh) If a trait `T` inherits a method `m` directly from a trait alteration `TA` and does not override it, then `m` of `T` behaves exactly as `m` of `TA`.
- (Path·Ali$_1$) If `p` of $\mathtt{TA}_1$ behaves exactly as `n` of $\mathtt{TA}_2$, then `m` of $\mathtt{TA}_1$ `with` {`p@m`} behaves exactly as `n` of $\mathtt{TA}_2$.
- (Path·Ali$_2$) If $\mathtt{m} \neq \mathtt{p}$ and $\mathtt{m} \neq \mathtt{q}$, and `m` of $\mathtt{TA}_1$ behaves exactly as `n` of $\mathtt{TA}_2$, then `m` of $\mathtt{TA}_1$ `with` {`p@q`} behaves exactly as `n` of $\mathtt{TA}_2$.
- (Path·Exl) If $\mathtt{m} \neq \mathtt{p}$, and `m` of $\mathtt{TA}_1$ behaves exactly as `n` of $\mathtt{TA}_2$, then `m` of $\mathtt{TA}_1$ `minus` {`p`} behaves exactly as `n` of $\mathtt{TA}_2$.

Reflexivity and transitivity rules are presented in Appendix A. Note that we could have also a symmetry rule, although the resulting lookup would be less algorithmic.

---

[4] Note that conflicts are resolved recursively.

Remark 4.2 (*Modified Lookup Rule*). If we drop rules (ATr·Ali$_1$) and (ATr·Ali$_4$) and (Path·Ali$_1$) and we add the "imperative-like" rule

$$\frac{\mathit{altlook}(\mathtt{n}, \mathtt{TA}) = \mathtt{M}_\perp}{\mathit{altlook}(\mathtt{m}, \mathtt{TA}\ \mathtt{with}\ \{\mathtt{n}\,@\,\mathtt{m}\}) = [\mathtt{this.m}/\mathtt{this.n}]\mathtt{M}_\perp}\ (\text{ATr·Alias·Imp}),$$

then the resulting system would still be sound (the curious reader can customize proofs of Lemmas 2, 3, 4 in Appendix C). This allows one to convert recursive calls via `this` to the new aliased name. The operation [`this.m`/`this.n`] is not strictly speaking a substitution, but rather a replacement, since `this.n` is not a variable. The purpose of such operation is to substitute every recursive and internal method call to `this.n` by `this.m`. Moreover, we assume the replacement changes the name of the method declaration, that is,

$$[\mathtt{this.m}/\mathtt{this.n}](\mathtt{B}\ \mathtt{n}\ (\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\ldots\mathtt{this.n}\ldots\}) \stackrel{\text{def}}{=} \mathtt{B}\ \mathtt{m}\ (\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\ldots\mathtt{this.m}\ldots\}.$$

The reason for this replacement is that in the *aliased* trait alteration `TA with {n@m}` we want to *alias* method `n` with `m` without altering the rest of the trait. From an external point of view, the aliased trait alteration will behave exactly as `TA`, except that the method `n` will be aliased. The (ATr·Alias·Imp) rule is not compatible with rules (ATr·Ali$_1$) and (ATr·Ali$_4$) and (Path·Ali$_1$) since it changes the body of the method we are looking for in the premises.

5. THE TYPE SYSTEM

This section introduces the most innovative rules of the FTJ type system. The full set of rules is presented in Appendix B. The type system has two steps: first, expression typing as in any statically typed language, proved as judgments of the form

$$\Gamma \vdash \mathtt{e} \in \mathtt{C}$$

second, class type-checking (as in FJ) is performed. Since everything in classes is explicitly typed, the system has only to check if the class declaration is correct. In contrast to FJ, the type-checker of FTJ also checks conflict resolutions. The FTJ type system proves judgments of the three forms

$$\mathtt{M}\ \mathtt{OK}\ \mathtt{IN}\ \mathtt{C} \quad \text{and} \quad \mathtt{TA}\ \mathtt{OK}\ \mathtt{IN}\ \mathtt{C}\ \mathtt{except}\ \overline{\mathtt{m}} \quad \text{and} \quad \mathtt{CL}\ \mathtt{OK},$$

where the tables `TT`, and `CT` are left implicit in the judgments. Traits and trait alterations are typed only with respect to a given class, the only complete unit of behavior devoted to instantiate truly "runnable" objects. Separate compilation of traits is possible but out of the scope of this article. For more advanced proposals, see Fisher and Reppy [2004] or Liquori and Spiwack [2008].

Basic Expression Checking and Valid Type Lookup. These rules (see Appendix B) have no novelties with respect to the corresponding ones of FJ.

Method Checking. The method type-checking rule is the same as in FJ (see Appendix B).

Trait Alteration Checking. The following type-checking rules are the core of this article. These rules derive judgments of the form $\mathtt{TA\ OK\ IN\ C\ except}\ \overline{\mathtt{m}}$, which means that `TA` is well typed with respect to a given class `C` where every method $\overline{\mathtt{m}}$ must be overridden. The rationale is as follows: every method occurring in the except part refers to a body that cannot be type-checked in `C`, and it is overridden by another

$$\frac{\overline{\mathtt{N}} \overset{\text{def}}{=} \{\mathtt{M} \in \overline{\mathtt{M}} \mid \neg\mathtt{M\ OK\ IN\ C}\} \quad \overline{\mathtt{TA}}\ \mathtt{OK\ IN\ C\ except}\ \overline{\mathtt{m}} \quad \cap\overline{\mathtt{TA}} \setminus \diamond\overline{\mathtt{TA}} \subseteq meth(\overline{\mathtt{M}})}{\mathtt{trait\ T\ imports}\ \overline{\mathtt{TA}}\ \{\overline{\mathtt{M}}\}\ \mathtt{OK\ IN\ C\ except}\ meth(\overline{\mathtt{N}}) \cup (\overline{\mathtt{m}} \setminus meth(\overline{\mathtt{M}}))}\ (\text{Tr·Ok})$$

$$\frac{\begin{array}{ll} altlook(\mathtt{n}, \mathtt{TA\ with}\ \{\mathtt{m@n}\}) = \mathtt{M} & \mathtt{M\ OK\ IN\ C} \\ \mathtt{TA\ OK\ IN\ C\ except}\ \overline{\mathtt{p}} & \mathtt{n} \notin meth(\mathtt{TA}) \end{array}}{\mathtt{TA\ with}\ \{\mathtt{m@n}\}\ \mathtt{OK\ IN\ C\ except}\ \overline{\mathtt{p}} \setminus \{\mathtt{m}\}}\ (\text{Alias·Ok}_1)$$

$$\frac{\begin{array}{ll} altlook(\mathtt{n}, \mathtt{TA\ with}\ \{\mathtt{m@n}\}) = \mathtt{M} & \neg\mathtt{M\ OK\ IN\ C} \\ \mathtt{TA\ OK\ IN\ C\ except}\ \overline{\mathtt{p}} & \mathtt{n} \notin meth(\mathtt{TA}) \end{array}}{\mathtt{TA\ with}\ \{\mathtt{m@n}\}\ \mathtt{OK\ IN\ C\ except}\ (\overline{\mathtt{p}} \setminus \{\mathtt{m}\}) \cup \{\mathtt{n}\}}\ (\text{Alias·Ok}_2)$$

$$\frac{\mathtt{TA\ OK\ IN\ C\ except}\ \ \overline{\mathtt{n}} \qquad \mathtt{m} \in meth(\mathtt{TA})}{\mathtt{TA\ minus}\ \{\mathtt{m}\}\ \mathtt{OK\ IN\ C\ except}\ \ \overline{\mathtt{n}} \setminus \{\mathtt{m}\}}\ (\text{Exlude·Ok})$$

Some comments are in order

- (Tr·Ok) Ensures that every $\mathtt{TA} \in \overline{\mathtt{TA}}$ is well-typed. Intuitively
  - We fetch all the methods $\overline{\mathtt{N}}$ defined in the trait `T` that are not type-checked in `C`.
  - We type-check the set of altered traits $\overline{\mathtt{TA}}$ in `C`, producing a set of illegal methods (the $\mathtt{except}\ \overline{\mathtt{m}}$ part) corresponding to the methods of $\overline{\mathtt{TA}}$ that do not type-check in `C`.
  - We check the key condition $\cap\overline{\mathtt{TA}} \setminus \diamond\overline{\mathtt{TA}} \subseteq meth(\overline{\mathtt{M}})$, ensuring that every conflict is resolved, and guaranteeing that the lookup algorithm provides the correct conflict resolution.
  - We build a new set of illegal methods for `T` with respect to `C`, that is, $meth(\overline{\mathtt{N}}) \cup (\overline{\mathtt{m}} \setminus meth(\overline{\mathtt{M}}))$, that is, the illegal methods of `TA` ($meth(\overline{\mathtt{N}})$) plus the non-overridden illegal methods from $\overline{\mathtt{TA}}$ ($\overline{\mathtt{m}} \setminus meth(\overline{\mathtt{M}})$).
- (Alias·Ok$_1$) Ensures that if `TA` is well typed, and the body `M` of the aliased method is well typed in `C`, and the new name `n` is fresh in `TA`, then the altered trait is well typed. The new set of illegal methods is the set $\overline{\mathtt{p}}$ less `m` (former method name).
- (Alias·Ok$_2$) Behaves as for (Alias·Ok$_1$), except that `M` is not well typed and `n` is added to the set of illegal methods.
- (Exlude·Ok) If `TA` is well typed, then excluding `m` just removes `m` from the set of illegal methods.

```
trait T1 {   int m(){return 1;}}
trait T2 {  bool m(){return true;}}
trait T3 imports T1 T2
        {String m(){return ‘‘hello world’’;}                    m is the winner method
         String n(){return this.m();}}                          n will call the winner m
class A extends Object imports T3
        {;A(){super();}}             m of T3 is the winner, and n of T3 will call m of T3
class B extends Object imports T1 T2
        {;B(){super();}
         String m(){return ‘‘how are you?’’;}                   m is the winner method
         String n(){return this.m();}}                     n of T3 will call the winner m

(new A()).n()                                                   return ‘‘hello world’’
(new B()).n()                                                   return ‘‘how are you?’’
```

Fig. 2. Blocked code.

Remark 5.1. *(Why not simply* `TA OK IN C`*?)* The reader may argue that a simpler set for type-checking traits would be more appropriate, namely

$$\frac{\overline{\mathtt{M}}\ \mathtt{OK\ IN\ C}\qquad \cap\overline{\mathtt{TA}}\setminus\diamond\overline{\mathtt{TA}}\subseteq \mathit{meth}(\overline{\mathtt{M}})\qquad \overline{\mathtt{TA}}\ \mathtt{OK\ IN\ C}}{\mathtt{trait\ T\ imports}\ \overline{\mathtt{TA}}\ \{\overline{\mathtt{M}}\}\ \mathtt{OK\ IN\ C}}\ (\mathrm{Tr{\cdot}Ok'})$$

plus rules (Alias·Ok$_1$), and (Exlude·Ok) (without the `except` part). This set of rules enforces the well-known restriction saying that overriding a method in a trait is possible only when the overridden method has the same type-interface. In this case, the `except` part is empty. However, this restriction blocks the legal code of Figure 2. Now the following two questions arise

—*Are the above rules sound?* Yes they are: but they block the legal code of Figure 2.

—*Are the* FTJ *rules sound?* Yes they are (see Section 6). Intuitively, (i) all methods defined or inherited in traits (respectively, in classes) are type-checked all at once except for the illegal methods that are fetched and not type-checked (the `except` part), and (ii) the lookup algorithm hide the (badly typed) methods that are overridden in trait alterations or in classes. As such, bodies of method `m` in traits `T1` and `T2` above are not type-checked and hence overridden by two new bodies of type `String` in trait `T3` and class `B`. This is one of the great achievements of the FTJ's type system.

Class Checking. The FTJ's type system culminates in the class checking rule

$$\frac{\begin{array}{lll}\mathtt{K}=\mathtt{C}(\overline{\mathtt{D}}\ \overline{\mathtt{g}},\overline{\mathtt{C}}\ \overline{\mathtt{f}})\{\mathtt{super}(\overline{\mathtt{g}});\mathtt{this}.\overline{\mathtt{f}}=\overline{\mathtt{f}};\} & & \\ \mathit{fields}(\mathtt{D})=\overline{\mathtt{D}}\ \overline{\mathtt{g}} & & \cap\overline{\mathtt{TA}}\setminus\diamond\overline{\mathtt{TA}}\subseteq\mathit{meth}(\overline{\mathtt{M}})\\ \overline{\mathtt{M}}\ \mathtt{OK\ IN\ C} & \overline{\mathtt{TA}}\ \mathtt{OK\ IN\ C\ except}\ \overline{\mathtt{m}} & \overline{\mathtt{m}}\subseteq\mathit{meth}(\overline{\mathtt{M}})\end{array}}{\mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\mathtt{K}\ \overline{\mathtt{M}}\ \overline{\mathtt{TA}}\}\ \mathtt{OK}}\ (\mathrm{Cla{\cdot}Ok})$$

Intuitively, this rule checks that all the components of the class are well typed, and that all conflicts are resolved. This type-checking rule ensures that FTJ is a proper extension of FJ, thanks to both occurrences in the premises of the $\subseteq$ symbol that ensures compatibility whenever $\overline{\mathtt{TA}}$ is empty. More precisely

- —We fetch the constructor `K` and the fields $\overline{\mathtt{g}}$.
- —We check the key condition $\cap\overline{\mathtt{TA}} \setminus \diamond\overline{\mathtt{TA}} \subseteq \mathit{meth}(\overline{\mathtt{M}})$, ensuring that every conflict is resolved (see the explanation about (Tr·Ok) above).
- —We type-check all methods $\overline{\mathtt{M}}$ defined in `C`.
- —We type-check the set of altered traits $\overline{\mathtt{TA}}$ in `C`, producing a set of illegal methods $\overline{\mathtt{m}}$ (the `except` part), that is, the methods of $\overline{\mathtt{TA}}$ which do not type-check in `C`.
- —We check the condition $\overline{\mathtt{m}} \subseteq \mathit{meth}(\overline{\mathtt{M}})$, ensuring that the $\overline{\mathtt{m}}$ illegal methods are overridden with methods $\overline{\mathtt{M}}$ of the class `C`.

Method-Type Lookup and Valid Method Overriding. These rules have no difficulties (see Appendix B).

6. PROPERTIES

Once the operational semantics and the type system are defined, the next step is to prove that (i) the static semantics matches the dynamic one, that is, types are preserved during computation (modulo subtyping), that (ii) the interpreter cannot get stuck if programs only include upcasts, and finally that (iii) the type system prevents compiled programs from the unfortunate run-time *message-not-understood* error.

The result we obtain in designing FTJ is that adding trait inheritance to FJ does not break the elegance of the semantics of FJ nor does it make the meta-theory excessively complicated. Of course, some care must be devoted when dealing with trait alterations. Full proofs of the theorems are provided in Appendix C.

The Conflict Resolution Theorem proves that the conflicts are resolved for well-typed programs. The conflicts are, mathematically, the sources of non-determinism in the lookup algorithm - specifically in *tlook*. The theorem states that there is none.

THEOREM 6.1 (CONFLICT RESOLUTION). *If, for all* $\mathtt{C_i} \in \mathtt{CL}$*, we have* $\mathtt{C_i}$ `OK`*, then both* mbody *and* mytype *are functions.*

Subject reduction proves that if an expression is typable and reduces to another expression, then the latter expression is typable too has a type which is a subtype of the type of the former.

THEOREM 6.2 (SUBJECT REDUCTION). *If* $\Gamma \vdash \mathtt{e} \in \mathtt{C}$ *and* $\mathtt{e} \longrightarrow \mathtt{e}'$*, then* $\Gamma \vdash \mathtt{e}' \in \mathtt{D}$*, for some* $\mathtt{D} <: \mathtt{C}$.

Then, progress shows that the only way for the interpreter to get stuck is by reaching a state where a downcast is impossible. Let # means cardinality, as in Igarashi et al. [2001].

THEOREM 6.3 (PROGRESS). *Suppose* `e` *is a well-typed expression*

```
T1 {int m(){return this.f+1;}
    int n(){return this.f*10;}
    int p(){return this.q()+10;}}
T2 {int m(){return this.f+2;}
    int n(){return this.f*20;}
    int q(){return this.m()+this.n();}}
T3 {int m(){return this.f+3;}
    int n(){return this.f*30;}}
class A extends Object imports T1 minus {m} T2 minus {n}
      {int f; A(int f){super();this.f=f;}}
class B extends Object imports T1 minus {n} T2 minus {m}
      {int f; B(int f){super();this.f=f;}}
class C extends Object imports T1 with {m@m_T1} T2 with {n@n_T2}
      {int f; C(int f){super();this.f=f;}
       int m(){return this.m_T1()+this.n_T2()}
       int n(){return this.m_T1()*this.n_T2()}}
class D extends A imports T3
      {; C(int f) {super(f);}
       int p(){return this.q()+100;}
       int q(){return this.m()*this.n();}}
```

receiver	this.m()	this.n()	this.p()	this.q()
(new A(3))	5	30	45	35
(new B(3))	4	60	74	64
(new C(3))	64	240	314	304
(new D(3))	6	90	640	540

Fig. 3. Synthetic example.

—*If* e *includes* new C($\overline{\mathtt{e}}$).f *as a subexpression, then* fields(C)=$\overline{\mathtt{T}}\ \overline{\mathtt{f}}$ *and* $\mathtt{f} \in \overline{\mathtt{f}}$*;*

—*If* e *includes* new C($\overline{\mathtt{e}}$).m($\overline{\mathtt{f}}$) *as a subexpression, then* mbody(m, C) $= (\overline{\mathtt{x}}, \mathtt{e}_0)$ *and* $\#(\overline{\mathtt{x}}) = \#(\overline{\mathtt{d}})$.

In accordance with FJ, we define the notion of *safe* expression e in Γ if the type derivation of the underlying (CT, TT) and $\Gamma \vdash \mathtt{e} \in \mathtt{C}$ contains no downcast or *stupid cast* (rules (Typ·DCast), and (Typ·SCast)). Recall that a stupid cast in FJ is needed to ensure the Subject Reduction Theorem. Then, we show that our semantics transforms safe expressions to safe expressions, and, moreover, type-casts in a safe expression will never fail.

THEOREM 6.4 (REDUCTION PRESERVES SAFETY). *If* e *is safe in* Γ*, and* $\mathtt{e} \longrightarrow \mathtt{e}'$*, then* $\mathtt{e}'$ *is safe in* Γ.

THEOREM 6.5 (PROGRESS OF SAFE PROGRAMS). *Suppose* e *is safe in* Γ. *If* e *has* (C)new D($\overline{\mathtt{e}}$) *as a subexpression, then* D <: C.

7. EXAMPLES

7.1 Synthetic Example à la Goldberg and Robson [1983]

The example in Figure 3 defines three simple traits and four classes that import those traits, some of them altered. The table summarizes all possible method calls (we assume types and algebras for integers).

```
trait Freedom    {Independence declaration(){return ...;}
                  Human_Rights  acclamation(){return ...;}
                  Food        freedom_food(){return ``Turkey'';}}
trait Democratic imports Freedom
                 {Kerry program(){return ...;}}
trait Republican imports Freedom
                 {Bush program(){return ...;}
                  Food freedom_food(){return ``Freedom_fries'';}}
trait Outsider   imports Democratic with {program@program_demo}
                 {Chirac program(){return ...;}
                  Food freedom_food(){return ``French_fries'';}}


class One_Candidate extends Object       class Two_Candidate extends Object
                   imports Democratic                        imports Republican
{...;                                    {...;
 One_Candidate(){super();}                Two_Candidate(){super();}
 Object merge(){return ...                Object  merge(){return ...
               this.declaration() ...                    this.declaration() ...
               this.acclamation() ...                    this.acclamation() ...
               this.program() ...                        this.program() ...
               this.freedom_food() ...                   this.freedom_food() ...
               this.program_demo();}                     this.program_repub();}
 Object   ask(){return this.merge();}}    Object    ask(){return this.merge();}}

class Three_Candidate extends Object     class Four_Candidate extends Object
                     imports Outsider                          imports
                                         Democratic with {program@program_demo}
                                         Republican with {program@program_repub}}
{...;                                    {...;
 Three_Candidate(){super();}              Four_Candidate(){super();}
 Object merge(){return ...                Object  merge(){return ...
               this.declaration() ...                    this.declaration() ...
               this.acclamation() ...                    this.acclamation() ...
               this.program() ...                        this.program() ...
               this.program_demo();}                     this.program_demo();
 Object  ask(){return this.merge();}}                    this.program_repub();}
                                          Object       ask(){return this.merge();}
                                          Blair    program(){return ...;}
                                          Food freedom_food(){return``Fish&Chips'';}}

(new   One_Candidate()).ask()             a merge of Freedom, Kerry program, and Turkey
(new   Two_Candidate()).ask()         a merge of Freedom, Bush program, and Freedom fries
(new Three_Candidate()).ask()  a merge of Freedom, Kerry and Chirac programs, and French fries
(new  Four_Candidate()).ask()   a ``strange'' merge of Freedom, Blair, Bush, and Kerry programs
                                                                      and Fish and Chips ...
```

Fig. 4. Funny example.

7.2 Funny Example à la FTJ

The example in Figure 4 shows how traits do not break legal code.[5] The ability to compose traits containing the same method name and different,

---

[5] Disclaimer: The names used here are chosen only for the purpose to show the power of combining traits with different types. There is absolutely no political message inside.

incompatible, signatures is one of the key results of this article. The ability also to detect and type-check innocuous methods inherited via a diamond is another achievement of FTJ. Roughly speaking, this corresponds to accept all "safe" Smalltalk-like trait based feature that would not raise exceptions of the shape *message-not-understood* at run-time.

8. RELATED WORK

In the past few years, many proposals for languages with typed traits have emerged. The first paper about trait inheritance in statically typed languages is the one by Fisher and Reppy [2004], presenting a core calculus (hereafter called TcoreMoby) featuring traits for the programming language Moby. Quitslund [2004] presented the first implementation of traits in a Java setting. Odersky's SCALA language [Scala Team 2007] features an interesting implementation of typed traits. Recently, Smith and Drossopoulou [2005] formally adds traits to Java [2005] (thereafter called Chai). Nierstrasz et al. [2006] formalize an untyped mechanism to compile FTJ into plain Java by exploiting the *flattening* property of traits. Finally, the new language Fortress by Allen et al. [2007] also feature traits-as-types. We shortly review these proposals and compare it with FTJ.

(TcoreMoby). It adds statically typed trait-based inheritance to an object-based calculus with first-class functions of the ML family. Fisher and Reppy [2004] have the same interest in typed traits as we do, and historically this paper can be considered as the first attempt to type-check statically traits. The key points of TcoreMoby are that (a) two traits can be combined only if they are disjoint, and that (b) one method can be overridden by another only if it has the same type interface, and that (c) in TcoreMoby traits can be type-checked only once. The paper comes with the full set of proofs. Our FTJ relaxes point (a) and (b), and leaves point (c) for further work (see Liquori and Spiwack [2008]).

(Chai). It adds statically-typed trait-based inheritance to Java; in fact there are three dialects defined: $\mathsf{Chai}_{1,2,3}$. As for TcoreMoby, the key points in Chai are that (a) two traits can be combined only if they are disjoint, and that (b) one method can be overridden by another only if it has the same type interface, and that (c) in $\mathsf{Chai}_{2,3}$ traits can be type-checked only once, and that (d) in Chai_3 traits can be substituted for one another dynamically. The paper comes with proof sketches for the theorems of Chai_1, and soundness theorems for $\mathsf{Chai}_{2,3}$, whose proofs are not yet published. Our FTJ can be compared with Chai_1: the biggest difference is that FTJ relaxes point (a) and (b), making the type system more expressive than Chai_1. Moreover, FTJ comes with a full metatheory. Point (c) is left for further work (see Liquori and Spiwack [2008]).

(Scala). It features traits as specific instance of an abstract class; thus the abstract modifier is redundant for it. Traits in Scala are a bit like interfaces in ClassicJava [Flatt et al. 1998], since they are used to define object types by specifying the signature of the supported methods. Besides in Scala the composition order of trait is irrelevant. A solid implementation is available on the Scala web site. A Featherweight Scala formal model with related meta-theory remains to be fleshed out.

(Fortress). The language specifications was published on the SUN's web site at the end of 2005. This language features traits-as-types (i.e. a trait is like an interface in Java with some concrete method bodies inside), and objects are trait instances, obtained by completing the imported trait by the body declaration of the abstract methods. A formal model with related meta-theory remains to be fleshed out.

We compare below some of the above proposals on typed traits having a formal static and dynamic semantics.

(1) TcoreMoby is a core calculus for languages of the ML family, whereas FTJ and Chai are core calculi for Java-like languages (Java is a notable example, not the absolute target).

(2) TcoreMoby and Chai are imperative, that is, they have a notion of state and store, whereas FTJ is purely functional.

(3) TcoreMoby allows one to define a method only inside a trait definition, whereas FTJ and Chai allows one also to define method in class definitions: methods defined in classes take precedence over methods defined in traits.

(4) TcoreMoby and Chai allow trait compositions only if the traits to be composed are disjoint, that is, no common methods, whereas FTJ permits one to compose traits even if they share common methods: in this case all common methods must be overridden in the trait itself to be disambiguated.

(5) TcoreMoby considers method override inside traits as a derived operation, whereas FTJ considers it as native; methods defined inside a trait take precedence over methods imported the trait itself.

(6) TcoreMoby aliases a method `m` with `n` by copying the body of `m` and associating to the method name `n`, and FTJ also does. Moreover, see Remark 4.2, a sound variant of FTJ could alias `m` in `n` and replace in the body of `m` every occurrence of `this.m` by `this.n`; in both cases, `m` will be is removed from the illegal methods.

(7) TcoreMoby evaluates traits to trait values (this correspond to a linking phase), whereas FTJ only type-checks a trait with respect to a class that imports that trait.

(8) TcoreMoby and Chai feature `this` and `super`, whereas FTJ supports only `this`.

(9) TcoreMoby has a type-system that features polymorphic-types and polymorphic-traits, whereas that of FTJ type-system features only first-order types.

(10) TcoreMoby and Chai subtype system features width subtyping, and FTJ also does.

(11) TcoreMoby does not have constructors, whereas FTJ and Chai do.

(12) TcoreMoby and Chai$_2$ type-check traits only once with a special type that keeps tracks of the *required* and the *provided* methods, whereas FTJ type-checks a trait only inside a class `C`, recording the methods that are

illegal (the `except` part of the trait typing judgment). Illegal methods can be overridden with a complete different type, provided that all methods that refers to those methods will continue to type-check it in the class `C`. TcoreMoby and $\text{Chai}_{1,2,3}$ are unable to type-check the examples of Section 7. This is because both proposals enforce the constraint that overriding a method in a trait is possible only when we respect the same type-interface as the overridden one, as explained in Remark 2. In contrast, the FTJ type system relaxes this constraint and permits overriding a method in a trait with a different type-interface.

9. CONCLUSIONS

In this article, we have presented a formal development of the theory of FTJ, a statically typed, purely functional, class-based language featuring classes, objects, and trait inheritance. Among the possible future directions, we list some questions on our agenda.

- The type system presented allows one to type-check traits only within classes; in fact, when typing a class, all requirements of all traits (the `except` part) must be resolved inside the class itself, otherwise the created instances would generate a *message-not-understood* upon some non-implemented message send. Type-checking traits only once is a reasonable feature to be added as in TcoreMoby and Chai_2. This suggests extending our type system for FTJ with *ad hoc* types for traits. Here, traits can be type-checked only once and considered as regular types, as Java-like interfaces with precise behavior. We are developing two solutions for that: a simpler and a more complex one. In the simpler solution, Liquori and Spiwack [2008], a trait can be seen either as a potentially incomplete class where objects can be assigned but not instantiated, that is, as an interface with some behavior inside but no state: this extension can be achieved by adding and modifying few rules in the FTJ type system. Another more complete solution would consider traits as potentially incomplete units of behaviors, where objects can be assigned and partially evaluated: to do this it could be encouraging to start from of a previous work on *potentially incomplete objects* [Bono et al. 1997] in an object-based setting. Other hints can be found in the theory of modules and mixins [Hirschowitz et al. 2004], and in the linking phase of Fisher and Reppy [2004].
- It would be interesting to add bounded polymorphic-types or even generic-types; those extension will greatly improve the usefulness of statically typed traits.
- It would be interesting to extend FTJ with *imperative features*.
- We would like to explore the impact of trait inheritance for the language C#; although this language is quite similar to Java, it has its peculiarities, which should be carefully interleaved and kept compatible with typed traits.
- We would like to compare the Fortress lookup algorithm with the FTJ lookup algorithm.

—Finally, it is our opinion that trait-based inheritance could be fruitfully applied to Aspect-Oriented Programming (AOP) *à la* Kiczales et al. [1997], and Variation-Oriented Programming (VOP) *à la* Mezini [2002], and Object-based Programming (OBP) *à la* Self [Ungar and Smith 1987].

APPENDIX

A. SYNTAX AND SEMANTICS OF FTJ

Syntax

$$\begin{array}{llll}
\mathtt{CL} & ::= & \mathtt{class\ C\ extends\ C\ [imports\ \overline{TA}]\{\overline{C}\ \overline{f};K\ \overline{M}\}} & \text{Class Declarations} \\
\mathtt{TL} & ::= & \mathtt{trait\ T\ [imports\ \overline{TA}]\{\overline{M}\}} & \text{Trait Declarations} \\
\mathtt{TA} & ::= & \mathtt{T} \mid \mathtt{TA\ with\ \{m@m\}} \mid \mathtt{TA\ minus\ \{m\}} & \text{Trait Alterations} \\
\mathtt{K} & ::= & \mathtt{C(\overline{C}\ \overline{f})\{super(\overline{f});this.\overline{f} = \overline{f};\}} & \text{Constructors} \\
\mathtt{M} & ::= & \mathtt{C\ m(\overline{C}\ \overline{x})\{return\ e;\}} & \text{Methods} \\
\mathtt{e} & ::= & \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m(\overline{e})} \mid \mathtt{new\ C(\overline{e})} \mid \mathtt{(C)e} & \text{Expressions}
\end{array}$$

Subtyping

$$\frac{}{\mathtt{C} <: \mathtt{C}}\ (\text{Sub·Refl})$$

$$\frac{\mathtt{C} <: \mathtt{D} \quad \mathtt{D} <: \mathtt{E}}{\mathtt{C} <: \mathtt{E}}\ (\text{Sub·Trans})$$

$$\frac{\mathtt{CT(C)} = \mathtt{class\ C\ extends\ D}\ \{\ldots\}}{\mathtt{C} <: \mathtt{D}}\ (\text{Sub·Cla})$$

Small-step semantics

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{(\mathtt{new\ C(\overline{e})}).\mathtt{f_i} \longrightarrow \mathtt{e_i}}\ (\text{Run·Field})$$

$$\frac{\mathit{mbody}(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e_0})}{(\mathtt{new\ C(\overline{e})}).\mathtt{m}(\overline{\mathtt{d}}) \longrightarrow [\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new\ C(\overline{e})/this}]\mathtt{e_0}}\ (\text{Run·Call})$$

$$\frac{\mathtt{C} <: \mathtt{D}}{(\mathtt{D})(\mathtt{new\ C(\overline{e})}) \longrightarrow \mathtt{new\ C(\overline{e})}}\ (\text{Run·Cast})$$

Congruence

$$\frac{\mathtt{e} \longrightarrow \mathtt{e}'}{\mathtt{e.f} \longrightarrow \mathtt{e}'\mathtt{.f}}\ (\text{Cgr}\cdot\text{Field}) \qquad \frac{\mathtt{e} \longrightarrow \mathtt{e}'}{\mathtt{e.m}(\overline{\mathtt{e}}) \longrightarrow \mathtt{e}'\mathtt{.m}(\overline{\mathtt{e}})}\ (\text{Cgr}\cdot\text{Receiver})$$

$$\frac{\mathtt{e}_\mathtt{i} \longrightarrow \mathtt{e}'_\mathtt{i}}{\mathtt{e.m}(\ldots, \mathtt{e}_\mathtt{i}, \ldots) \longrightarrow \mathtt{e.m}(\ldots, \mathtt{e}'_\mathtt{i}, \ldots)}\ (\text{Cgr}\cdot\text{Args})$$

$$\frac{\mathtt{e}_\mathtt{i} \longrightarrow \mathtt{e}'_\mathtt{i}}{\mathtt{new}\ \mathtt{C}(\ldots, \mathtt{e}_\mathtt{i}, \ldots) \longrightarrow \mathtt{new}\ \mathtt{C}(\ldots, \mathtt{e}'_\mathtt{i}, \ldots)}\ (\text{Cgr}\cdot\text{New})$$

$$\frac{\mathtt{e} \longrightarrow \mathtt{e}'}{(\mathtt{C})\mathtt{e} \longrightarrow (\mathtt{C})\mathtt{e}'}\ (\text{Cgr}\cdot\text{Cast})$$

Field lookup

$$\frac{}{\mathit{fields}(\mathtt{Object}) = \bullet}\ (\text{Field}\cdot\text{Top})$$

$$\frac{\begin{array}{l}\mathtt{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{D}\ \mathtt{imports}\ \overline{\mathtt{TA}}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}}; \mathtt{K}\ \overline{\mathtt{M}}\} \\ \mathit{fields}(\mathtt{D}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}}\end{array}}{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}}, \overline{\mathtt{C}}\ \overline{\mathtt{f}}}\ (\text{Field}\cdot\text{Cla})$$

Method body lookup

$$\frac{\begin{array}{l}\mathtt{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{D}\ \mathtt{imports}\ \overline{\mathtt{TA}}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}}; \mathtt{K}\ \overline{\mathtt{M}}\} \\ \mathtt{B}\ \mathtt{m}\ (\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\mathtt{return}\ \mathtt{e};\} \in \overline{\mathtt{M}}\end{array}}{\mathit{mbody}(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e})}\ (\text{MBdy}\cdot\text{Cla})$$

$$\frac{\begin{array}{l}\mathtt{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{D}\ \mathtt{imports}\ \overline{\mathtt{TA}}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}}; \mathtt{K}\ \overline{\mathtt{M}}\} \\ \mathtt{m} \not\in \mathit{meth}(\overline{\mathtt{M}}) \quad \mathit{tlook}(\mathtt{m}, \overline{\mathtt{TA}}) = \mathtt{B}\ \mathtt{m}(\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\mathtt{return}\ \mathtt{e};\}\end{array}}{\mathit{mbody}(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e})}\ (\text{MBdy}\cdot\text{Tr})$$

$$\frac{\begin{array}{l}\mathtt{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{D}\ \mathtt{imports}\ \overline{\mathtt{TA}}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}}; \mathtt{K}\ \overline{\mathtt{M}}\} \\ \mathtt{m} \not\in \mathit{meth}(\overline{\mathtt{M}}) \qquad \mathit{tlook}(\mathtt{m}, \overline{\mathtt{TA}}) = \mathit{fail} \qquad \mathit{mbody}(\mathtt{m}, \mathtt{D}) = (\overline{\mathtt{x}}, \mathtt{e})_\perp\end{array}}{\mathit{mbody}(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e})_\perp}\ (\text{MBdy}\cdot\text{SCla})$$

Trait lookup

$$\frac{\exists \texttt{TA} \in \overline{\texttt{TA}}.\ \mathit{altlook}(\texttt{m}, \texttt{TA}) \neq \mathit{fail}}{\mathit{tlook}(\texttt{m}, \overline{\texttt{TA}}) = \mathit{altlook}(\texttt{m}, \texttt{TA})}\ (\text{Tr}\cdot\text{Ok})$$

$$\frac{\forall \texttt{TA} \in \overline{\texttt{TA}}.\ \mathit{altlook}(\texttt{m}, \texttt{TA}) = \mathit{fail}}{\mathit{tlook}(\texttt{m}, \overline{\texttt{TA}}) = \mathit{fail}}\ (\text{Tr}\cdot\text{Ko})$$

Trait alteration lookup

$$\frac{\begin{array}{l}\texttt{TT(T)} = \texttt{trait T imports } \overline{\texttt{TA}}\ \{\overline{\texttt{M}}\}\\ \texttt{B m}(\overline{\texttt{B}}\ \overline{\texttt{x}})\{\texttt{return e;}\} \in \overline{\texttt{M}}\end{array}}{\mathit{altlook}(\texttt{m}, \texttt{T}) = \texttt{B m}(\overline{\texttt{B}}\ \overline{\texttt{x}})\{\texttt{return e;}\}}\ (\text{ATr}\cdot\text{Found})$$

$$\frac{\begin{array}{l}\texttt{TT(T)} = \texttt{trait T imports } \overline{\texttt{TA}}\ \{\overline{\texttt{M}}\}\\ \texttt{m} \notin \mathit{meth}(\overline{\texttt{M}}) \qquad \mathit{tlook}(\texttt{m}, \overline{\texttt{TA}}) = \texttt{M}_\perp\end{array}}{\mathit{altlook}(\texttt{m}, \texttt{T}) = \texttt{M}_\perp}\ (\text{ATr}\cdot\text{Inh})$$

$$\frac{\mathit{altlook}(\texttt{n}, \texttt{TA}) = \texttt{B n}(\overline{\texttt{B}}\ \overline{\texttt{x}})\{\texttt{return e;}\}}{\mathit{altlook}(\texttt{m}, \texttt{TA with }\{\texttt{n@m}\}) = \texttt{B m}(\overline{\texttt{B}}\ \overline{\texttt{x}})\{\texttt{return e;}\}}\ (\text{ATr}\cdot\text{Ali}_1)$$

$$\frac{\texttt{m} \neq \texttt{p} \qquad \texttt{m} \neq \texttt{q} \qquad \mathit{altlook}(\texttt{m}, \texttt{TA}) = \texttt{M}_\perp}{\mathit{altlook}(\texttt{m}, \texttt{TA with }\{\texttt{p@q}\}) = \texttt{M}_\perp}\ (\text{ATr}\cdot\text{Ali}_2)$$

$$\frac{\texttt{m} \neq \texttt{n}}{\mathit{altlook}(\texttt{m}, \texttt{TA with }\{\texttt{m@n}\}) = \mathit{fail}}\ (\text{ATr}\cdot\text{Ali}_3)$$

$$\frac{\mathit{altlook}(\texttt{n}, \texttt{TA}) = \mathit{fail}}{\mathit{altlook}(\texttt{m}, \texttt{TA with }\{\texttt{n@m}\}) = \mathit{fail}}\ (\text{ATr}\cdot\text{Ali}_4)$$

$$\frac{\texttt{m} \neq \texttt{n}}{\mathit{altlook}(\texttt{m}, \texttt{TA minus }\{\texttt{n}\}) = \mathit{altlook}(\texttt{m}, \texttt{TA})}\ (\text{ATr}\cdot\text{Exl}_1)$$

$$\frac{}{\mathit{altlook}(\texttt{m}, \texttt{TA minus }\{\texttt{m}\}) = \mathit{fail}}\ (\text{ATr}\cdot\text{Exl}_2)$$

Method names

$$\frac{}{\mathit{meth}(\mathtt{C}\ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\{\mathtt{return}\ \mathtt{e};\}) = \{\mathtt{m}\}}\ (\text{Mth}\cdot\text{Mth})$$

$$\frac{\mathtt{TT}(\mathtt{T}) = \mathtt{trait}\ \mathtt{T}\ \mathtt{imports}\ \overline{\mathtt{TA}}\ \{\overline{\mathtt{M}}\}}{\mathit{meth}(\mathtt{T}) = \mathit{meth}(\overline{\mathtt{M}}) \cup \mathit{meth}(\overline{\mathtt{TA}})}\ (\text{Mth}\cdot\text{Tr})$$

$$\frac{}{\mathit{meth}(\mathtt{TA}\ \mathtt{with}\ \{\mathtt{m}\,@\,\mathtt{n}\}) = (\mathit{meth}(\mathtt{TA}) \setminus \{\mathtt{m}\}) \cup \{\mathtt{n}\}}\ (\text{Mth}\cdot\text{Ali})$$

$$\frac{}{\mathit{meth}(\mathtt{TA}\ \mathtt{minus}\ \{\mathtt{m}\}) = \mathit{meth}(\mathtt{TA}) \setminus \{\mathtt{m}\}}\ (\text{Mth}\cdot\text{Exl})$$

Method paths in trait alterations

$$\frac{\mathtt{m}\ \in \mathit{meth}(\mathtt{TA})}{\mathtt{m}\ \mathsf{in}\ \mathtt{TA} \trianglelefteq \mathtt{m}\ \mathsf{in}\ \mathtt{TA}}\ (\text{Path}\cdot\text{Refl})$$

$$\frac{\mathtt{m}\ \mathsf{in}\ \mathtt{TA}_1 \trianglelefteq \mathtt{p}\ \mathsf{in}\ \mathtt{TA}_2 \quad \mathtt{p}\ \mathsf{in}\ \mathtt{TA}_2 \trianglelefteq \mathtt{n}\ \mathsf{in}\ \mathtt{TA}_3}{\mathtt{m}\ \mathsf{in}\ \mathtt{TA}_1 \trianglelefteq \mathtt{n}\ \mathsf{in}\ \mathtt{TA}_3}\ (\text{Path}\cdot\text{Trans})$$

$$\frac{\begin{array}{l}\mathtt{TT}(\mathtt{T}) = \mathtt{trait}\ \mathtt{T}\ \mathtt{imports}\ \overline{\mathtt{TA}}\ \{\overline{\mathtt{M}}\}\\ \mathtt{TA} \in \overline{\mathtt{TA}} \qquad \mathtt{m} \in \mathit{meth}(\mathtt{TA}) \setminus \mathit{meth}(\overline{\mathtt{M}})\end{array}}{\mathtt{m}\ \mathsf{in}\ \mathtt{T} \trianglelefteq \mathtt{m}\ \mathsf{in}\ \mathtt{TA}}\ (\text{Path}\cdot\text{Inh})$$

$$\frac{\mathtt{p}\ \mathsf{in}\ \mathtt{TA}_1 \trianglelefteq \mathtt{n}\ \mathsf{in}\ \mathtt{TA}_2}{\mathtt{m}\ \mathsf{in}\ \mathtt{TA}_1\ \mathtt{with}\ \{\mathtt{p}\,@\,\mathtt{m}\} \trianglelefteq \mathtt{n}\ \mathsf{in}\ \mathtt{TA}_2}\ (\text{Path}\cdot\text{Ali}_1)$$

$$\frac{\mathtt{m}\ \mathsf{in}\ \mathtt{TA}_1 \trianglelefteq \mathtt{n}\ \mathsf{in}\ \mathtt{TA}_2 \qquad \mathtt{m} \neq \mathtt{p} \qquad \mathtt{m} \neq \mathtt{q}}{\mathtt{m}\ \mathsf{in}\ \mathtt{TA}_1\ \mathtt{with}\ \{\mathtt{p}\,@\,\mathtt{q}\} \trianglelefteq \mathtt{n}\ \mathsf{in}\ \mathtt{TA}_2}\ (\text{Path}\cdot\text{Ali}_2)$$

$$\frac{\mathtt{m}\ \mathsf{in}\ \mathtt{TA}_1 \trianglelefteq \mathtt{n}\ \mathsf{in}\ \mathtt{TA}_2 \qquad \mathtt{m} \neq \mathtt{p}}{\mathtt{m}\ \mathsf{in}\ \mathtt{TA}_1\ \mathtt{minus}\ \{\mathtt{p}\} \trianglelefteq \mathtt{n}\ \mathsf{in}\ \mathtt{TA}_2}\ (\text{Path}\cdot\text{Exl})$$

B. THE TYPE SYSTEM OF FTJ

Method type lookup

$$\frac{\begin{array}{l}\mathtt{CT(C)} = \mathtt{class\ C\ extends\ D\ imports}\ \overline{\mathtt{TA}}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\mathtt{K}\ \overline{\mathtt{M}}\}\\ \mathtt{B\ m}(\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\mathtt{return\ e};\} \in \overline{\mathtt{M}}\end{array}}{\mathit{mtype}(\mathtt{m},\mathtt{C}) = \overline{\mathtt{B}} \to \mathtt{B}}\ \text{(MTyp·Self)}$$

$$\frac{\begin{array}{l}\mathtt{CT(C)} = \mathtt{class\ C\ extends\ D\ imports}\ \overline{\mathtt{TA}}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\mathtt{K}\ \overline{\mathtt{M}}\}\\ \mathtt{m} \notin \mathit{meth}(\overline{\mathtt{M}}) \quad \mathit{tlook}(\mathtt{m},\overline{\mathtt{TA}}) = \mathtt{B\ m}(\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\mathtt{return\ e};\}\end{array}}{\mathit{mtype}(\mathtt{m},\mathtt{C}) = \overline{\mathtt{B}} \to \mathtt{B}}\ \text{(MTyp·Tr)}$$

$$\frac{\begin{array}{l}\mathtt{CT(C)} = \mathtt{class\ C\ extends\ D\ imports}\ \overline{\mathtt{TA}}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\mathtt{K}\ \overline{\mathtt{M}}\}\\ \mathtt{m} \notin \mathit{meth}(\overline{\mathtt{M}}) \qquad \mathit{tlook}(\mathtt{m},\overline{\mathtt{TA}}) = \mathit{fail}\end{array}}{\mathit{mtype}(\mathtt{m},\mathtt{C}) = \mathit{mtype}(\mathtt{m},\mathtt{D})}\ \text{(MTyp·Super)}$$

Valid method overriding

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

Basic expression typing

$$\frac{}{\Gamma \vdash \mathtt{X} \in \Gamma(\mathtt{X})}\ \text{(Typ·Var)}$$

$$\frac{\begin{array}{ll}\Gamma \vdash \mathtt{e}_0 \in \mathtt{C}_0 & \mathit{mtype}(\mathtt{m},\mathtt{C}_0) = \overline{\mathtt{D}} \to \mathtt{C}\\ \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{C}} & \overline{\mathtt{C}} <: \overline{\mathtt{D}}\end{array}}{\Gamma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) \in \mathtt{C}}\ \text{(Typ·Call)}$$

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{f}} \quad \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{C}} \quad \overline{\mathtt{C}} <: \overline{\mathtt{D}}}{\Gamma \vdash \mathtt{new\ C}(\overline{\mathtt{e}}) \in \mathtt{C}}\ \text{(Typ·New)}$$

$$\frac{\Gamma \vdash \mathtt{e}_0 \in \mathtt{C}_0 \quad \mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{\Gamma \vdash \mathtt{e}_0.\mathtt{f}_\mathtt{i} \in \mathtt{C}_\mathtt{i}}\ \text{(Typ·Field)}$$

$$\frac{\Gamma \vdash \mathtt{e}_0 \in \mathtt{D} \quad \mathtt{D} <: \mathtt{C}}{\Gamma \vdash (\mathtt{C})\mathtt{e}_0 \in \mathtt{C}}\ (\text{Typ}\cdot\text{UCast}) \qquad \frac{\Gamma \vdash \mathtt{e}_0 \in \mathtt{D} \quad \mathtt{C} <: \mathtt{D} \quad \mathtt{C} \neq \mathtt{D}}{\Gamma \vdash (\mathtt{C})\mathtt{e}_0 \in \mathtt{C}}\ (\text{Typ}\cdot\text{DCast})$$

$$\frac{\textit{stupid warning} \quad \Gamma \vdash \mathtt{e}_0 \in \mathtt{D} \quad \mathtt{C} \not<: \mathtt{D} \quad \mathtt{D} \not<: \mathtt{C}}{\Gamma \vdash (\mathtt{C})\mathtt{e}_0 \in \mathtt{C}}\ (\text{Typ}\cdot\text{SCast})$$

Method Intersection and Diamond Detection

$$\cap\overline{\mathtt{TA}} \overset{\text{def}}{=} \{\mathtt{m} \mid \exists \mathtt{TA}_1 \neq \mathtt{TA}_2 \in \overline{\mathtt{TA}}.\ \mathtt{m} \in \mathit{meth}(\mathtt{TA}_1) \cap \mathit{meth}(\mathtt{TA}_2)\}$$

$$\diamond\overline{\mathtt{TA}} \overset{\text{def}}{=} \{\mathtt{m} \mid \exists \mathtt{n}, \mathtt{TA}_1.\forall \mathtt{TA}_2 \in \overline{\mathtt{TA}}.\ \mathtt{m} \in \mathit{meth}(\overline{\mathtt{TA}}) \implies \mathtt{m}\ \text{in}\ \mathtt{TA}_2 \trianglelefteq \mathtt{n}\ \text{in}\ \mathtt{TA}_1\}$$

Method typing

$$\frac{\begin{array}{l}\mathtt{CT(C)} = \mathtt{class\ C\ extends\ D\ imports}\ \overline{\mathtt{TA}}\ \{\ldots\} \\ \overline{\mathtt{x}}{:}\overline{\mathtt{C}}, \mathtt{this{:}C} \vdash \mathtt{e} \in \mathtt{F} \quad \mathit{override}(\mathtt{m}, \mathtt{D}, \overline{\mathtt{C}} \to \mathtt{E}) \quad \mathtt{F} <: \mathtt{E}\end{array}}{\mathtt{E\ m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\{\mathtt{return\ e};\}\ \mathtt{OK\ IN\ C}}\ (\text{Mth}\cdot\text{Ok}\cdot\text{Cla})$$

Trait typing

$$\frac{\overline{\mathtt{N}} \overset{\text{def}}{=} \{\mathtt{M} \in \overline{\mathtt{M}} \mid \neg\mathtt{M\ OK\ IN\ C}\} \quad \overline{\mathtt{TA}}\ \mathtt{OK\ IN\ C\ except}\ \overline{\mathtt{m}} \quad \cap\overline{\mathtt{TA}} \setminus \diamond\overline{\mathtt{TA}} \subseteq \mathit{meth}(\overline{\mathtt{M}})}{\mathtt{trait\ T\ imports}\ \overline{\mathtt{TA}}\ \{\overline{\mathtt{M}}\}\ \mathtt{OK\ IN\ C\ except}\ \mathit{meth}(\overline{\mathtt{N}}) \cup (\overline{\mathtt{m}} \setminus \mathit{meth}(\overline{\mathtt{M}}))}\ (\text{Tr}\cdot\text{Ok})$$

$$\frac{\begin{array}{ll}\mathit{altlook}(\mathtt{n}, \mathtt{TA\ with}\ \{\mathtt{m@n}\}) = \mathtt{M} & \mathtt{M\ OK\ IN\ C} \\ \mathtt{TA\ OK\ IN\ C\ except}\ \overline{\mathtt{p}} & \mathtt{n} \notin \mathit{meth}(\mathtt{TA})\end{array}}{\mathtt{TA\ with}\ \{\mathtt{m@n}\}\ \mathtt{OK\ IN\ C\ except}\ \overline{\mathtt{p}} \setminus \{\mathtt{m}\}}\ (\text{Alias}\cdot\text{Ok}_1)$$

$$\frac{\begin{array}{ll}\mathit{altlook}(\mathtt{n}, \mathtt{TA\ with}\ \{\mathtt{m@n}\}) = \mathtt{M} & \neg\mathtt{M\ OK\ IN\ C} \\ \mathtt{TA\ OK\ IN\ C\ except}\ \overline{\mathtt{p}} & \mathtt{n} \notin \mathit{meth}(\mathtt{TA})\end{array}}{\mathtt{TA\ with}\ \{\mathtt{m@n}\}\ \mathtt{OK\ IN\ C\ except}\ (\overline{\mathtt{p}} \setminus \{\mathtt{m}\}) \cup \{\mathtt{n}\}}\ (\text{Alias}\cdot\text{Ok}_2)$$

$$\frac{\mathtt{TA\ OK\ IN\ C\ except}\ \overline{\mathtt{n}} \quad \mathtt{m} \in \mathit{meth}(\mathtt{TA})}{\mathtt{TA\ minus}\ \{\mathtt{m}\}\ \mathtt{OK\ IN\ C\ except}\ \overline{\mathtt{n}} \setminus \{\mathtt{m}\}}\ (\text{Exlude}\cdot\text{Ok})$$

Class typing

$$\frac{\begin{array}{lll}\multicolumn{3}{l}{\mathtt{K} = \mathtt{C}(\overline{\mathtt{D}}\ \overline{\mathtt{g}}, \overline{\mathtt{C}}\ \overline{\mathtt{f}})\{\mathtt{super}(\overline{\mathtt{g}}); \mathtt{this}.\overline{\mathtt{f}} = \overline{\mathtt{f}};\}} \\ \mathit{fields}(\mathtt{D}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}} & & \cap\overline{\mathtt{TA}} \setminus \diamond\overline{\mathtt{TA}} \subseteq \mathit{meth}(\overline{\mathtt{M}}) \\ \overline{\mathtt{M}}\ \mathtt{OK\ IN\ C} & \overline{\mathtt{TA}}\ \mathtt{OK\ IN\ C\ except}\ \overline{\mathtt{m}} & \overline{\mathtt{m}} \subseteq \mathit{meth}(\overline{\mathtt{M}})\end{array}}{\mathtt{class\ C\ extends\ D\ imports}\ \overline{\mathtt{TA}}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}}; \mathtt{K}\ \overline{\mathtt{M}}\}\ \mathtt{OK}}\ (\text{Cla}\cdot\text{Ok})$$

C. THE FULL PROOFS

The following lemma proves that the method path relation only designs paths for existing methods.

LEMMA C.1 (NONVIRTUAL PATHS). *If* $\mathtt{m}$ in $\mathtt{TA}_1 \trianglelefteq \mathtt{n}$ in $\mathtt{TA}_2$*, then* $\mathtt{m} \in \mathit{meth}(\mathtt{TA}_1)$ *and* $\mathtt{n} \in \mathit{meth}(\mathtt{TA}_2)$.

PROOF. By induction on the derivation of $\mathtt{m}$ in $\mathtt{TA}_1 \trianglelefteq \mathtt{n}$ in $\mathtt{TA}_2$.

- (Path·Refl) Clear since $\mathtt{TA}_1 = \mathtt{TA}_2$.
- (Path·Trans) Straightforward by induction hypothesis.
- (Path·Inh) By hypothesis of the rule we have that $\mathtt{m} \in \mathit{meth}(\mathtt{TA}_2)$, and, by rule (Mth·Tr), $\mathtt{m} \in \mathit{meth}(\mathtt{TA}_1)$.
- (Path·Ali$_1$) By induction hypothesis we have that $\mathtt{m} \in \mathit{meth}(\mathtt{TA}_1)$ and $\mathtt{n} \in \mathit{meth}(\mathtt{TA}_2)$, and, by rule (Mth·Ali), we get $\mathtt{m} \in \mathit{meth}(\mathtt{TA}_1\ \mathtt{with}\ \{\mathtt{p@m}\})$.
- (Path·Ali$_2$) By induction hypothesis we have that $\mathtt{m} \in \mathit{meth}(\mathtt{TA}_1)$ and $\mathtt{n} \in \mathit{meth}(\mathtt{TA}_2)$, and, by rule (Mth·Ali), we get $\mathtt{m} \in \mathit{meth}(\mathtt{TA}_1\ \mathtt{with}\ \{\mathtt{p@q}\})$.
- (Path·Exl) By induction hypothesis we have that $\mathtt{m} \in \mathit{meth}(\mathtt{TA}_1)$ and $\mathtt{n} \in \mathit{meth}(\mathtt{TA}_2)$, and, by rule (Mth·Exl), we get $\mathtt{m} \in \mathit{meth}(\mathtt{TA}_1\ \mathtt{minus}\ \{\mathtt{p}\})$. □

The following lemma ensures that *altlook* provides a method with the proper name.

LEMMA C.2 (NAMING SOUNDNESS). *If* $\mathit{altlook}(\mathtt{m}, \mathtt{TA}) = \mathtt{M}_\perp$*, then either* $\mathtt{M} = \mathit{fail}$ *or* $\mathtt{M} = \mathtt{B}\ \mathtt{m}\ (\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\ldots\}$.

PROOF. By straightforward induction on the derivation of $\mathit{altlook}(\mathtt{m}, \mathtt{TA}) = \mathtt{M}$. □

The following lemma proves that a method path relation preserves the body of the method. It is the first step for proving determinism of well-typed programs.

LEMMA C.3 (DIAMOND PROTO-SOUNDNESS). *If* $\mathtt{m}$ in $\mathtt{TA}_1 \trianglelefteq \mathtt{n}$ in $\mathtt{TA}_2$*, then* $\mathit{altlook}(\mathtt{n}, \mathtt{TA}_1) = \mathtt{B}\ \mathtt{n}(\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\mathtt{return}\ \mathtt{e};\}$ *implies* $\mathit{altlook}(\mathtt{m}, \mathtt{TA}_2) = \mathtt{B}\ \mathtt{m}(\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\mathtt{return}\ \mathtt{e};\}$.

PROOF. By induction on the derivation of $\mathtt{m}$ in $\mathtt{TA}_1 \trianglelefteq \mathtt{n}$ in $\mathtt{TA}_2$.

- (Path·Refl) Clear since $\mathtt{TA}_1 = \mathtt{TA}_2$.
- (Path·Trans) Straightforward by induction hypothesis.
- (Path·Inh) Since $\mathtt{m} \notin \mathit{meth}(\overline{\mathtt{M}})$, the rule (ATr·Inh) can apply to $\mathtt{TA}_1$, which implies the result.
- (Path·Ali$_1$) The rule (ATr·Ali$_1$) (respectively (ATr·Ali$_4$)) can apply to $\mathtt{TA}_1$, which implies the result.
- (Path·Ali$_2$) Since $\mathtt{m} \neq \mathtt{p}$ and $\mathtt{m} \neq \mathtt{q}$, the rule (ATr·Ali$_2$) can apply to $\mathtt{TA}_1$, which implies the result.
- (Path·Exl) Since $\mathtt{m} \neq \mathtt{p}$, the rule (ATr·Exl$_1$) can apply to $\mathtt{TA}_1$, which implies the result. □

The following lemma proves that if a trait is well typed, then *meth* refers to the set of methods where *altlook* does not fail.

LEMMA C.4 (*meth* SOUNDNESS). *If* $\mathtt{TA\ OK\ IN\ C\ except}\ \overline{\mathtt{m}}$*, then* $\mathtt{m} \in \mathit{meth}(\mathtt{TA})$ *if and only if* $\mathit{altlook}(\mathtt{m}, \mathtt{TA}) \neq \mathit{fail}$.

PROOF. By induction on the derivation of $\mathit{altlook}(\mathtt{m}, \mathtt{TA})$.

- —(ATr·Found) Then $\mathit{altlook}(\mathtt{m}, \mathtt{TA}) \neq \mathit{fail}$ and from rule (Mth·Tr), we have $\mathtt{m} \in \mathit{meth}(\mathtt{TA})$.
- —(ATr·Inh) Then $\mathit{altlook}(\mathtt{m}, \mathtt{TA}) \neq \mathit{fail} \iff \exists \mathtt{TA_i} \in \overline{\mathtt{TA}}.\ \mathit{altlook}(\mathtt{m}, \mathtt{TA_i}) \neq \mathit{fail}$. Thus we have, by induction hypothesis
  $$\mathit{altlook}(\mathtt{m}, \mathtt{TA}) \neq \mathit{fail} \iff \exists \mathtt{TA_i} \in \overline{\mathtt{TA}}.\ \mathtt{m} \in \mathit{meth}(\mathtt{m}, \mathtt{TA_i}) \iff \mathtt{m} \in \mathit{meth}(\mathtt{TA}).$$
  The latter comes from rule (Mth·Tr) and the statement $\mathtt{m} \notin \mathit{meth}(\overline{\mathtt{M}})$.
- —(ATr·Ali$_1$) Then $\mathtt{TA} = \mathtt{TA_1}\ \mathtt{with}\ \{\mathtt{n@m}\}$. Since $\mathtt{TA}$ is well-typed, we have that $\mathtt{n} \in \mathit{meth}(\mathtt{TA_1})$ and $\mathtt{m} \notin \mathit{meth}(\mathtt{TA_1})$. Then, by induction hypothesis, we have that $\mathit{altlook}(\mathtt{n}, \mathtt{TA_1}) \neq \mathit{fail}$, and thus $\mathit{altlook}(\mathtt{m}, \mathtt{TA}) \neq \mathit{fail}$, and rule (Mth·Ali) states that $\mathtt{m} \in \mathit{meth}(\mathtt{TA})$.
- —(ATr·Ali$_2$) Straightforward as for (ATr·Ali$_1$).
- —(ATr·Ali$_3$) Clear.
- —(ATr·Ali$_4$) By induction hypothesis.
- —(ATr·Exl$_1$) Straightforward using rule (Mth·Exl).
- —(ATr·Exl$_2$) Clear. □

We prove that *altlook* is a function when the program type-checks.

LEMMA C.5 (CONFLICT RESOLUTION IN TRAIT ALTERATIONS). *If* $\mathtt{TA\ OK\ IN\ C\ except}\ \overline{\mathtt{m}}$*, then* $\mathit{altlook}(\,\cdot\,, \mathtt{TA})$ *is a function.*

PROOF. By induction on the derivation of *altlook*.

- —(ATr·Found) Direct.
- —(ATr·Inh) If $\mathit{tlook}(\mathtt{m}, \overline{\mathtt{TA}}) = \mathit{fail}$, then the property obviously holds. Else, by induction hypothesis, for all $\mathtt{TA_i} \in \overline{\mathtt{TA}}$, $\mathit{altlook}(\,\cdot\,, \mathtt{TA_i})$ is a function.
  - —If $\mathtt{m} \notin \cap\overline{\mathtt{TA}}$, then there is a unique $\mathtt{TA_i} \in \overline{\mathtt{TA}}$ where $\mathit{altlook}(\mathtt{m}, \mathtt{TA_i}) \neq \mathit{fail}$.
  - —If $\mathtt{m} \in \cap\overline{\mathtt{TA}}$, then since $\mathtt{TA}$ is well typed, the rule (Tr·Ok) enforces that $\mathtt{m} \in \diamond\overline{\mathtt{TA}}$. Then, for all $\mathtt{TA_i} \in \overline{\mathtt{TA}}$, we have $\mathtt{m} \in \mathit{meth}(\mathtt{TA_i}) \Rightarrow \mathtt{m}\ \mathsf{in}\ \mathtt{TA_i} \lessdot \mathtt{n}\ \mathsf{in}\ \mathtt{TA_1}$. Moreover, we know that $\mathtt{n} \in \mathit{meth}(\mathtt{TA_1})$, by Lemma C.1 (Nonvirtual Paths), which means that there is at least one $\mathit{altlook}(\mathtt{n}, \mathtt{TA_1}) = \mathtt{B\ n}\ (\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\ldots\}$ which is derivable, by Lemma C.4 (*meth* Soundness). Thus, $\mathit{altlook}(\mathtt{m}, \mathtt{TA_i}) = \mathtt{B\ m}\ (\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\ldots\}$ is derivable for all $\mathtt{TA_i}$ such that $\mathtt{m} \in \mathit{meth}(\mathtt{TA_i})$, by Lemma C.3 (Diamond Proto-Soundness). To conclude, we know that $\mathit{altlook}(\cdot, \mathtt{TA_i})$ is a function which ensures they are all equal.

  Which obviously gives the result.
- —(ATr·Ali$_1$) Then, $\mathtt{TA} = \mathtt{TA_1}\ \mathtt{with}\ \{\mathtt{n@m}\}$. The induction hypothesis ensures that $\mathit{altlook}(\,\cdot\,, \mathtt{TA_1})$ is a function. Then, it is straightforward.
- —(ATr·Ali$_2$) Straightforward as for (ATr·Ali$_1$).
- —(ATr·Ali$_3$) Clear.

- —(ATr·Ali$_4$) By induction hypothesis.
- —(ATr·Exl$_1$) Straightforward as for (ATr·Ali$_2$).
- —(ATr·Exl$_2$) Clear. □

The system is kept nondeterministic to emphasize the fact that the order of trait composition does not matter in the result. We prove that all conflict are resolved both for static (typing) and dynamic semantics.

THEOREM C.1 (CONFLICT RESOLUTION). *If for all* $\mathtt{C_i} \in \mathtt{CL}$*, we have* $\mathtt{C_i}$ $\mathtt{OK}$*, then both* $mbody(\,\cdot\,, \mathtt{C_i})$ *and* $mtype(\,\cdot\,, \mathtt{C_i})$ *are functions.* □

PROOF. We prove that $mbody(\,\cdot\,, \mathtt{C_i})$ is a function by induction on the derivation of $mbody(\mathtt{m}, \mathtt{C_i})$, the proof for $mtype(\,\cdot\,, \mathtt{C_i})$ being similar.

- —(MBdy·Cla) Direct.
- —(MBdy·SCla) Straightforward by induction hypothesis.
- —(MBdy·Tr) For all $\mathtt{TA_i} \in \overline{\mathtt{TA}}$, $altlook(\,\cdot\,, \mathtt{TA_i})$ is a function, by Lemma C.5 (Conflict Resolution in Trait Alterations).
  - ○ If $\mathtt{m} \notin \cap\overline{\mathtt{TA}}$ then, obviously, there is a unique $\mathtt{TA_i} \in \overline{\mathtt{TA}}$ where $altlook(\mathtt{m}, \mathtt{TA_i}) \neq fail$.
  - ○ If $\mathtt{m} \in \cap\overline{\mathtt{TA}}$, then, since $\mathtt{C_i}$ is well-typed, the rule (Cla·Ok) enforces that $\mathtt{m} \in \diamond\overline{\mathtt{TA}}$. Then, for all $\mathtt{TA_i} \in \overline{\mathtt{TA}}$, we have $\mathtt{m}$ in $\mathtt{TA_i} \lhd \mathtt{n}$ in $\mathtt{TA_1}$. Moreover, we know that $\mathtt{m} \in meth(\mathtt{TA_1})$, by Lemma C.1 (Nonvirtual Paths), which means that there is at least one $altlook(\mathtt{n}, \mathtt{TA_1}) = \mathtt{B}\ \mathtt{n}\ (\overline{\mathtt{B}}\ \bar{\mathtt{x}})\{\ldots.\}$ which is derivable, by Lemma C.4 (*meth* Soundness). Thus, $altlook(\mathtt{m}, \mathtt{TA_i}) = \mathtt{B}\ \mathtt{m}\ (\overline{\mathtt{B}}\ \bar{\mathtt{x}})\{\ldots.\}$ is derivable, by Lemma C.3 (Diamond Proto-Soundness). To conclude, we know that $altlook(\cdot, \mathtt{TA_i})$ is a function, which ensures they are all equal.

Which gives the result. □

In the following, we suppose that the classes are well typed, so that Theorem C.1 holds, and we can address *mbody* and *mtype* as mathematical functions. Moreover, unless explicitly mentioned, when citing proofs in Igarashi et al. [2001], we mean that they apply exactly for FTJ. From now on the lemma and theorem sequence is the same as in FJ.

LEMMA C.6 (*mtype* SOUNDNESS). *If* $mtype(\mathtt{m}, \mathtt{D}) = \overline{\mathtt{C}} \to \mathtt{E}$*, then* $mtype(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{C}} \to \mathtt{E}$*, for all* $\mathtt{C} <: \mathtt{D}$.

PROOF. The proof is as in Igarashi et al. [2001], by induction on the derivation of $\mathtt{C} <: \mathtt{D}$. □

LEMMA C.7 (SUBSTITUTION LEMMA). *If* $\Gamma, \bar{\mathtt{x}}{:}\overline{\mathtt{B}} \vdash \mathtt{e} \in \mathtt{D}$ *and* $\Gamma \vdash \bar{\mathtt{d}} \in \overline{\mathtt{A}}$*, where* $\overline{\mathtt{A}} <: \overline{\mathtt{B}}$*, then* $\Gamma \vdash [\bar{\mathtt{d}}/\bar{\mathtt{x}}]\mathtt{e} \in \mathtt{C}$ *for some* $\mathtt{C} <: \mathtt{D}$.

PROOF. The proof is as in Igarashi et al. [2001]. By straightforward induction on the derivation of $\Gamma, \bar{\mathtt{x}}{:}\overline{\mathtt{B}} \vdash \mathtt{e} \in \mathtt{D}$. □

LEMMA C.8 (WEAKENING). *If* $\Gamma \vdash \mathtt{e} \in \mathtt{C}$*, then* $\Gamma, \mathtt{X}{:}\mathtt{D} \vdash \mathtt{e} \in \mathtt{C}$.

PROOF. The proof is as in Igarashi et al. [2001]. By straightforward induction on the derivation of $\Gamma \vdash \mathtt{e} \in \mathtt{C}$. □

LEMMA C.9 (METHOD BODY TYPE). *If* $\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{B}} \to \mathtt{B}$ *and* $\mathit{mbody}(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e})$*, then, for some* $\mathtt{D}$ *with* $\mathtt{C} <: \mathtt{D}$*, there exists* $\mathtt{A} <: \mathtt{B}$ *such that* $\overline{\mathtt{x}}{:}\overline{\mathtt{B}}, \mathtt{this{:}D} \vdash \mathtt{e} \in \mathtt{A}$*.*

PROOF. By induction on the derivation of $\mathit{mbody}(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e})$. If $\mathit{mbody}(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e})$, then $\mathtt{B\ m\ (\overline{B}\ \overline{x})\{return\ e;\}\ OK\ IN\ D}$ for some $\mathtt{D}$ with $\mathtt{C} <: \mathtt{D}$ and some $\overline{\mathtt{B}} \to \mathtt{B}$. Then, by Lemma C.6 (*mtype* Soundness), $\overline{\mathtt{B}} \to \mathtt{B} = \mathit{mtype}(\mathtt{m}, \mathtt{C})$ holds.

- —(MBdy·Cla) Immediate from (Cla·Ok) rule.
- —(MBdy·Tr) Let $\mathtt{CT(C)} = \mathtt{class\ C\ extends\ D\ imports\ \overline{TA}\ \{\overline{C}\ \overline{f};K\ \overline{M}\}}$. By straightforward induction on the derivation of $\mathtt{TA\ OK\ IN\ C\ except\ \overline{m}}$, where $\mathtt{TA} \in \overline{\mathtt{TA}}$ is the trait alteration selected by *tlook*.
- —(MBdy·SCla) By induction hypothesis. □

Lastly, we are ready to prove the main theorems.

THEOREM C.2 (SUBJECT REDUCTION). *If* $\Gamma \vdash \mathtt{e} \in \mathtt{C}$ *and* $\mathtt{e} \longrightarrow \mathtt{e}'$*, then* $\Gamma \vdash \mathtt{e}' \in \mathtt{D}$*, for some* $\mathtt{D} <: \mathtt{C}$*.*

PROOF. The proof is as in Igarashi et al. [2001]. □

THEOREM C.3 (PROGRESS). *Suppose* $\mathtt{e}$ *is a well-typed expression.*

- —*If* $\mathtt{e}$ *includes* $\mathtt{new\ C(\overline{e}).f}$ *as a subexpression, then* $\mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$ *and* $\mathtt{f} \in \overline{\mathtt{f}}$*.*
- —*If* $\mathtt{e}$ *includes* $\mathtt{new\ C(\overline{e}).m(\overline{f})}$ *as a subexpression, then* $\mathit{mbody}(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e}_0)$ *and* $\#(\overline{\mathtt{x}}) = \#(\overline{\mathtt{d}})$.

PROOF. The proof is almost the same as in Igarashi et al. [2001]. There is a very little to add, just remember that we need Theorem C.1 (Conflict Resolution) to achieve the full proof. □

THEOREM C.4 (REDUCTION PRESERVES SAFETY). *If* $\mathtt{e}$ *is safe in* Γ*, and* $\mathtt{e} \longrightarrow \mathtt{e}'$*, then* $\mathtt{e}'$ *is safe in* Γ*.*

PROOF. As in Igarashi et al. [2001], this proof is just similar to the Subject Reduction proof. □

THEOREM C.5 (PROGRESS OF SAFE PROGRAMS). *Suppose* $\mathtt{e}$ *is safe in* Γ*. If* $\mathtt{e}$ *has* $\mathtt{(C)new\ D(\overline{e})}$ *as a subexpression, then* $\mathtt{D} <: \mathtt{C}$*.*

PROOF. The proof is almost the same as in Igarashi et al. [2001]. The only rule that we can apply to derive the type of $\mathtt{(C)new\ C_0(\overline{e})}$, if $\mathtt{e}$ is safe, is (Typ·UCast). □

ACKNOWLEDGMENTS

The authors are sincerely grateful to all members of the Software Composition Group in Bern, for their deep understanding of the untyped trait model and for several fruitful discussions. We also warmly thank all the anonymous referees for their insightful comments and suggestions that helped us to improve the article. Thanks to Benjamin Pierce for suggesting the name FeatherTrait Java. Thanks to Andrew Black for revising an earlier version of this article. We finally warmly thank Pierre Lescanne, Dan Dougherty, and Ralf Klasing for the careful reading of this article.

REFERENCES

ABADI, M. AND CARDELLI, L. 1996. *A Theory of Objects*. Springer Verlag, New York.

ALLEN, E., CHASE, D., LUCHANGCO, V., MAESSEN, J.-W., RYU, S., STEELE, G. L., AND TOBIN-HOCHSTADT, S. 2007. The Fortress Language Specification, version 1.0. `http://research.sun.com/projects/plrg/fortress.pdf`.

ALLEN, E. E., BANNET, J., AND CARTWRIGHT, R. 2003. Mixins in generic java are sound. Tech. rep., Rice University.

ANCONA, D., LAGORIO, G., AND ZUCCA, E. 2003. Jam - designing a Java extension with mixins. *ACM Trans. Prog. Lang. Syst. 25,* 5, 641–712.

ANCONA, D. AND ZUCCA, E. 2002a. A calculus of module system. *J. Funct. Program 12,* 12, 91–132.

ANCONA, D. AND ZUCCA, E. 2002b. A theory of mixin modules: Algebraic laws and reduction semantics. *Math. Struct. Comput. Sci. 12,* 5, 701–737.

BONO, V., BUGLIESI, M., DEZANI-CIANCAGLINI, M., AND LIQUORI, L. 1997. Subtyping constraint for incomplete objects. In *Proceedings of TAPSOFT/CAAP*. Lecture Notes in Computer Science, vol. 1214. Springer Verlag, New York, 465–477.

BONO, V., PATEL, A., AND SHMATIKOV, V. 1999. A core calculus of classes and mixins. In *Proceedings of ECOOP*. Lecture Notes in Computer Science, vol. 1628. Springer-Verlag, New York.

BRACHA, G. 1992. The Programming Language Jigsaw: Mixins, modularity and multiple inheritance. Ph.D. dissertation, University of Utah.

BRACHA, G. AND COOK, W. R. 1990. Mixin-based inheritance. In *Proceedings of OOPSLA/ECOOP*. SIGPLAN Notices, vol. 25(10). ACM, New York, 303–311.

CARDELLI, L. 1995. Obliq: A language with distributed scope. *Comput. Syst. 8,* 1, 27–59.

DI GIANANTONIO, P., HONSELL, F., AND LIQUORI, L. 1998. A lambda calculus of objects with self-inflicted extension. In *Proceedings of OOPSLA*. ACM, New York, 166–178.

DUCASSE, S., NIERSTRASZ, O., SCHÄRLI, N., WUYTS, R., AND BLACK, A. P. 2006. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst. 28,* 2, 331–388.

DUGGAN, D. AND SOURELIS, C. 1996. Mixin modules. In *Proceedings of ICFP*. SIGPLAN Notices, vol. 31(6). ACM, New York, 262–273.

FINDLER, R. B. AND FLATT, M. 1998. Modular object-oriented programming with units and mixins. In *Proceedings of ICFP*. SIGPLAN Notices. ACM, New York, 94–104.

FISHER, K. AND REPPY, J. 2004. Statically typed traits. `http://www.cs.uchicago.edu/files/tr_authentic/TR-2003-13.pdf`. (The early version *"A Typed Calculus of Traits"* has been presented at FOOL 10.)

FLATT, M. AND FELLEISEN, M. 1998. Units: Cool modules for HOT languages. In *Proceedings of PLDI*. SIGPLAN Notices. ACM, New York, 236–248.

FLATT, M., KRISHNAMURTHI, S., AND FELLEISEN, M. 1998. Classes and mixins. In *Proceedings of POPL*. ACM, New York, 171–183.

GOLDBERG, A. AND ROBSON, D. 1983. *Smalltalk-80: the Language and its Implementation*. Addison-Wesley, Reading, MA.

HIRSCHOWITZ, T., LEROY, X., AND WELLS, J. B. 2004. Call-by-value mixin modules: Reduction semantics, side effects, types. In *Proceedings of ESOP*. Lecture Notes in Computer Science, 2986, Springer-Verlag, New York, 64–78.

IGARASHI, A., PIERCE, B., AND WADLER, P. 2001. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Prog. Lang. Syst. 23,* 3, 396–450.

KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C. V., LOINGTIER, J.-M., AND IRWIN, J. 1997. Aspect-oriented programming. In *Proceedings of ECOOP*. Lecture Notes in Computer Science, 1241, Springer-Verlag, New York, 220–242.

LIQUORI, L. 1997. An extended theory of primitive objects: first order system. In *Proceedings of ECOOP*. Lecture Notes in Computer Science, 1241, Springer-Verlag, New York, 146–169.

LIQUORI, L. 1998. On object extension. In *Proceedings of ECOOP*. Lecture Notes in Computer Science, 1445, Springer-Verlag, New York, 498–552.

LIQUORI, L. AND SPIWACK, A. 2004. Featherweight-trait Java : A trait-based extension for FJ. Tech. Rep. RR-5247, INRIA. Juin. `http://www.inria.fr/rrrt/rr-5247.html`.

LIQUORI, L. AND SPIWACK, A. 2008. Extending FeatherTrait Java with interfaces. *Theoret. Comput. Sci.* To appear.

MEZINI, M. 2002. Towards variational object-oriented programming: The rondo model. Tech. Rep. TUD-ST-2002-02, Software Technology Group, Darmstadt University of Technology.

MICROSOFT. The C# Home Page. `http://msdn.microsoft.com/vcsharp/`.

MILNER, R., TOFTE, M., HARPER, R., AND MACQUEEN, D. 1997. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA.

MOBY TEAM. 2007. The Moby home page. `http://moby.cs.uchicago.edu/`.

NIERSTRASZ, O., DUCASSE, S., AND SCHÄRLI, N. 2006. Flattening traits. *J. Obj. Tech. 5,* 4, 129–148.

QUITSLUND, P. J. 2004. Java traits – Improving opportunities for reuse. Tech. Rep. CSE-04-005, OGI School of Science and Engineering. `http://www.ogi.edu/csee/tech-reports/2004/04-005.pdf`.

SCALA TEAM. 2007. The scala home page. `http://scala.epfl.ch/`.

SCHÄRLI, N., DUCASSE, S., NIERSTRASZ, O., AND BLACK, A. 2003. Traits: Composable units of behaviour. In *Proceedings of ECOOP*. Lecture Notes in Computer Science, 2743, Springer-Verlag, New York, 248–274.

SMITH, C. AND DROSSOPOULOU, S. 2005. Chai: Typed traits in Java. In *Proceedings of ECOOP*. Lecture Notes in Computer Science, 3586, Springer-Verlag, New York, 453–478.

SNYDER, A. 1987. Inheritance and the development of encapsulated software systems. In *Research Directions in Object-Oriented Programming*. MIT Press, Cambridge, MA, 165–188.

STROUSTRUP, B. 1997. *The C++ Programming Language, Ch. 15, Third Ed.* Addison Wesley, Reading, MA.

SUN. Java Technology. `http://java.sun.com/`.

UNGAR, D. AND SMITH, R. B. 1987. Self: The power of simplicity. In *Proceedings of OOPSLA*. ACM, New York, 227–241.

WELLS, J. B. AND VESTERGAARD, R. 2000. Equational reasoning for linking with first-class primitive modules. In *Proceedings of ESOP*. Lecture Notes in Computer Science, vol. 1782. Springer Verlag, New York, 412–428.

Received May 2005; revised June 2006; accepted February 2007