Featherweight-Trait Java: A Trait-based Extension for FJ

Luigi Liquori, Arnaud Spiwack

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Featherweight-Trait Java
A Trait-based Extension for FJ

Luigi Liquori — Arnaud Spiwack

N° 5247

Juin 2004

Thème SYM

ISSN 0249-6399 ISRN INRIA/RR--5247--FR+ENG

INRIA
SOPHIA ANTIPOLIS

Featherweight-Trait Java
A Trait-based Extension for FJ

Luigi Liquori*, Arnaud Spiwack †

Thème SYM — Systèmes symboliques
Projets Mirho

Rapport de recherche n° 5247 — Juin 2004 — 27 pages

Abstract: In the context of statically typed class-based languages, we investigate classes which might extend upon trait composition. Building classes by composing method-clusters is a well-known technique in implementing object- and class-based languages with simple inheritance, that has been explored in an untyped setting as a first-class mechanism available to the user only recently.

This paper presents Featherweight-Trait Java (FTJ), a conservative extension of Featherweight Java (FJ), a simple light-weight class-based calculus take off with statically typed traits. In FTJ, classes can be built using traits as a basic behavioral brick; every trait contains only behavior and no state. Method conflicts between traits must be resolved explicitly by the user either (1) by aliasing or excluding method names, or (2) overriding explicitly in the class. A special emphasis has been put on dealing with "diamond" conflict, a classical issue playing with multiple inheritance.

That makes a kind of multiple inheritance not only possible but simple. As such, FTJ is as a proper extension of FJ.

We present a new operational semantics with a new dynamic lookup algorithm, and a new sound type system that guarantees that evaluating a well-typed expression never yields a "message not understood" run-time error nor get the interpreter stuck. We give an example which illustrates the increased expressive power of the typed trait-based (multiple inheritance) model of FTJ with respect to the classical (single inheritance) typed class-based one of FJ. The resulting calculus appears to be a good starting point for a rigorous mathematical analysis of typed trait-based languages, and a novel paradigm for typed trait-oriented programming.

Key-words: Theoretical foundations, language design and implementation, multiple inheritance, types, traits, objects

* INRIA Sophia-Antipolis - Projet Mirho
† ENS Cachan

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Featherweight-Trait Java
une extension de FJ basée sur les traits

Résumé : Dans le contexte des langages typés statiquement et basés sur les classes, nous étudions un mécanisme pour étendre les classes par la composition de traits. Construire les classes en composant des ensembles de méthodes est une technique connue pour implémenter les langages basés sur les objets ou les classes avec héritage simple, qui a été étudiée, seulement récemment, comme méchanisme de première classe disponible à l'utilisateur dans un contexte non typé.

Ce papier présente Featherweight-Trait Java (FTJ), une extension conservative de Featherweight Java (FJ), un calcul simple et léger basé sur les classes auquel on rajoute des traits typés statiquement. Dans FTJ, les classes peuvent être construitent en utilisant les traits comme des briques basiques de comportements : les traits contiennent seulement des comportements, et pas d'états. Les conflit entre les méthodes doivent être résolues explicitement par l'utilisateur soit (1) en renomant ou en excluant la méthode, soit (2) en redéfinissant explicitement les méthodes concernées dans la classes. Un certain accent a été mis sur les "conflits en diamants", une question habituelle de l'héritage multiple.

Ce qui rend une sorte d'héritage multiple non seulement possible, mais facile.

Nous présentons une nouvelle sémantique opérationelle avec un nouvel algorithme de recherche, et un nouveau système de typage qui garanti que l'évaluation ne soulève pas une erreur "message pas compris", et que l'interpréteur ne se bloque pas. Le calcul proposé semble être un bon point de dépar pour une analyse mathématique rigoureuse de langages typés basés sur les traits, et un nouveau paradigm pour la programmation orientée trait.

Mots-clés : Fondements théoriques, implémentation et création de langages, héritage multiple, types, traits, objets

1 Introduction

"Inside every large language is a small language
struggling to get out ..." [IPW01]

"... 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 five kinds of inheritance, namely single inheritance, multiple inheritance, mixin inheritance, object-based inheritance, and trait inheritance. In a nutshell:

- **single inheritance** is the simplest model: a class can inherit methods and variables from its direct superclass; it is well known that this may force some code duplications;
- **multiple inheritance** is a richer but debated model: a class can inherit from many superclasses; conflicts between common methods in superclasses are usually resolved by the compiler. The literature presents a rich list of potential problems with multiple inheritance, such as fork-join inheritance, accessing overridden methods, complication of typechecking in presence of parametric classes;
- **mixin inheritance** is a model which takes place between the previous ones: a mixin is like a class that can be applied to various parent classes in order to extend them with the methods contained in the mixin itself. The model works quite well when we plug a single mixin, but composing mixins is a delicate operations though. It can break hierarchies, and that sometimes (with some notable exceptions) it depends on the way mixins are composed;
- **object-based inheritance**, also called *delegation-based inheritance*. 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. Any message sent to the object is handled directly by this object if it contains the corresponding method, or it is "passed back", *i.e. delegated*, to the prototype. Because of its extreme dynamicity, it is very difficult to typecheck it statically;
- **trait inheritance** has recently emerged as a novel model to build composable units of behaviors [SDNB03]. Intuitively a trait is just a cluster of methods, *i.e.* behavior without state. Subtraits can be built from a *unordered* list of supertraits, together with new methods declarations. Thus, traits are (incomplete) classes without state. Trait can be composed in any order. Traits are just behavior without state. A trait makes sense only when "imported" (used) by a class that, provides state variables, and possibly some additional methods to disambiguate conflict-names arising between the imported traits. The order for importing traits in classes is irrelevant. Traits have a

strong similarity with the inheritance model of CLOS [BDG+88], and the implementation of object-based inheritance in the Object-Calculus of Abadi-Cardelli's [AC96], at the basis of *e.g.* the programming language Obliq [Car95]

Goals of this Paper This paper extends the simple calculus of FJ with statically typed traits. The presentation is intentionally kept informal, with syntax and semantics of FTJ intentionally as close as possible to the one of FJ, with few definitions and few key theorems. The main aim of this paper is to introduce the typed trait-based inheritance in a simple typed object-oriented calculus based on (mutually recursive) class definition, object creation, field access, method invocation and override, method recursion through `this` and `super`, subtyping and simple casting. As for FJ, the features of Java that we *do not model* include assignment, interfaces, overloading, base types (`int`, `bool`, *etc*), `null` pointers, abstract method declaration, shadowing of superclass fields by subclass fields, access control (`public`, `private`, *etc*), and exceptions. Since FJ and FTJ provide no operations with side effect, a method body always consists of `return` followed by an expression.

The main contributions of this paper are thus:

- to present the calculus FTJ, a comfortable extension of FJ featuring typed trait inheritance; traits can be inherited via multiple inheritance, and conflict between common methods in two or more inherited traits must resolved *explicitly* by the user either (1) by aliasing or excluding method names, or (2) by overriding explicitly the conflicted methods in the class that imports those traits;
- to show how enforce a clean "separation of concerns" between traits and classes, the former intended as separate, recompilable, and Lego©-reusable units of behaviors, while the latter intended as "state container" pluggable with units of behavior (*i.e.* traits) plus few behaviors needed to explicitly solve conflicts in traits. Moreover, a class is the only entity devoted to generate runnable objects;
- to provide a simple type system that typechecks traits when imported in classes; this results in a sharp and lightweight extension of the type system of FJ that could be considered as a first-step to adding statically typed trait inheritance to the full Java language.

Road Map. The paper is structured as follows. In Section 2, we review the *untyped* trait-inheritance model and we present the main ideas underneath our *typed* trait 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 shows the type system of FTJ. Section 7 shows just the main metatheoretical results, and, finally, Section 8 conclude. A little knowledge of syntax, semantics and type system of FJ may be useful.

2 Traits inheritance

2.1 Untyped Traits inheritance

One nice feature of untyped *trait inheritance* model is when a conflict arises between traits included in the same class (*e.g.* a method `m` defined in two different traits), the conflict is signaled and it is up to the user to *explicitly* and *manually* resolve the conflict. Two simple rules can be easily implemented in the method-lookup algorithm for that purpose:

1. methods defined in class take precedence over those defined in traits;
2. methods defined in traits take precedence over those defined in superclasses.

It is a nice feature of the model. It greatly increases the flexibility. Indeed in the previous models dealing with multiple inheritance, some design choices are done *a priori* by the compiler and cannot be modified by the programmer.

Another nice property of trait inheritance is that a composite trait is equivalent to a *flattened* trait containing the same methods. Similarly, a class that imports traits is semantically equivalent to a class that defines *in situ* all the methods defined in traits.

By definition a trait is incomplete (but it can import from other traits). Hence it *requires* methods that are not defined in the trait, but however necessary in order to complete its behavior. In FTJ jargon:

```
trait T is
{m return (...this.p()...);          p is a this-required method
 n return (...this.q()...);          q is a this-required method
...                                  trait importing
}
```

Conflict Resolution. When dealing with multiple inheritance, conflict arises. Conflicts between traits must be solved *manually*, *i.e.* there is no special, rigid discipline to learn using traits. This has a strong similarity with the inheritance model of CLOS [BDG+88]. Once a conflict is detected (suppose a class `C` that imports two traits T_1, and T_2 defining the same method `m` with different behavior), there are essentially three ways to solve the conflict:

1. **Defining a new method `m` inside the class.** A new method `m` is redefined inside the class with an adequate behavior; the (trait-based) dynamic lookup algorithm will hide the conflict in traits in favor of the overridden method defined in the class[1]. In a FTJ jargon:

   ```
   class C extends Object
   {...;                  instance vars
     ...;                  constructor
   ```

[1] This may require some code duplication.

```
  D m(...){...}        new behavior for m
  T1;T2                each trait defines a (different) behavior for m
}
```

2. **Aliasing the method in traits.** The method `m` is aliased in T_1 and/or T_2 with new different names. A new behavior for `m` can be now given in the class `C` (possibly re-using the aliased methods `m_of_T1` and `m_of_T2` which are not anymore in conflict)[2]. In a FTJ jargon:

```
class C extends Objects
{
...;                   instance vars
...;                   constructor
D m(...){...};         new behavior for m (may use this.m_of_T1/2)
T1 with {m@m_of_T1};   trait T1 aliases method m with m_of_T1
T2 with {m@m_of_T2}    trait T2 aliases method m with m_of_T2
}
```

3. **Excluding method in traits.** One method `m` in traits T_1 or T_2 is excluded. This solves the conflict in favor of one trait or one class method and avoids some code duplication. In a FTJ-like jargon:

```
class C extends Objects
{
...;                   instance vars
...;                   constructor
T1;                    may contains method m
T2 minus {m}           method m is now hidden
}
```

Diamond problem. When dealing with multiple inheritance and conflicts, the diamond problem always appears. In our trait based inheritance, we can define it as follows: let T_0 be a trait with a method `m`, and T_1 and T_2 be two traits that inherit a method `m` from T_0. Then, a trait or class that uses both T_1 and T_2 would appear to have a conflict within the method `m`, however, the conflict seems harmless since both methods `m` are identical. Snyder [Sny87] suggest not to bother with diamonds and consider it a conflict anyway. We chose to detect as many diamond conflicts as possible and to consider it legal, and we prove it safe.

2.2 Typed Trait Inheritance

Trait inheritance were introduced and shown useful in a untyped object-oriented framework; it is natural to ask whether this inheritance model is fruitful in a statically typed framework

[2] This may avoid code duplication.

Syntax

CL	::=	`class C extends C` $\{\overline{\texttt{C}}\ \overline{\texttt{f}}; \texttt{K}\ \overline{\texttt{M}}\ \overline{\texttt{T}_\texttt{A}}\}$	Class list
TL	::=	`trait T is` $\{\overline{\texttt{M}}; \overline{\texttt{T}_\texttt{A}}\}$	Trait list
$\texttt{T}_\texttt{A}$	::=	$\texttt{T} \mid \texttt{T}_\texttt{A}$ `with` $\{\texttt{m} @ \texttt{m}\} \mid \texttt{T}_\texttt{A}$ `minus` $\{\texttt{m}\}$	Traits Alterations
K	::=	$\texttt{C}(\overline{\texttt{C}}\ \overline{\texttt{f}})\{\texttt{super}(\overline{\texttt{f}}); \texttt{this.}\overline{\texttt{f}} = \overline{\texttt{f}};\}$	Class Constructors
M	::=	$\texttt{C m}(\overline{\texttt{C}}\ \overline{\texttt{x}})\{\texttt{return e};\}$	Class methods
e	::=	x	FTJ expressions
	|	e.f	
	|	$\texttt{e.m}(\overline{\texttt{e}})$	
	|	$\texttt{new C}(\overline{\texttt{e}})$	
	|	(C)e	

Figure 1: Syntax of FTJ

à la Java, where programs are first typechecked and then executed forgetting all(most) type information. Compilation ensures the absence of *message-not-understood* run-time errors, enforcing greatly safety and speed up of the compiled code. However, it is well known that working in a typed setting has advantages and drawbacks:

- (*pro*) An advantage is that *provided* and *required* methods can be typechecked at compile time inside a class. The flattening property allows to typecheck traits used by a class *all-at-once w.r.t.* a single class. This ensures that the resulting class is complete, *i.e.* it does not contains unimplemented methods, nor method conflicts. Finally, typechecked code should be better compiled, and in principle, should run faster than an untyped one;
- (*contra*) The presence of types has the drawback of blocking legal (untyped) code, especially with a first-order type system *à la* Java; this limitation can be weakened by extending the present system with *mytype method specialization* (*i.e.* self types), or with *genericity à la* GJ [BOSW98].

3 Featherweight-Trait Java

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

3.1 Notational Conventions

- We adopt the same notational conventions and *sanity conditions* adopted in the seminal paper of FJ that we omit here.
- The metavariable $\mathtt{T}$ ranges over trait names, $\mathtt{T_A}$ ranges over trait alterations; TT ranges over trait declarations, and CT over class declarations.
- The $\overline{\mathtt{x}}$ set notation enforces every element be distinct; in the methods's case: $\overline{\mathtt{M}}$ enforces that there is only one method per method label.
- For a function/predicate $\mathcal{P}$, and a set $\overline{\mathtt{A}}$:
  - if $\mathcal{P}$ is a function returning sets, then $\mathcal{P}(\overline{\mathtt{A}}) = \bigcup_{\mathtt{A}\in\overline{\mathtt{A}}} \mathcal{P}(\mathtt{A})$;
  - if $\mathcal{P}$ is a predicate, then $\mathcal{P}(\overline{\mathtt{A}}) = \bigwedge_{\mathtt{A}\in\overline{\mathtt{A}}} \mathcal{P}(\mathtt{A})$.

3.2 Syntax

The syntax, and the (small-step) computational rules for FTJ are given in Figure 1. An FTJ program is a triple $(\mathsf{CT}, \mathsf{TT}, \mathtt{e})$ of a class table, a trait table, and an expression. A class in FTJ is composed by field declarations, a constructor, some methods that will help the typechecker to disambiguate conflict inside trait import, plus a list of used traits. A trait is composed by some list of methods and some other traits imported by the trait itself. imported trait can be altered by aliasing or removing methods. All conflicts will be discovered during the type checking phase, and solved using class checking rule. As in the untyped model, the diamond problem is not considered as a conflict. Expression are the usual ones of FJ.

3.3 Trait-Inheritance in FTJ

When presenting a small foundational object-oriented calculus, a special care has to be put in designing the essential features (syntax, method lookup, types, classes, traits). Although FTJ is a direct extension of FJ it has his own peculiarities, essentially because it features trait multiple inheritance, and because traits can be altered. With respect to this model, we list the novel features making FTJ an extension of FJ:

- the `super` is added in order to perform a method-lookup in the superclass;
- the *flattened property* is preserved (a non-overridden method in a trait has the same semantics as the same method implemented in the class);
- a class may import many traits: the composition order of traits and methods in traits is *irrelevant*;

- a trait can be *altered* either by dropping a method `m`, or by *aliasing* a method `m` with another method `n`;
- a *modified method lookup* is implemented to deal with trait and trait alterations;
- the *class methods* are useful and meaningful when a conflict between two or more traits arises; class methods takes precedence over trait methods which in turn take precedence over superclass methods. This is enforced by the type system;
- the *diamond problem* is solved in accordance with the design choices of the untyped model, keeping in mind that we are now in a "statically typed world", and as such, potential conflict detectable before run-time must be resolved at compile-time;
- FTJ, as FJ is, is a *functional* calculus, *i.e.* there is no notion of state and no assignment; as example [IPW01], when assigning a different value to a variable, we completely build another object from scratch with the new modified value in place of the old one. The possibility to enhance FTJ with imperative features is not problematic and it will be subject of further study;
- using traits is not enforced by the type system: in fact one cannot use traits at all, so resulting FTJ as a proper extension of FJ;
- we can typecheck traits only inside a class, *i.e.* inside a complete unit of behavior (and state); the possibility of typechecking traits as incomplete classes is worth of study (see conclusions).

4 Operational Semantics

The reduction relation is of the form $\mathtt{e} \longrightarrow \mathtt{e}'$, read "expression `e` reduces to expression $\mathtt{e}'$ in one step". The reduction rules are given in Appendix A. The sole difference with FJ is the dedicated use of the variable `super`. For this purpose, the function *mbody* return an extra result, namely the class where the method was declared in. As usual, calling a method `m` on `super` is just like calling `m` on an object of the direct superclass of the one where the current method is defined, with the corresponding types. In FJ, `super` was not added since it was syntactic sugar. However, adding traits prevents from handling `super` with usual syntax: if `super` appears in a trait, it acts exactly as if it appeared in the class the trait is used in, which makes `super` have a dynamic behavior. As in FJ, the variable `this` denotes the receiver itself (in the substitution). This is modeled by the following rule:

$$\frac{\mathit{mbody}(\mathtt{m},\mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e}_0, \mathtt{D}) \quad \mathtt{D} <:_{\mathsf{one}} \mathtt{E}}{(\mathtt{new\ C}(\overline{\mathtt{e}})).\mathtt{m}(\overline{\mathtt{d}}) \longrightarrow [\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new\ C}(\overline{\mathtt{e}})/\mathtt{this}, \mathtt{new\ E}(\overline{\mathtt{e}})/\mathtt{super}]\mathtt{e}_0} \text{ (Run·Call)}$$

The function *mbody* is a slight extension of the corresponding in FJ. It gives an extra result (as already discussed), and have an extra rule that uses the trait lookup function.

As usual, those reduction rules can be applied in any point of the computation, so the classical congruence rules apply, which we omit here. Also the usual rules of subclassing are omitted here (see Appendix A).

4.1 Dynamic Lookup.

The lookup algorithm (see Appendix A) takes into account the two simple rules of precedence of class methods over trait methods that, in turn, take precedence over superclass methods. Extra complications *w.r.t.* the lookup in FJ arises because of trait multiple inheritance and because traits can be altered.

Field lookup. This is performed as in FJ (see Appendix A).

Method lookup. This is performed by rules (MBody·Self) (first search in the current class) and (MBody·Trait) (then search in all imported traits) and (MBody·Super) (finally search in the direct superclass). The trait lookup routine just look for a method $\mathtt{m}$ only when $\mathtt{m}$ *is not* in the class methods; this forces the unicity of the search, otherwise a conflict would arise (since a class method could had overridden method $\mathtt{m}$). In particular the rule (MBody·Trait) looks as follows:

$$\frac{\begin{array}{l}\mathtt{CT(C)} = \mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}}; \mathtt{K}\ \overline{\mathtt{M}}\ \overline{\mathtt{T_A}}\} \\ \mathtt{m} \notin \mathit{meth}(\overline{\mathtt{M}}) \qquad \mathit{tlook}(\mathtt{m}, \overline{\mathtt{T_A}}) = \mathtt{B\ m}(\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\mathtt{return\ e;}\}\end{array}}{\mathit{mbody}(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e}, \mathtt{C})}\ \text{(MBody·Trait)}$$

Trait lookup. This is performed by the function *tlook* that intuitively searches a method into a trait alterations sequence. The two simple inference rules are as follows:

$$\frac{\exists \mathtt{T_A} \in \overline{\mathtt{T}}_\mathtt{A}.\ \mathit{altlook}(\mathtt{m}, \mathtt{T_A}) \neq \mathit{fail}}{\mathit{tlook}(\mathtt{m}, \overline{\mathtt{T}}_\mathtt{A}) = \mathit{altlook}(\mathtt{m}, \mathtt{T_A})}\ \text{(Trait·First)} \qquad \frac{\forall \mathtt{T_A} \in \overline{\mathtt{T}}_\mathtt{A}.\ \mathit{altlook}(\mathtt{m}, \mathtt{T_A}) = \mathit{fail}}{\mathit{tlook}(\mathtt{m}, \overline{\mathtt{T}}_\mathtt{A}) = \mathit{fail}}\ \text{(Trait·Last)}$$

The auxiliary function *altlook* takes into account altered traits (*i.e.* traits with dropped methods or with aliased methods). Finding a method which has been dropped or aliases is one of the key part of the lookup algorithm.

Trait alteration lookup. The trait alteration lookup are completely shown in the Appendix A: the "sexiest" trait alteration rules are the following ones:

$$\frac{\theta \equiv [\mathtt{this.m}/\mathtt{this.n}, \mathtt{super.m}/\mathtt{super.n}]}{\mathit{altlook}(\mathtt{m}, \mathtt{T_A\ with}\ \{\mathtt{n} \mathbin{@} \mathtt{m}\}) = \theta \mathit{altlook}(\mathtt{n}, \mathtt{T_A})}\ \text{(Alt·AliasedTo)}$$

$$\frac{\theta \equiv [\mathtt{this.q}/\mathtt{this.p}, \mathtt{super.q}/\mathtt{super.p}] \quad \mathtt{p} \neq \mathtt{m} \neq \mathtt{q}}{\mathit{altlook}(\mathtt{m}, \mathtt{T_A\ with}\ \{\mathtt{p} \mathbin{@} \mathtt{q}\}) = \theta \mathit{altlook}(\mathtt{m}, \mathtt{T_A})}\ \text{(Alt·NAliased)}$$

$$\frac{\mathtt{m} \neq \mathtt{n}}{\mathit{altlook}(\mathtt{m}, \mathtt{T_A\ with}\ \{\mathtt{m} \mathbin{@} \mathtt{n}\}) = \mathit{fail}}\ \text{(Alt·Aliased)}$$

First note that in the rules (Alt·AliasedTo) and (Alt·NAliased), θ is not a proper substitution, however it is as easy to compute as a real one. The purpose of that "substitution" is to catch every recursive and internal method call to `self`, and `super`. A trait is expected to be "a behaviour", One would not expect it to behave differently if a method is aliased, though it would obviously after a method gets excluded or overridden.

- In the (Alt·AliasedTo) rule, the run-time condition $\mathtt{m} \neq \mathtt{n}$ is not required since enforced by the type system (rules (Alias·Ok 1) and (Alias·Ok 2));
- In the (Alt·NAliased) rule, the run-time condition $\mathtt{p} \neq \mathtt{m} \neq \mathtt{q}$ guarantees that it is another method which is aliased;
- In the (Alt·AliasedTo) rule, the run-time condition $\mathtt{m} \neq \mathtt{n}$ is not necessary, since it is enforce by type system as precised above however, it has been left to emphasize the case disjunction.

On inheritance by diamonds and other dæmonic stuffs. Let *meth* be a simple function, that intuitively applied to a trait alteration, gives the set of each method label that is available in the alteration.

Definition 1 (Method Intersection and Diamond Detection)

$$\cap\overline{\mathtt{T}}_{\mathtt{A}} \triangleq \{\mathtt{m} \mid \exists \mathtt{T}_{\mathtt{A1}}, \mathtt{T}_{\mathtt{A2}} \in \overline{\mathtt{T}}_{\mathtt{A}}.\ \mathtt{m} \in \mathit{meth}(\mathtt{T}_{\mathtt{A1}}) \cap \mathit{meth}(\mathtt{T}_{\mathtt{A2}})\}$$

$$\diamond\overline{\mathtt{T}}_{\mathtt{A}} \triangleq \{\mathtt{m} \mid \exists \mathtt{T}_{\mathtt{A1}}, \mathtt{n}.\ \forall \mathtt{T}_{\mathtt{A2}} \in \overline{\mathtt{T}}_{\mathtt{A}}.\ \mathtt{m} \in \mathit{meth}(\mathtt{T}_{\mathtt{A2}}) \implies \mathtt{m}\ \mathsf{in}\ \mathtt{T}_{\mathtt{A2}} \trianglelefteq \mathtt{n}\ \mathsf{in}\ \mathtt{T}_{\mathtt{A1}}\}$$

- *The set $\cap\overline{\mathtt{T}}_{\mathtt{A}}$ denotes methods defined in more than one trait; this set is used to avoid conflicts when importing traits;*
- *The set $\diamond\overline{\mathtt{T}}_{\mathtt{A}}$ denotes methods that potentially determine a diamonds when playing with multiple inheritance; those methods are expected to be "non conflictual" hence accepted by the type system.*

In a nutshell: if there is no conflict in every $\mathtt{T}_{\mathtt{A}} \in \overline{\mathtt{T}}_{\mathtt{A}}$, $\cap\overline{\mathtt{T}}_{\mathtt{A}}$ grabs every conflict in $\overline{\mathtt{T}}_{\mathtt{A}}$ and, $\diamond\overline{\mathtt{T}}_{\mathtt{A}}$ grabs every diamond. It is used as follows:

$$\mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}}; \mathtt{K}\ \overline{\mathtt{M}}\ \overline{\mathtt{T}}_{\mathtt{A}}\}$$

only the traits satisfying the constraint

$$\cap\overline{\mathtt{T}}_{\mathtt{A}} \setminus \diamond\overline{\mathtt{T}}_{\mathtt{A}} \subseteq \mathit{meth}(\overline{\mathtt{M}})$$

have the "dignity" to be defined in the class `C` itself. The $\subseteq$ relation instead of a more restrictive $=$ relation is given in order to make FTJ a proper extension of FJ.

Subtraiting. To compute $\diamond\overline{\mathtt{T}}_{\mathtt{A}}$ we need a rather simple "subtraiting relation" (presented in Appendix A) whose most interesting rules are as follows (the other are routine)

$$\frac{\begin{array}{l}\mathsf{TT}(\mathtt{T}) = \mathtt{trait\ T\ is}\ \{\overline{\mathtt{M}};\overline{\mathtt{T}}_{\mathtt{A}}\}\\ \mathtt{T}_{\mathtt{A}} \in \overline{\mathtt{T}}_{\mathtt{A}} \qquad \mathtt{m} \in \mathit{meth}(\mathtt{T}_{\mathtt{A}}) \setminus \mathit{meth}(\overline{\mathtt{M}})\end{array}}{\mathtt{m\ in\ T} \trianglelefteqslant \mathtt{m\ in\ T}_{\mathtt{A}}}\ (\text{Path}\cdot\text{Inherit})$$

$$\frac{\mathtt{m\ in\ T}_{\mathtt{A1}} \trianglelefteqslant \mathtt{m\ in\ T}_{\mathtt{A2}} \qquad \mathtt{m} \neq \mathtt{p}}{\mathtt{m\ in\ T}_{\mathtt{A1}}\ \mathtt{minus}\ \{\mathtt{p}\} \trianglelefteqslant \mathtt{m\ in\ T}_{\mathtt{A2}}}\ (\text{Path}\cdot\text{Exclude})$$

The intuition behind a judgment

$$\mathtt{m\ in\ T}_{\mathtt{A1}} \trianglelefteqslant \mathtt{m\ in\ T}_{\mathtt{A2}}$$

is that m is a method provided by both $\mathtt{T}_{\mathtt{A1}}$ and $\mathtt{T}_{\mathtt{A2}}$ and the m provided by $\mathtt{T}_{\mathtt{A1}}$ is inherited through any number of trait definition or alteration steps from m provided by $\mathtt{T}_{\mathtt{A2}}$.

5 The Type System

This section introduces the most exciting rules of the type system of FTJ. The full set of rules is reported in Appendix B. The type system has two step: first, expression typing as in any statically typed language, where judgment of the following shape are proved:

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

and class typechecking (as in FJ): since everything in classes is typed explicitly the system only has to check if the declaration is correct. In addition to FJ our typechecking system checks also conflict resolutions. It proves judgment of the shape:

$$\mathtt{M\ OK\ IN\ C} \quad \text{and} \quad \mathtt{T\ OK\ IN\ C\ except}\ \overline{\mathtt{m}} \quad \text{and} \quad \mathtt{C\ OK}$$

Basic expression typing and Valid type lookup. These rules (presented in Appendix B) deserve no surprises *w.r.t.* the corresponding ones of FJ.

Method Checking. The only rule has the following shape:

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

This rule checks if a method is well typed as a method of class C: it first checks if the type of the body is correct (*i.e.* it has no unbound variable or method call, and the type correspond to the declared type of the method), then, thanks to the the *override* predicate, it ensures there is no overloading of an inherited method (the result type of the method may be covariant *w.r.t.* the result type assigned to $\mathtt{e}_{\mathtt{m}}$.)

Trait checking. These rules are quite interesting and tricky: we display here the three "sexiest" but we comment all four rules. This rules derive judgment of the shape $\mathtt{T_A\ OK\ IN\ C\ except}\ \overline{\mathtt{m}}$ which mean that $\mathtt{T_A}$ is well-typed, however, to define it in class C every method of $\overline{\mathtt{m}}$ must be overridden.

$$\frac{\mathsf{TT}(\mathtt{T}) = \mathtt{trait\ T\ is}\ \{\overline{\mathtt{M}}; \overline{\mathtt{T}}_\mathtt{A}\} \quad \overline{\mathtt{N}} \triangleq \{\mathtt{M} \in \overline{\mathtt{M}} \mid \neg \mathtt{M\ OK\ IN\ C}\} \\ \overline{\mathtt{T}}_\mathtt{A}\ \mathtt{OK\ IN\ C\ except}\ \overline{\mathtt{m}} \qquad \cap\overline{\mathtt{T}}_\mathtt{A} \setminus \diamond\overline{\mathtt{T}}_\mathtt{A} \subseteq \mathit{meth}(\overline{\mathtt{M}})}{\mathtt{T\ OK\ IN\ C\ except}\ \mathit{meth}(\overline{\mathtt{N}}) \cup (\overline{\mathtt{m}} \setminus \mathit{meth}(\overline{\mathtt{M}}))}\ (\text{T}\cdot\text{Ok})$$

$$\frac{\mathit{altlook}(\mathtt{n}, \mathtt{T_A\ with}\ \{\mathtt{m} @ \mathtt{n}\}) = \mathtt{M} \quad \mathtt{M\ OK\ IN\ C} \\ \mathtt{T_A\ OK\ IN\ C\ except}\ \overline{\mathtt{p}} \qquad \mathtt{n} \notin \mathit{meth}(\mathtt{T_A})}{\mathtt{T_A\ with}\ \{\mathtt{m} @ \mathtt{n}\}\ \mathtt{OK\ IN\ C\ except}\ \overline{\mathtt{p}} \setminus \{\mathtt{m}\}}\ (\text{Alias}\cdot\text{Ok 1})$$

$$\frac{\mathit{altlook}(\mathtt{n}, \mathtt{T_A\ with}\ \{\mathtt{m} @ \mathtt{n}\}) = \mathtt{M} \quad \neg\mathtt{M\ OK\ IN\ C} \\ \mathtt{T_A\ OK\ IN\ C\ except}\ \overline{\mathtt{p}} \qquad \mathtt{n} \notin \mathit{meth}(\mathtt{T_A})}{\mathtt{T_A\ with}\ \{\mathtt{m} @ \mathtt{n}\}\ \mathtt{OK\ IN\ C\ except}\ (\overline{\mathtt{p}} \setminus \{\mathtt{m}\}) \cup \{\mathtt{n}\}}\ (\text{Alias}\cdot\text{Ok 2})$$

- (T·Ok) ensures that every $\mathtt{T_A} \in \overline{\mathtt{T}}_\mathtt{A}$ is well-typed.Intuitively,
  - we fetch the trait T in the trait-table TT;
  - we fetch all the methods $\overline{\mathtt{N}}$ defined in the trait T that do not type check in C;
  - we typecheck the set of altered trait $\overline{\mathtt{T}}_\mathtt{A}$ in C producing a set of illegal method (the except $\overline{\mathtt{m}}$ part) which are the methods of $\overline{\mathtt{T}}_\mathtt{A}$ which do not typecheck in C;
  - we check the key condition $\cap\overline{\mathtt{T}}_\mathtt{A} \setminus \diamond\overline{\mathtt{T}}_\mathtt{A} \subseteq \mathit{meth}(\overline{\mathtt{M}})$ that ensures that every conflict is resolved, *i.e.* every new-born conflict $(\cap\overline{\mathtt{T}}_\mathtt{A})$(note that conflicts are resolved recursively) which is not a diamond ones $(\diamond\overline{\mathtt{T}}_\mathtt{A})$ is being overridden, so that precedence rules provide a conflict resolution;
  - we build a new set of illegal methods in $\overline{\mathtt{T}}$ *w.r.t.* C, *i.e.* $\mathit{meth}(\overline{\mathtt{N}}) \cup (\overline{\mathtt{m}} \setminus \mathit{meth}(\overline{\mathtt{M}}))$, *i.e.* the illegal method of $\mathtt{T_A}$ $(\mathit{meth}(\overline{\mathtt{N}}))$ plus the non-overridden methods of $\overline{\mathtt{T}}_\mathtt{A}$ $(\overline{\mathtt{m}} \setminus \mathit{meth}(\overline{\mathtt{M}}))$, since overridden ones do not matter anymore thanks to the previous rules;
- (Alias·Ok 1) ensures that $\mathtt{T_A}$ is well-typed, then that M (new aliased method) is well-typed in C and just remove m (former method name) from the illegal methods;
- (Alias·Ok 2) behaves as for (Alias·Ok 1), except that M is not well-typed, and n is added to the illegal methods;
- (Exclude·Ok) is intuitive: if $\mathtt{T_A}$ is well-typed, then excluding m just remove m from the illegal-methods.

Class checking. The only rule performing class checking is the *summa* of all previous inference rules. The rule has the following shape:

$$\frac{\begin{array}{l}\texttt{K} = \texttt{C}(\overline{\texttt{D}}\ \overline{\texttt{g}}, \overline{\texttt{C}}\ \overline{\texttt{f}})\{\texttt{super}(\overline{\texttt{g}}); \texttt{this.}\overline{\texttt{f}} = \overline{\texttt{f}}; \} \\ \mathit{fields}(\texttt{D}) = \overline{\texttt{D}}\ \overline{\texttt{g}} \quad \cap\overline{\texttt{T}}_{\texttt{A}} \setminus \diamond\overline{\texttt{T}}_{\texttt{A}} \subseteq \mathit{meth}(\overline{\texttt{M}}) \\ \overline{\texttt{M}}\ \texttt{OK IN C} \quad \overline{\texttt{T}}_{\texttt{A}}\ \texttt{OK IN C except}\ \overline{\texttt{m}} \quad \overline{\texttt{m}} \subseteq \mathit{meth}(\overline{\texttt{M}})\end{array}}{\texttt{class C extends D}\ \{\overline{\texttt{C}}\ \overline{\texttt{f}}; \texttt{K}\ \overline{\texttt{M}}\ \overline{\texttt{T}}_{\texttt{A}}\}\ \texttt{OK}}\ (\text{C}\cdot\text{Ok})$$

Intuitively, this checks that all the components of the class are well typed. and that all conflicts are resolved. This typechecking rule assures, that FTJ is a proper extension of FJ, thanks to both $\subseteq$ symbols which assure compatibility whenever $\overline{\texttt{T}}_{\texttt{A}}$ is empty. More precisely:

- we fetch the constructor K, and the fields $\overline{\texttt{g}}$;
- we check the key condition $\cap\overline{\texttt{T}}_{\texttt{A}} \setminus \diamond\overline{\texttt{T}}_{\texttt{A}} \subseteq \mathit{meth}(\overline{\texttt{M}})$ that ensures that every conflict is resolved (see the explanation about (T·Ok) above);
- we typecheck all method $\overline{\texttt{M}}$ defined in C;
- we typecheck the set of altered trait $\overline{\texttt{T}}_{\texttt{A}}$ in C producing a set of illegal method (the except $\overline{\texttt{m}}$ part) that are the methods of $\texttt{T}_{\texttt{A}}$ which do not typecheck in C;
- we check that those illegal methods are redefined in methods $\overline{\texttt{M}}$ defined in the class C.

The $\overline{\texttt{m}} \subseteq \mathit{meth}(\overline{\texttt{M}})$ deserve a little discussion. Note that there is two main possibilities for m to be illegal in C:

1. The method has not the expected type for the *override* predicate. Which means that you must override it with a method m with the right type in C. However, if there is a method p that makes some use of m via this then, p would immediately not typecheck in C since m has not the correct type. If such case arise, you could need to override many of the trait methods; which is not the expected use of traits of course. In such a case, you would better use aliasing;
2. The type of m's body is not correct (failure of $\overline{\texttt{x}}{:}\overline{\texttt{C}}$, this:C, super:D $\vdash \texttt{e}_{\texttt{m}} \in \texttt{F}$ or $\texttt{F} <: \texttt{E}$). Then, you need to override the method m because it doesn't behave as expected (most likely it would require methods you do not intend to provide). However, it wouldn't cause any unsound use of this.m() in the other methods of the trait, what make this case harmless, for typing issue.

5.1 Method type lookup and valid method overriding

These rules deserve no surprise (see Appendix B).

6 Properties

Once defined the operational semantics and the type system, the classical theoretical "sport" is to prove that *i*) the static semantics matches with the dynamic one, *i.e.* types are preserved during computation (modulo subtyping), and that *ii*) the interpreter cannot get stuck if programs only includes upcasts, and finally that *iii*) the type system prevents compiled programs from the unfortunate run-time *message-not-understood* error. The nice "catch of the day" in designing FJ was that the typed trait model fits perfectly with a first-order type system *à la* Java! Of course, a lot of care must devoted in dealing with multiple inheritance and trait alterations.

Full proofs of the theorems are provided in Appendix C.

Definition 2 (Determinism) *A function $\mathcal{F}$ defined by inference rules is said to have property $\mathcal{D}$, if $\forall v_1, v_2.\ \mathcal{F}(\overline{x})=v_1$ is derivable and $\mathcal{F}(\overline{x})=v_2$ is derivable, then $v_1=v_2$.*

The Conflict Resolution Theorem proves that the conflicts are resolved deterministically for well-typed programs. It is the main theorem of FTJ since it proves that traits does not affect all nice properties of FJ.

Theorem 1 (Conflict Resolution)
$\forall \mathtt{C}_i \in \mathtt{CL}$, *such that* $\mathtt{C}_i$ OK, *then both mbody and mytype have property* $\mathcal{D}$. □

Subject reduction proves that if an expression is typable and it reduces to a second expression, then the latter expression is typable too, with a subtype of the type of the former expression.

Theorem 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 to stuck the interpreter is reach a state where a downcast is impossible.

Theorem 3 (Progress)
Suppose e *is a well-typed expression.*

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

In total similarity with FJ, we define the notion of *safe* expression e in Γ if the type derivation of the underlying $(\mathsf{CT}, \mathsf{TT})$ and $\Gamma \vdash \mathtt{e} \in \mathtt{C}$ contains no downcast or stupid cast (rules (Type·DCast), and (Type·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 typecast in a safe expression will never fail.

Theorem 4 (Reduction preserves safety)
If e *is safe in* Γ, *and* $\mathtt{e} \to \mathtt{e}'$, *then* e′ *is safe in* Γ. □

Theorem 5 (Progress of safe programs)
Suppose $\mathtt{e}$ *is safe in* Γ. *If* $\mathtt{e}$ *has* $(\mathtt{C})\mathtt{new}\ \mathtt{D}(\overline{\mathtt{e}})$ *as a subexpression, then* $\mathtt{D} <: \mathtt{C}$. □

7 Related work and Conclusions

Related. The more intriguing works about handling statically typed traits in a multiple inheritance setting are the ones of Fisher and Reppy [FR04] (in the contexts of the programming language Moby (of the ML family), and the Scala language [?] by Odersky *et al.*. In a nutshell:

- the main difference with [FR04] and our work is the absence of imperative features. The calculus introduce a statically typed trait-based inheritance class-based calculus. This calculus is very interesting since it is imperative, it is patented object-based, it have a sophisticated type system based on polymorphic types.
- the Scala language uses a version of non-linear mixin-based inheritance which even subsumes trait inheritance. Intuitively a class can be used as a mixin in the definition of another class, if this other class extends a subclass of the superclass of the mixin; traits here represents essentially *interfaces* in Java. We expect a great flexibility from this model and we conjecture (without proof) that Scala inheritance could subsume trait multiple inheritance.

Conclusions. In this paper, we have presented a formal development of the theory of FTJ, a statically (non imperative) typed class-based language featuring classes, objects, and traits. To our knowledge, no similar study of statically typed class-based languages can be found in the literature.

Among the possible future direction, the next questions on our agenda are:

- It is the author opinion that trait-based inheritance shares and mix some of the philosophy of Aspect-Oriented Programming (*AOP*) *à la* Kitzcales *et al.* [KLM+97], and Variation-Oriented Programming (*VOP*) *à la* Mezini [Mez02], and Object-based Programming (*OBP*) *à la* Self [US87]. We argue that (yet) another "dialect" of such paradigms, which we call Trait-Oriented Programming (*TOP*) should be worth of study.
- the presented type system allows to typecheck traits only within classes (*i.e.* under the assumption that we compile one class at a time); in fact, by typing a class all requirements of all traits must be solved inside the class itself (otherwise the created instances would generate a *message-not-understood* upon some non implemented message send); it would be very interesting to enhance the type system of FTJ with *separate trait compilation* by typing also the required methods of the trait itself. This is reminiscent of a previous author's work on *incomplete objects* [BBDL97] in an object-based setting. The resulting calculus would be more adapt in a setting where trait compilation is a desired feature. In a Java setting this would result in allowing *dynamic trait loading* at run-time.

- Adding bounded polymorphic-types or even generic-types will greatly improve the usefulness of statically typed traits. Those (possible) extensions are treated in a drafty companion paper [Liq04].
- study the mechanism of trait in a basic calculus with *imperative features*.
- analyze the similarities and the differences underneath Scala, Moby, and FTJ typed multiple inheritance models.
- study the impact of trait-inheritance in for the language C# ; although this language is quite similar to Java is has its peculiarities, that should be carefully be interleaved and compatible with the typed trait inheritance model.

References

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[BBDL97] V. Bono, M. Bugliesi, M. Dezani-Ciancaglini, and L. Liquori. Subtyping Constraint for Incomplete Objects. In *Proc. of TAPSOFT/CAAP*, volume 1214 of *LNCS*, pages 465–477, 1997.

[BDG+88] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kicsales, and D. A. Moon. Common lisp object system specification x3j13 document 88-002r. *ACM SIGPLAN Notices*, 23:1–143, 1988.

[BOSW98] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the java programming language. In *Proc. OOPSLA*, 1998.

[Car95] Luca Cardelli. Obliq: A Language with Distributed Scope. *Computing Systems*, 8(1):27–59, 1995.

[FR04] K. Fisher and J. Reppy. Statically Typed Traits. Presented at FOOL 10, 2004.

[IPW01] A. Igarashi, B.C. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proc. of ECOOP*, number 1241 in LNCS, pages 220–242, 1997.

[Liq04] L. Liquori. Adding Genericity and bounded Polymorphism to FTJ. Manuscript, 2004.

[Mez02] M. Mezini. Towards Variational Object-Oriented Programming: The Rondo Model. Technical Report TUD-ST-2002-02, Software Technology Group, Darmstadt University of Technology, Alexanderstrasse 10, 64289 Darmstadt, Germany, 2002.

[SDNB03] N. Schärli, S. Ducasse, O. Nierstrasz, and A.P. Black. Traits: Composable Units of Behaviour. In *Proc. of ECOOP*, number 2743 in LNCS, pages 248–274, 2003.

[Sny87] A. Snyder. Inheritance and the Development of Encapsulated Software Systems. In *Research Directions in Object-Oriented Programming*, pages 165–188. MIT Press, 1987.

[US87] D. Ungar and B. Smith, R. Self: The Power of Simplicity. In *Proc. of OOPSLA*, pages 227–241. The ACM Press, 1987.

A Small Step Operational Semantics

A.1 Syntax of FTJ

Syntax

CL	::=	$\texttt{class C extends C } \{\overline{\texttt{C}}\ \overline{\texttt{f}}; \texttt{K}\ \overline{\texttt{M}}\ \overline{\texttt{T}_\texttt{A}}\}$	Class list
TL	::=	$\texttt{trait T is } \{\overline{\texttt{M}}; \overline{\texttt{T}_\texttt{A}}\}$	Trait list
$\texttt{T}_\texttt{A}$	::=	$\texttt{T} \mid \texttt{T}_\texttt{A}\ \texttt{with } \{\texttt{m@m}\} \mid \texttt{T}_\texttt{A}\ \texttt{minus } \{\texttt{m}\}$	Traits Alterations
K	::=	$\texttt{C}(\overline{\texttt{C}}\ \overline{\texttt{f}})\{\texttt{super}(\overline{\texttt{f}}); \texttt{this.}\overline{\texttt{f}} = \overline{\texttt{f}};\}$	Class Constructors
M	::=	$\texttt{C m}(\overline{\texttt{C}}\ \overline{\texttt{x}})\{\texttt{return e};\}$	Class methods
e	::=	x	FTJ expressions
	|	e.f	
	|	$\texttt{e.m}(\overline{\texttt{e}})$	
	|	$\texttt{new C}(\overline{\texttt{e}})$	
	|	(C)e	

A.2 Operational semantics of FTJ

Subtyping

$$\frac{}{\texttt{C} <: \texttt{C}}\ \text{(Sub·Refl)} \qquad \frac{\texttt{C} <: \texttt{D} \quad \texttt{D} <: \texttt{E}}{\texttt{C} <: \texttt{E}}\ \text{(Sub·Trans)} \qquad \frac{\texttt{C} <:_{\texttt{one}} \texttt{D}}{\texttt{C} <: \texttt{D}}\ \text{(Sub·Step)}$$

$$\frac{\texttt{CT(C)} = \texttt{class C extends D } \{\ldots\}}{\texttt{C} <:_{\texttt{one}} \texttt{D}}\ \text{(Sub·One)}$$

Computation

$$\frac{\mathit{fields}(\texttt{C}) = \overline{\texttt{C}}\ \overline{\texttt{f}}}{(\texttt{new C}(\overline{\texttt{e}}))\texttt{.f}_\texttt{i} \longrightarrow \texttt{e}_\texttt{i}}\ \text{(Run·Field)} \qquad \frac{\texttt{C} <: \texttt{D}}{(\texttt{D})(\texttt{new C}(\overline{\texttt{e}})) \longrightarrow \texttt{new C}(\overline{\texttt{e}})}\ \text{(Run·Cast)}$$

$$\frac{\mathit{mbody}(\texttt{m}, \texttt{C}) = (\overline{\texttt{x}}, \texttt{e}_0, \texttt{D}) \quad \texttt{D} <:_{\texttt{one}} \texttt{E}}{(\texttt{new C}(\overline{\texttt{e}}))\texttt{.m}(\overline{\texttt{d}}) \longrightarrow [\overline{\texttt{d}}/\overline{\texttt{x}}, \texttt{new C}(\overline{\texttt{e}})/\texttt{this}, \texttt{new E}(\overline{\texttt{e}})/\texttt{super}]\texttt{e}_0}\ \text{(Run·Call)}$$

A.3 Fields/Body/Trait Lookup in FTJ

Field lookup

$$\frac{}{\mathit{fields}(\texttt{Object}) = \bullet}\ (\text{Field·Top}) \qquad \frac{\mathsf{CT}(\texttt{C}) = \texttt{class C extends D } \{\overline{\texttt{C}}\ \overline{\texttt{f}}; \texttt{K}\ \overline{\texttt{M}}\ \overline{\texttt{T}_\texttt{A}}\} \quad \mathit{fields}(\texttt{D}) = \overline{\texttt{D}}\ \overline{\texttt{g}}}{\mathit{fields}(\texttt{C}) = \overline{\texttt{D}}\ \overline{\texttt{g}}, \overline{\texttt{C}}\ \overline{\texttt{f}}}\ (\text{Field·Class})$$

Method body lookup

$$\frac{\mathsf{CT}(\texttt{C}) = \texttt{class C extends D } \{\overline{\texttt{C}}\ \overline{\texttt{f}}; \texttt{K}\ \overline{\texttt{M}}\ \overline{\texttt{T}_\texttt{A}}\} \quad \texttt{B m } (\overline{\texttt{B}}\ \overline{\texttt{x}})\{\texttt{return e;}\} \in \overline{\texttt{M}}}{\mathit{mbody}(\texttt{m}, \texttt{C}) = (\overline{\texttt{x}}, \texttt{e}, \texttt{C})}\ (\text{MBody·Self})$$

$$\frac{\mathsf{CT}(\texttt{C}) = \texttt{class C extends D } \{\overline{\texttt{C}}\ \overline{\texttt{f}}; \texttt{K}\ \overline{\texttt{M}}\ \overline{\texttt{T}_\texttt{A}}\} \quad \texttt{m} \notin \mathit{meth}(\overline{\texttt{M}}) \quad \mathit{tlook}(\texttt{m}, \overline{\texttt{T}_\texttt{A}}) = \texttt{B m}(\overline{\texttt{B}}\ \overline{\texttt{x}})\{\texttt{return e;}\}}{\mathit{mbody}(\texttt{m}, \texttt{C}) = (\overline{\texttt{x}}, \texttt{e}, \texttt{C})}\ (\text{MBody·Trait})$$

$$\frac{\mathsf{CT}(\texttt{C}) = \texttt{class C extends D } \{\overline{\texttt{C}}\ \overline{\texttt{f}}; \texttt{K}\ \overline{\texttt{M}}\ \overline{\texttt{T}_\texttt{A}}\} \quad \texttt{m} \notin \mathit{meth}(\overline{\texttt{M}}) \quad \mathit{tlook}(\texttt{m}, \overline{\texttt{T}_\texttt{A}}) = \mathit{fail}}{\mathit{mbody}(\texttt{m}, \texttt{C}) = \mathit{mbody}(\texttt{m}, \texttt{D})}\ (\text{MBody·Super})$$

Trait lookup

$$\frac{\exists \texttt{T}_\texttt{A} \in \overline{\texttt{T}_\texttt{A}}.\ \mathit{altlook}(\texttt{m}, \texttt{T}_\texttt{A}) \neq \mathit{fail}}{\mathit{tlook}(\texttt{m}, \overline{\texttt{T}_\texttt{A}}) = \mathit{altlook}(\texttt{m}, \texttt{T}_\texttt{A})}\ (\text{Trait·First}) \qquad \frac{\forall \texttt{T}_\texttt{A} \in \overline{\texttt{T}_\texttt{A}}.\ \mathit{altlook}(\texttt{m}, \texttt{T}_\texttt{A}) = \mathit{fail}}{\mathit{tlook}(\texttt{m}, \overline{\texttt{T}_\texttt{A}}) = \mathit{fail}}\ (\text{Trait·Last})$$

Trait alteration lookup

$$\frac{\mathsf{TT}(\texttt{T}) = \texttt{trait T is } \{\overline{\texttt{M}}; \overline{\texttt{T}_\texttt{A}}\} \quad \texttt{B m}(\overline{\texttt{B}}\ \overline{\texttt{x}})\{\texttt{return e;}\} \in \overline{\texttt{M}}}{\mathit{altlook}(\texttt{m}, \texttt{T}) = \texttt{B m}(\overline{\texttt{B}}\ \overline{\texttt{x}})\{\texttt{return e;}\}}\ (\text{Alt·Top}) \qquad \frac{\mathsf{TT}(\texttt{T}) = \texttt{trait T is } \{\overline{\texttt{M}}; \overline{\texttt{T}_\texttt{A}}\} \quad \texttt{m} \notin \mathit{meth}(\overline{\texttt{M}})}{\mathit{altlook}(\texttt{m}, \texttt{T}) = \mathit{tlook}(\texttt{m}, \overline{\texttt{T}_\texttt{A}})}\ (\text{Alt·Rec})$$

$$\frac{\theta \equiv [\texttt{this.m/this.n}, \texttt{super.m/super.n}]}{\mathit{altlook}(\texttt{m}, \texttt{T}_\texttt{A}\ \texttt{with}\ \{\texttt{n} \mathbin{@} \texttt{m}\}) = \theta\mathit{altlook}(\texttt{n}, \texttt{T}_\texttt{A})}\ (\text{Alt·AliasedTo})$$

$$\frac{\theta \equiv [\texttt{this.q/this.p}, \texttt{super.q/super.p}] \quad \texttt{p} \neq \texttt{m} \neq \texttt{q}}{\mathit{altlook}(\texttt{m}, \texttt{T}_\texttt{A}\ \texttt{with}\ \{\texttt{p} \mathbin{@} \texttt{q}\}) = \theta\mathit{altlook}(\texttt{m}, \texttt{T}_\texttt{A})}\ (\text{Alt·NAliased})$$

$$\frac{\mathtt{m} \neq \mathtt{n}}{\mathit{altlook}(\mathtt{m}, \mathtt{T_A\ with\ \{m @ n\}}) = \mathit{fail}}\ \text{(Alt·Aliased)}$$

$$\frac{\mathtt{m} \neq \mathtt{n}}{\mathit{altlook}(\mathtt{m}, \mathtt{T_A\ minus\ \{n\}}) = \mathit{altlook}(\mathtt{m}, \mathtt{T_A})}\ \text{(Alt·NExclude)}$$

$$\frac{}{\mathit{altlook}(\mathtt{m}, \mathtt{T_A\ minus\ \{m\}}) = \mathit{fail}}\ \text{(Alt·Excluded)}$$

A.4 Method search in FTJ

Method label

$$\frac{}{\mathit{meth}(\mathtt{C\ m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\{\mathtt{return\ e;}\}) = \{\mathtt{m}\}}\ \text{(Meth·Labels)}$$

$$\frac{\mathsf{TT}(\mathtt{T}) = \mathtt{trait\ T\ is}\ \{\overline{\mathtt{M}}; \overline{\mathtt{T}}_\mathtt{A}\} \quad \overline{\mathtt{m}} = \mathit{meth}(\overline{\mathtt{M}})}{\mathit{meth}(\mathtt{T}) = \overline{\mathtt{m}} \cup \mathit{meth}(\overline{\mathtt{T}}_\mathtt{A})}\ \text{(Meth·Trait)}$$

$$\frac{}{\mathit{meth}(\mathtt{T_A\ with\ \{m @ n\}}) = (\mathit{meth}(\mathtt{T_A}) \setminus \{\mathtt{m}\}) \cup \{\mathtt{n}\}}\ \text{(Meth·Alias)}$$

$$\frac{}{\mathit{meth}(\mathtt{T_A\ minus\ \{m\}}) = \mathit{meth}(\mathtt{T_A}) \setminus \{\mathtt{m}\}}\ \text{(Meth·Exclude)}$$

Method labels in trait intersection and in diamond trait

$$\cap\overline{\mathtt{T}}_\mathtt{A} \triangleq \{\mathtt{m} \mid \exists \mathtt{T_{A1}}, \mathtt{T_{A2}} \in \overline{\mathtt{T}}_\mathtt{A}.\ \mathtt{m} \in \mathit{meth}(\mathtt{T_{A1}}) \cap \mathit{meth}(\mathtt{T_{A2}})\}$$

$$\diamond\overline{\mathtt{T}}_\mathtt{A} \triangleq \{\mathtt{m} \mid \exists \mathtt{T_{A1}}.\ \forall \mathtt{T_{A2}} \in \overline{\mathtt{T}}_\mathtt{A}.\ \mathtt{m} \in \mathit{meth}(\mathtt{T_{A2}}) \implies \mathtt{m\ in\ T_{A2}} \trianglelefteq \mathtt{m\ in\ T_{A1}}\}$$

Methods paths in traits alterations

$$\frac{\mathtt{m\ in\ T_{A1}} \trianglelefteq \mathtt{m\ in\ T_{A2}} \qquad \mathtt{m} \neq \mathtt{p}}{\mathtt{m\ in\ T_{A1}\ minus\ \{p\}} \trianglelefteq \mathtt{m\ in\ T_{A2}}}\ \text{(Path·Exclude)} \qquad \frac{\mathtt{m} \in \mathit{meth}(\mathtt{T_A})}{\mathtt{m\ in\ T_A} \trianglelefteq \mathtt{m\ in\ T_A}}\ \text{(Path·Refl)}$$

$$\frac{\begin{array}{l}\mathsf{TT}(\mathtt{T}) = \mathtt{trait\ T\ is}\ \{\overline{\mathtt{M}}; \overline{\mathtt{T}}_\mathtt{A}\} \\ \mathtt{T_A} \in \overline{\mathtt{T}}_\mathtt{A} \qquad \mathtt{m} \in \mathit{meth}(\mathtt{T_A}) \setminus \mathit{meth}(\overline{\mathtt{M}})\end{array}}{\mathtt{m\ in\ T} \trianglelefteq \mathtt{m\ in\ T_A}}\ \text{(Path·Inherit)} \qquad \frac{\begin{array}{l}\mathtt{m\ in\ T_{A1}} \trianglelefteq \mathtt{m\ in\ T_{A2}} \\ \mathtt{m\ in\ T_{A2}} \trianglelefteq \mathtt{m\ in\ T_{A3}}\end{array}}{\mathtt{m\ in\ T_{A1}} \trianglelefteq \mathtt{m\ in\ T_{A3}}}\ \text{(Path·Trans)}$$

B The Type System

B.1 Basic Typing in FTJ

Basic Expression typing

$$\frac{}{\Gamma \vdash \mathtt{x} \in \Gamma(\mathtt{x})}\ (\text{Type·Var})$$

$$\frac{\Gamma \vdash \mathtt{e}_0 \in \mathtt{C}_0 \quad \mathit{mytype}(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{D}} \to \mathtt{C} \quad \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{C}} \quad \overline{\mathtt{C}} <: \overline{\mathtt{D}}}{\Gamma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) \in \mathtt{C}}\ (\text{Type·Call})$$

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

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

$$\frac{\Gamma \vdash \mathtt{e}_0 \in \mathtt{D} \quad \mathtt{D} <: \mathtt{C}}{\Gamma \vdash (\mathtt{C})\mathtt{e}_0 \in \mathtt{C}}\ (\text{Type·UCast})$$

$$\frac{\Gamma \vdash \mathtt{e}_0 \in \mathtt{D} \quad \mathtt{C} <: \mathtt{D} \quad \mathtt{C} \neq \mathtt{D}}{\Gamma \vdash (\mathtt{C})\mathtt{e}_0 \in \mathtt{C}}\ (\text{Type·DCast})$$

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

Method type lookup

$$\frac{\begin{array}{l}\mathtt{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}}; \mathtt{K}\ \overline{\mathtt{M}}\ \overline{\mathtt{T}_\mathtt{A}}\} \\ \mathtt{B}\ \mathtt{m}(\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\mathtt{return}\ \mathtt{e};\} \in \overline{\mathtt{M}}\end{array}}{\mathit{mytype}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{B}} \to \mathtt{B}}\ (\text{MType·Self})$$

$$\frac{\begin{array}{l}\mathtt{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}}; \mathtt{K}\ \overline{\mathtt{M}}\ \overline{\mathtt{T}_\mathtt{A}}\} \\ \mathtt{m} \notin \mathit{meth}(\overline{\mathtt{M}}) \quad \mathit{tlook}(\mathtt{m}, \overline{\mathtt{T}_\mathtt{A}}) = \mathtt{B}\ \mathtt{m}(\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\mathtt{return}\ \mathtt{e};\}\end{array}}{\mathit{mytype}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{B}} \to \mathtt{B}}\ (\text{MType·Trait})$$

$$\frac{\begin{array}{l}\mathtt{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}}; \mathtt{K}\ \overline{\mathtt{M}}\ \overline{\mathtt{T}_\mathtt{A}}\} \\ \mathtt{m} \notin \mathit{meth}(\overline{\mathtt{M}}) \qquad \mathit{tlook}(\mathtt{m}, \overline{\mathtt{T}_\mathtt{A}}) = \mathit{fail}\end{array}}{\mathit{mytype}(\mathtt{m}, \mathtt{C}) = \mathit{mytype}(\mathtt{m}, \mathtt{D})}\ (\text{MType·Super})$$

B.2 Trait's Type system in FTJ

Valid method overriding

$$\frac{\mathit{mytype}(\mathtt{m},\mathtt{D}) = \overline{\mathtt{D}} \to \mathtt{D}_0 \ \mathit{implies}\ \overline{\mathtt{C}} = \overline{\mathtt{D}} \ \mathit{and}\ \mathtt{C}_0 = \mathtt{D}_0}{\mathit{override}(\mathtt{m},\mathtt{D},\overline{\mathtt{C}} \to \mathtt{C}_0)}\ (\text{M}\cdot\text{Over})$$

Method checking

$$\frac{\begin{array}{ll}\mathtt{CT}(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\ldots\} & \mathit{override}(\mathtt{m},\mathtt{D},\overline{\mathtt{C}} \to \mathtt{E}) \\ \overline{\mathtt{x}}{:}\overline{\mathtt{C}}, \mathtt{this}{:}\mathtt{C}, \mathtt{super}{:}\mathtt{D} \vdash \mathtt{e_m} \in \mathtt{F} & \mathtt{F} <: \mathtt{E}\end{array}}{\mathtt{E\ m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\{\mathtt{return\ e_m};\}\ \mathtt{OK\ IN\ C}}\ (\text{M}\cdot\text{Ok})$$

Trait checking

$$\frac{\begin{array}{ll}\mathtt{TT}(\mathtt{T}) = \mathtt{trait\ T\ is}\ \{\overline{\mathtt{M}};\overline{\mathtt{T}_\mathtt{A}}\} & \overline{\mathtt{N}} \triangleq \{\mathtt{M} \in \overline{\mathtt{M}} \mid \neg \mathtt{M\ OK\ IN\ C}\} \\ \overline{\mathtt{T}_\mathtt{A}}\ \mathtt{OK\ IN\ C\ except}\ \overline{\mathtt{m}} & \cap\overline{\mathtt{T}_\mathtt{A}} \setminus \diamond\overline{\mathtt{T}_\mathtt{A}} \subseteq \mathit{meth}(\overline{\mathtt{M}})\end{array}}{\mathtt{T\ OK\ IN\ C\ except}\ \mathit{meth}(\overline{\mathtt{N}}) \cup (\overline{\mathtt{m}} \setminus \mathit{meth}(\overline{\mathtt{M}}))}\ (\text{T}\cdot\text{Ok})$$

$$\frac{\begin{array}{ll}\mathit{altlook}(\mathtt{n},\mathtt{T_A\ with}\ \{\mathtt{m}\,@\,\mathtt{n}\}) = \mathtt{M} & \mathtt{M\ OK\ IN\ C} \\ \mathtt{T_A\ OK\ IN\ C\ except}\ \overline{\mathtt{p}} & \mathtt{n} \notin \mathit{meth}(\mathtt{T_A})\end{array}}{\mathtt{T_A\ with}\ \{\mathtt{m}\,@\,\mathtt{n}\}\ \mathtt{OK\ IN\ C\ except}\ \overline{\mathtt{p}} \setminus \{\mathtt{m}\}}\ (\text{Alias}\cdot\text{Ok 1})$$

$$\frac{\begin{array}{ll}\mathit{altlook}(\mathtt{n},\mathtt{T_A\ with}\ \{\mathtt{m}\,@\,\mathtt{n}\}) = \mathtt{M} & \neg\mathtt{M\ OK\ IN\ C} \\ \mathtt{T_A\ OK\ IN\ C\ except}\ \overline{\mathtt{p}} & \mathtt{n} \notin \mathit{meth}(\mathtt{T_A})\end{array}}{\mathtt{T_A\ with}\ \{\mathtt{m}\,@\,\mathtt{n}\}\ \mathtt{OK\ IN\ C\ except}\ (\overline{\mathtt{p}} \setminus \{\mathtt{m}\}) \cup \{\mathtt{n}\}}\ (\text{Alias}\cdot\text{Ok 2})$$

$$\frac{\mathtt{T_A\ OK\ IN\ C\ except}\ \ \overline{\mathtt{n}} \qquad \mathtt{m} \in \mathit{meth}(\mathtt{T_A})}{\mathtt{T_A\ minus}\ \{\mathtt{m}\}\ \mathtt{OK\ IN\ C\ except}\ \ \overline{\mathtt{n}} \setminus \{\mathtt{m}\}}\ (\text{Exclude}\cdot\text{Ok})$$

Class checking

$$\frac{\begin{array}{l}\mathtt{K} = \mathtt{C}(\overline{\mathtt{D}}\ \overline{\mathtt{g}},\overline{\mathtt{C}}\ \overline{\mathtt{f}})\{\mathtt{super}(\overline{\mathtt{g}});\mathtt{this}.\overline{\mathtt{f}} = \overline{\mathtt{f}};\} \\ \mathit{fields}(\mathtt{D}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}} \qquad \cap\overline{\mathtt{T}_\mathtt{A}} \setminus \diamond\overline{\mathtt{T}_\mathtt{A}} \subseteq \mathit{meth}(\overline{\mathtt{M}}) \\ \overline{\mathtt{M}}\ \mathtt{OK\ IN\ C} \quad \overline{\mathtt{T}_\mathtt{A}}\ \mathtt{OK\ IN\ C\ except}\ \overline{\mathtt{m}} \quad \overline{\mathtt{m}} \subseteq \mathit{meth}(\overline{\mathtt{M}})\end{array}}{\mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\mathtt{K}\ \overline{\mathtt{M}}\ \overline{\mathtt{T}_\mathtt{A}}\}\ \mathtt{OK}}\ (\text{C}\cdot\text{Ok})$$

C The Full Proofs

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

Lemma 1 (Non Virtual Paths)
If $\mathtt{m}$ in $\mathtt{T}_{\mathtt{A}1} \trianglelefteq \mathtt{m}$ in $\mathtt{T}_{\mathtt{A}2}$ *then,* $\mathtt{m} \in \mathit{meth}(\mathtt{T}_{\mathtt{A}1})$ *and* $\mathtt{m} \in \mathit{meth}(\mathtt{T}_{\mathtt{A}2})$.

Proof *By induction on the derivation of* $\mathtt{m}$ in $\mathtt{T}_{\mathtt{A}1} \trianglelefteq \mathtt{m}$ in $\mathtt{T}_{\mathtt{A}2}$.

- (Path·Refl) *by hypothesis of the rule;*
- (Path·Trans) *straightforward since the result is a transitive relation;*
- (Path·Inherit) *by hypothesis of the rule,* $\mathtt{m} \in \mathit{meth}(\mathtt{T}_{\mathtt{A}2})$, *and by rule* (Meth·Trait) $\mathtt{m} \in \mathit{meth}(\mathtt{T}_{\mathtt{A}1})$*;*
- (Path·Exclude) *straightforward by hypothesis of induction plus the rule* (Meth·Exclude).

□

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 2 (Diamond Proto-Soundness)
$\mathtt{m}$ in $\mathtt{T}_{\mathtt{A}1} \trianglelefteq \mathtt{m}$ in $\mathtt{T}_{\mathtt{A}2}$, *then for all* $\mathtt{M}$ *such that* $\mathit{altlook}(\mathtt{m}, \mathtt{T}_{\mathtt{A}2}) = \mathtt{M}$ *is derivable we have* $\mathit{altlook}(\mathtt{m}, \mathtt{T}_{\mathtt{A}1}) = \mathtt{M}$ *is derivable.*

Proof *Let's prove this statement by induction on the derivation of* $\mathtt{m}$ in $\mathtt{T}_{\mathtt{A}1} \trianglelefteq \mathtt{m}$ in $\mathtt{T}_{\mathtt{A}2}$.

- (Path·Refl) *clear since* $\mathtt{T}_{\mathtt{A}1} = \mathtt{T}_{\mathtt{A}2}$*;*
- (Path·Trans) *straightforward since implication is transitive;*
- (Path·Inherit) *since* $\mathtt{m} \notin \mathit{meth}(\overline{\mathtt{M}})$ *the rule* (Alt·Rec) *can apply to* $\mathtt{T}_{\mathtt{A}1}$ *which implies the result;*
- (Path·Exclude) *since* $\mathtt{m} \neq \mathtt{p}$ *the rule* (Alt·NExclude) *can apply to* $\mathtt{T}_{\mathtt{A}1}$ *which implies the result.*

□

This lemma proves that if the program is well-typed, *meth* has the same behavior as *altlook*

Lemma 3 (*meth* Soundness)
If, for any $\mathtt{C}$ *and* $\overline{\mathtt{m}}$, $\mathtt{T}_{\mathtt{A}}$ `OK IN` $\mathtt{C}$ `except` $\overline{\mathtt{m}}$, *then,* $\mathtt{m} \in \mathit{meth}(\mathtt{T}_{\mathtt{A}})$ *if and only if* $\mathit{altlook}(\mathtt{m}, \mathtt{T}_{\mathtt{A}}) \neq$ *fail.*

Proof *By induction on the derivation of* $\mathit{altlook}(\mathtt{m}, \mathtt{T}_{\mathtt{A}})$.

- (Alt·Top) $\mathit{altlook}(\mathtt{m}, \mathtt{T_A}) \neq \mathit{fail}$ *and from rule* (Meth·Trait)*, we have* $\mathtt{m} \in \mathit{meth}(\mathtt{T_A})$*;*
- (Alt·Rec) $\mathit{altlook}(\mathtt{m}, \mathtt{T_A}) \neq \mathit{fail} \iff \exists \mathtt{T_{Ai}} \in \overline{\mathtt{T}}_\mathtt{A}.\ \mathit{altlook}(\mathtt{m}, \mathtt{T_{Ai}}) \neq \mathit{fail}$*. By induction hypothesis, we thus have:*

$$\mathit{altlook}(\mathtt{m}, \mathtt{T_A}) \neq \mathit{fail} \iff \exists \mathtt{T_{Ai}} \in \overline{\mathtt{T}}_\mathtt{A}.\ \mathtt{m} \in \mathit{meth}(\mathtt{m}, \mathtt{T_{Ai}}) \iff \mathtt{m} \in \mathit{meth}(\mathtt{T_A})$$

  Latter is from rule (Meth·Trait) *and the statement* $\mathtt{m} \notin \mathit{meth}(\overline{\mathtt{M}})$*;*
- (Alt·AliasedTo) *since* $\mathtt{T_A}$ *is well typed, we have that* $\mathtt{T_A} = \mathtt{T_{A1}}\ \mathtt{with}\ \{n@m\}$ *where* $\mathtt{n} \in \mathit{meth}(\mathtt{T_{A1}})$ *and* $\mathtt{m} \notin \mathit{meth}(\mathtt{T_{A1}})$*. Then, by induction hypothesis we have that* $\mathit{altlook}(\mathtt{n}, \mathtt{T_{A1}}) \neq \mathit{fail}$ *thus,* $\mathit{altlook}(\mathtt{m}, \mathtt{T_A}) \neq \mathit{fail}$ *and, rule* (Meth·Alias) *states that* $\mathtt{m} \in \mathit{meth}(\mathtt{T_A})$*;*
- (Alt·NAliased) *straightforward;*
- (Alt·Aliased) *straightforward;*
- (Alt·NExclude) *straightforward (using rule* (Meth·Exclude)*);*
- (Alt·Excluded) *straightforward.*

□

Lemma 4 (Conflict Resolution in Trait Alterations)
If $\mathtt{T_A}\ \mathtt{OK\ IN}\ \mathtt{C}\ \mathtt{except}\ \overline{\mathtt{m}}$ *for any* C *and* $\overline{\mathtt{m}}$ *then,* $\mathit{altlook}(\,\cdot\,, \mathtt{T_A})$ *has property* $\mathcal{D}$*.*

Proof *We prove the result by induction on* altlook *derivation.*

- (Alt·Top) *It is enforced by the structure of the calculus;*
- (Alt·Rec) *if* $\mathit{tlook}(\mathtt{m}, \overline{\mathtt{T}}_\mathtt{A}) = \mathit{fail}$ *then the property obviously holds. Else, by induction hypothesis, for all* $\mathtt{T_{Ai}} \in \overline{\mathtt{T}}_\mathtt{A}$*,* $\mathit{altlook}(\,\cdot\,, \mathtt{T_{Ai}})$ *has property* $\mathcal{D}$*;*
  - *if* $\mathtt{m} \notin \cap\overline{\mathtt{T}}_\mathtt{A}$ *then, obviously there is a unique* $\mathtt{T_{Ai}} \in \overline{\mathtt{T}}_\mathtt{A}$ *where* $\mathit{altlook}(\mathtt{m}, \mathtt{T_{Ai}}) \neq \mathit{fail}$*;*
  - *if* $\mathtt{m} \in \cap\overline{\mathtt{T}}_\mathtt{A}$ *then, since* $\mathtt{T_A}$ *is well typed, the rule* (T·Ok) *enforces that* $\mathtt{m} \in \diamond\overline{\mathtt{T}}_\mathtt{A}$*. Then, for all* $\mathtt{T_{Ai}} \in \overline{\mathtt{T}}_\mathtt{A}$ *we have* $\mathtt{m}\ \mathtt{in}\ \mathtt{T_{Ai}} \trianglelefteq \mathtt{m}\ \mathtt{in}\ \mathtt{T_{A1}}$*; Plus, we know that* $\mathtt{m} \in \mathit{meth}(\mathtt{T_{A1}})$ *(by Lemma 1 (Non Virtual Paths)) which means that* $\mathit{altlook}(\mathtt{m}, \mathtt{T_{A1}}) = \mathtt{M}$ *is derivable for at least one* M *(by Lemma 3 (meth Soundness)), thus,* $\mathit{altlook}(\mathtt{m}, \mathtt{T_{Ai}}) = \mathtt{M}$ *is derivable (by Lemma 2 (Diamond Proto-Soundness)). To conclude, we know that* $\mathit{altlook}(\cdot, \mathtt{T_{Ai}})$ *have* $\mathcal{D}$*. That ensures they are all equal;*

  Which obviously gives the result;
- (Alt·AliasedTo) $\mathtt{T_A} = \mathtt{T_A}'\ \mathtt{with}\ \{\mathtt{n}\,@\,\mathtt{m}\}$*. The induction hypothesis ensures that* $\mathit{altlook}(\,\cdot\,, \mathtt{T_A}')$ *has* $\mathcal{D}$*. Then it is straightforward;*
- (Alt·NAliased) *Straightforward as for* (Alt·Aliased)*;*

- (Alt·Aliased) *Clear;*
- (Alt·NExclude) *Straightforward as for* (Alt·Aliased)*;*
- (Alt·Excluded) *Clear.*

□

Theorem 6 (Conflict Resolution)
$\forall \mathtt{C}_i \in \mathtt{CL}$, *such that* $\mathtt{C}_i$ OK, *then both* $mbody(\cdot, \mathtt{C}_i)$ *and* $mytype(\cdot, \mathtt{C}_i)$ *have property* $\mathcal{D}$.

Proof *Let's prove that* $mbody(\cdot, \mathtt{C}_i)$ *has* $\mathcal{D}$ *by induction on the derivation of* $mbody(\mathtt{m}, \mathtt{C}_i)$.

- (MBody·Self) *It is enforced by the structure of the language;*
- (MBody·Super) *Straightforward by induction hypothesis;*
- (MBody·Trait) *for all* $\mathtt{T}_{\mathtt{Ai}} \in \overline{\mathtt{T}}_{\mathtt{A}}$, $altlook(\cdot, \mathtt{T}_{\mathtt{Ai}})$ *has property* $\mathcal{D}$ *by Lemma 4;*
  - *if* $\mathtt{m} \notin \cap\overline{\mathtt{T}}_{\mathtt{A}}$ *then, obviously there is a unique* $\mathtt{T}_{\mathtt{Ai}} \in \overline{\mathtt{T}}_{\mathtt{A}}$ *where* $altlook(\mathtt{m}, \mathtt{T}_{\mathtt{Ai}}) \neq fail$*;*
  - *if* $\mathtt{m} \in \cap\overline{\mathtt{T}}_{\mathtt{A}}$ *then, since* $\mathtt{C}_i$ *is well typed, the rule* (C·Ok) *enforces that* $\mathtt{m} \in \diamond\overline{\mathtt{T}}_{\mathtt{A}}$*. Then, for all* $\mathtt{T}_{\mathtt{Ai}} \in \overline{\mathtt{T}}_{\mathtt{A}}$ *we have* $\mathtt{m}$ in $\mathtt{T}_{\mathtt{Ai}} \trianglelefteq \mathtt{m}$ in $\mathtt{T}_{\mathtt{A1}}$*. Plus, we know that* $\mathtt{m} \in meth(\mathtt{T}_{\mathtt{A1}})$ *(by Lemma 1 (Non Virtual Paths)) which means that* $altlook(\mathtt{m}, \mathtt{T}_{\mathtt{A1}}) = \mathtt{M}$ *is derivable for at least one* M *(by Lemma 3 (meth Soundness)), thus,* $altlook(\mathtt{m}, \mathtt{T}_{\mathtt{Ai}}) = \mathtt{M}$ *is derivable (by Lemma 2 (Diamond Proto-Soundness)). To conclude, we know that* $altlook(\cdot, \mathtt{T}_{\mathtt{Ai}})$ *have* $\mathcal{D}$*. That ensures they are all equal.*

  Which gives the result.

□

In the following, we suppose that the classes are well-typed, so that the theorem 6 holds, and we can address *mbody* and *mytype* as mathematical functions.

Lemma 5 (*mytype*) *If* $mytype(\mathtt{m}, \mathtt{D}) = \overline{\mathtt{C}} \to \mathtt{E}$ *, then* $mytype(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{C}} \to \mathtt{E}$ *for all* $\mathtt{C} <: \mathtt{D}$.

Proof *As in [IPW01] the proof is a straightforward induction on the derivation of* $\mathtt{C} <: \mathtt{D}$.
□

Lemma 6 (Substitution lemma) *if* $\Gamma, \overline{\mathtt{x}}{:}\overline{\mathtt{B}} \vdash \mathtt{e} \in \mathtt{D}$ *and* $\Gamma \vdash \overline{\mathtt{d}} \in \overline{\mathtt{A}}$ *where* $\overline{\mathtt{A}} <: \overline{\mathtt{B}}$*, then* $\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e} \in \mathtt{C}$ *for some* $\mathtt{C} <: \mathtt{D}$.

Proof *The proof is exactly the same as in [IPW01]. By induction on the derivation of* $\Gamma, \overline{\mathtt{x}} : \overline{\mathtt{B}} \vdash \mathtt{e} \in \mathtt{D}$. □

Lemma 7 (Weakening) *If* $\Gamma \vdash \mathtt{e} \in \mathtt{C}$ *, then* $\Gamma, \mathtt{x} : \mathtt{D} \vdash \mathtt{e} \in \mathtt{C}$.

Proof *Straightforward induction (as in [IPW01]).* □

Lemma 8 (Method Body Type)
If $mytype(\mathtt{m},\mathtt{C}) = \overline{\mathtt{B}} \to \mathtt{B}$ *and* $mbody(\mathtt{m},\mathtt{C}) = (\overline{\mathtt{x}},\mathtt{e},\mathtt{D})$ *and* $\mathtt{D} <:_{\mathtt{one}} \mathtt{E}$*, then* $\overline{\mathtt{x}}{:}\overline{\mathtt{B}}, \mathtt{this}{:}\mathtt{D}, \mathtt{super}{:}\mathtt{E} \vdash \mathtt{e} \in \mathtt{C}$.

Proof *We prove the following statement by induction on the derivation of* $mbody(\mathtt{m},\mathtt{C}) = (\overline{\mathtt{x}},\mathtt{e},\mathtt{D})$*. If* $mbody(\mathtt{m},\mathtt{C}) = (\overline{\mathtt{x}},\mathtt{e},\mathtt{D})$*, then* $\mathtt{B}\ \mathtt{m}\ (\overline{\mathtt{B}}\ \overline{\mathtt{x}})\{\mathtt{return}\ \mathtt{e};\}\ \ \mathtt{OK\ IN}\ \mathtt{D}$*, where* $\overline{\mathtt{B}} \to \mathtt{B} = mytype(\mathtt{m},\mathtt{C})$.

- (MBody·Self) *immediate from* (C·Ok) *rule;*
- (MBody·Trait) *straightforward induction on the derivation of the derivation of* $\mathtt{T_A}\ \mathtt{OK\ IN}\ \mathtt{D}\ \mathtt{except}\ \overline{\mathtt{m}}$*;*
- (MBody·Super) *induction case.*

□

Lastly, we are ready to prove the main theorem.

Theorem 7 (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 the same as in [IPW01], the slight modification of Lemma 8 (Method Body Type) does not change the proof.* □

Theorem 8 (Progress)
Suppose e *is a well-typed expression.*

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

Proof *The proof sketch is given in [IPW01]. There is little to add, just remember that we need the Conflict Resolution Theorem to achieve the full proof.* □

Theorem 9 (Reduction preserves safety)
If e *is safe in* Γ*, and* $\mathtt{e} \to \mathtt{e}'$*, then* $\mathtt{e}'$ *is safe in* Γ.

Proof *As in [IPW01], this proof is just similar to Subject Reduction proof.* □

Theorem 10 (Progress of safe programs)
Suppose e *is safe in* Γ*. If* e *has* $(\mathtt{C})\mathtt{new}\ \mathtt{D}(\overline{\mathtt{e}})$ *as a subexpression, then* $\mathtt{D} <: \mathtt{C}$.

Proof *As in [IPW01], it is easy, since the one rule that can apply to derive the type of* $(\mathtt{C})\mathtt{new}\ \mathtt{C_0}(\overline{\mathtt{e}})$ *if* e *is safe is* (Type·UCast) □

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399