Extending FeatherTrait Java with Interfaces

Dedicated to Mario Coppo, Mariangiola Dezani-Ciancaglini, and Simona Ronchi della Rocca on the occasion of their 60th birthday

Luigi Liquori\textsuperscript{a} and Arnaud Spiwack\textsuperscript{b}

\textsuperscript{a}INRIA, France
\textsuperscript{b}ENS Cachan, France

Abstract

In the context of Featherweight Java by Igarashi, Pierce, and Wadler, and its recent extension FeatherTrait Java (FTJ) by the authors, we investigate classes that can be extended with trait composition. A trait is a collection of methods, \textit{i.e.}, behaviors without state; it can be viewed as an “incomplete stateless class” \textit{i.e.}, an interface with some already written behavior. 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 with Interfaces (iFTJ), where traits need to be typechecked only once, which is necessary for compiling them in isolation, and considering them as regular types, like Java-interfaces with a behavioral content.

\textit{Key words:} Object-oriented language design, inheritance, types.

1 Introduction

\textbf{Untyped Traits}, introduced by Schärli, Ducasse, Nierstrasz, Wuyts, and Black [5, 13, 16, 17], have recently emerged as a novel technique for building composable units of behavior in a dynamically-typed language \textit{à la Smalltalk}. Intuitively, a trait is just a collection of methods, \textit{i.e.}, behavior \textit{without} state. Derived traits can be built from an \textit{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 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 \textit{and} behavior, have been originally employed in the pure object-based languages Self [20], or in the language Obliq [4], or for the encoding of classes as records-of-premethods in the Abadi-Cardelli’s Object Calculus [1].

\textit{Preprint submitted to Elsevier}
Contributions. The starting point of this paper is the FeatherTrait Java (FTJ) calculus, by Liquori and Spiwack [10], that conservatively extends the simple calculus of Featherweight Java (FJ) by Igarashi, Pierce, and Wadler [9] with statically typed traits. The main aim of FTJ was to introduce a typed trait-based inheritance in a class-based calculus à la Java. Because of the simplicity of the FTJ calculus, traits could be typechecked only inside a class, thus they needed to be typechecked once for every class. This behavior is not compatible with the idea of compiling traits in isolation. In iFTJ, the traits need only be typechecked once and for all.

(1) We define the FeatherTrait Java with Interfaces calculus (iFTJ), a variation of FTJ and a conservative extension of FJ, which allows traits to be typechecked only once. Traits in iFTJ look like Java-interfaces with some partial behavior inside. An example of what traits can look like is

```scala
trait TA {String p(){return this.r()+this.s()+this.q();}
    String s(){return ‘Java’;};
    String r();
    String q();
}
trait TB {String r(){return ‘Hallo World, my name is’;};
    String s(){return ‘FeatherTrait Java’;};;
}
```

Traits TA and TB are typechecked only once, thus could be compiled in isolation; trait TA “defines” method s, and method p which “requires” methods r and q (declared as interfaces). They can be both imported in a class declaration as follows:

```scala
class Presentation extends Object imports TA TB
  {
    ;Presentation(){super();}
    String ciao(){return this.p();}
    String s(){return ‘FeatherTrait Java with Interfaces,’;};
    String q(){return ‘I hope you will like me’;};}
```

Multiple traits can be imported by one class, and conflicts between common methods, defined in two or more inherited traits, must be resolved explicitly by the user, either by aliasing or excluding method names in traits, or by overriding the conflicted methods in the class that imports those traits or in the trait itself. As such, the evaluation of (new Presentation()).ciao() will produce “Hallo World, my name is FeatherTrait Java with Interfaces, I hope you will like me”.

Typed traits, intended as pure behavior without state à la Schärli et al. [5,16,17], have been introduced by Fisher and Reppy in an object-based core calculus for the Moby language (of the ML [11] family) [7,12]. Then, traits have been immerged in Igarashi, Pierce, and Wadler Featherweight Java by Liquori and Spiwack [10], studied by Smith and Drossopoulou in a Java setting [18], and implemented by Odersky et al. in the class-based language Scala [15], and in the new language Fortress by Allen, Chase, Luchangco, Maessen, Ryu, Steele, and Tobin-Hochstadt [2].
(2) We define a type system for iFTJ that typechecks traits only once, in order to be compatible with compilation in isolation. In a nutshell, every trait is typechecked using a judgment which lists the signatures of methods that are required in order to complete the missing behavior of the trait itself.

**Outline of the paper.** The paper is structured as follows. In Section 2, we quickly review the trait inheritance model adopted in iFTJ. In Section 3, we present the syntax, the operational semantics, and the type system of iFTJ. Section 4 shows the type soundness of iFTJ. Section 5 presents an example of using traits in iFTJ. Section 6 discusses related work and concludes. Appendix A sums up the operational semantics and the type system of iFTJ. Appendix B contains the detailed soundness proof for iFTJ. Because of a lack of space in this volume, a longer version can be found on the authors’ web pages.

2 **Trait Inheritance**

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. Traits syntactically require the methods which are necessary to “complete” their behavior. They can also import other traits, from which they gain both implemented and required methods.

**Conflict Resolution.** When dealing with trait inheritance, conflicts may arise; 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, i.e., 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

1. **Overriding a new method p inside the class.** A new method p is redefined inside the class with a new behavior. The (trait-based) lookup
algorithm will hide the conflict in the traits in favor of the overridden method defined in the class.

class C extends Object imports T1 T2 { ... D p(...){...}}   \( T1 \) and \( T2 \) both define \( p \)
new behavior for \( p \)

(2) **Aliasing the method \( p \) in traits and redefining the method in the class.** The method \( p \) is aliased 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 conflicting).

\[
\text{class C extends Object imports} \\
T1 \{ p@p\_of\_T1 \} \quad \text{aliases } p \text{ with } p\_of\_T1 \\
T2 \{ p@p\_of\_T2 \} \quad \text{aliases } p \text{ with } p\_of\_T2 \\
\{ ... D p(...){...}\} \quad \text{new behavior for } p, \text{ it may use } p\_of\_T1/2
\]

(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.

\[
\text{class C extends Object imports} \\
T1 \text{ (} T2 \text{ minus } \{p\}\text{) } \{ ... \} \quad \text{method } p \text{ is hidden from } T2
\]

3 FeatherTrait Java with Interfaces

In \( \text{iFTJ} \), a program consists of a collection of class and trait declarations, and an expression to be evaluated. We adopt the same notational conventions and hygiene conditions as the \( \text{FJ} \) paper [9], with the metavariables \( S \) and \( I \), ranging over signatures and types, respectively, and the metavariables \( M_\bot, S_\bot, (x,e)_\bot \) ranging over methods (resp. signatures and method bodies) and the special failure value \( \text{fail} \).

\[
\begin{align*}
\text{CL} & : = \text{class } C \text{ extends } C \text{ [imports } TA\text{]}\{I \; F; K \; M\} & \text{Class Declarations} \\
\text{TL} & : = \text{trait } T \text{ [imports } TA\text{]}\{M; S\} & \text{Trait Declarations} \\
I & : = C | T & \text{Types} \\
TA & : = T | TA \text{ with } \{m@m\} | TA \text{ minus } \{m\} & \text{Trait Alterations} \\
K & : = C(I\; F)\{\text{super}(F); \text{this.}F = F;\} & \text{Constructors} \\
M & : = I \text{ m}(I\; x)\{\text{return } e;\} & \text{Methods} \\
S & : = I \text{ m}(I\; x) & \text{Signatures} \\
e & : = x | e.f | e.m(\overline{e}) | \text{new } C(\overline{e}) | (I)e & \text{Expressions}
\end{align*}
\]

Figure 1. Syntax of \( \text{iFTJ} \)

3.1 Syntax and Operational Semantics

The syntax of \( \text{iFTJ} \), presented in Figure 1, extends the syntax of \( \text{FJ} \). An \( \text{iFTJ} \) program is a triple \((CT, TT, e)\) of a class and a trait table, and an expression.
A class \texttt{class C extends C imports $\mathcal{T}$} in \texttt{iFTJ} is composed of field declarations $\mathcal{T}$, a constructor $\mathcal{K}$, some new or redefined methods $\mathcal{R}$, plus a list of imported, possibly altered, traits $\mathcal{TA}$. A trait \texttt{trait T imports $\mathcal{TA} \{M, S\}$} is composed of a list of methods $\mathcal{R}$, some other, possibly altered, traits $\mathcal{TA}$ imported by the trait itself, and a list of abstract method signatures $\mathcal{S}$, which are the methods that aren’t implemented in the trait but are yet required by it. The conflicts are handled by typechecking: all the conflicts must be resolved manually by the program. If any is found during type checking, then the program is rejected. The well-known Snyder’s “diamond problem” \cite{Snyder} is not considered as a conflict, \textit{i.e.} if two traits $\mathcal{T}_1$, $\mathcal{T}_2$ inherit a method $m$ from the same trait $\mathcal{T}_0$, and a trait imports both $\mathcal{T}_1$ and $\mathcal{T}_2$, then the method $m$ from $\mathcal{T}_1$ and the method $m$ from $\mathcal{T}_2$ will not be considered as conflicting as they are both exactly the same. Expressions are the usual ones of \texttt{FJ}. The subtyping rules are essentially the same as those of \texttt{FJ} plus the two rules:

$$
\begin{align*}
\mathcal{TA} &\in \mathcal{TA} \\
\text{trait T imports } &\mathcal{TA} \{\ldots\} \quad \text{(Sub-Tr)} \\
\mathcal{TA} &\in \mathcal{TA} \\
\text{class C extends D imports } &\mathcal{TA} \{\ldots\} \quad \text{(Sub-Cla-Tr)}
\end{align*}
$$

T \:<: \: \text{head}(\mathcal{TA}) \quad \text{C} \:<: \: \text{head}(\mathcal{TA})

The subtyping relation does not only compare classes but also traits. To give an intuition about the above rules, we will remind that \texttt{Java} typing and subtyping is \textit{name-based} (a type is the name of a class or an interface). We intend to stick to this policy in \texttt{iFTJ}. The function \texttt{head} returns the name of the trait which is the \textit{head} of the trait alteration. The rationale is that the alterations do indeed alter the behavioral content of traits, but they do not change their interface\footnote{The rigid type discipline makes \texttt{iFTJ} a proper extension of \texttt{FJ} but not of \texttt{FTJ}.}; another point of view is that alterations transform implemented methods into required methods, both being identified at type level. For instance, an object which inhabits \texttt{class C imports (T minus \{m\})\{\ldots\}} does qualify as being also of type $T$. The other subtyping rules, the simple definition of \texttt{head} and the other standard definition of the operational semantics of \texttt{iFTJ} are collected in Appendix A.

### 3.2 A Virtual Tour Through the Auxiliary Functions

\textbf{The Functions \texttt{meth} and \texttt{sig}.} The function \texttt{meth} has two purposes. The first one, simpler, is to extract the names from either a method declaration (with a body), or a method signature (without a body); it is used in the rules to convert sets of declaration into sets of names. The second purpose is to compute the set of all methods in a class or a trait or a trait alteration which has an \textit{available, real} implementation, not simply a typed interface. Note that the required methods of a trait or trait alteration are not considered by this function. The function \texttt{sig} simply extract the set of the signatures of every method (both implemented and required ones) of a trait or a trait alteration. Both functions are presented in Appendix A.
The Functions \textit{altlook} and \textit{tlook}. The function \textit{altlook} looks up a trait alteration for the complete implementation of a method \( m \) (or fails if it has none, even if there is a declared signature). \textit{altlook} is not a function but it becomes a function for well-typed programs. The function \textit{tlook} is the extension of \textit{altlook} to a set of trait alterations; these two “functions” are mutually recursive. \textit{tlook} is used to find a method in a set of trait alterations; it has no specific strategy to select among multiply defined methods, this is why it is not a function (and subsequently why \textit{altlook} is not a function either). However, typing prevents conflicts, turning both into functions. The most significant rules of \textit{altlook} are

\[
\frac{\text{TT}(T) = \text{trait } T \text{ imports } \overline{TA} \{ \overline{M}; S \} \quad m \not\in \text{meth}(\overline{M})}{\text{altlook}(m, T) = \bot} \quad \text{(ATr-Inh)}
\]

\[
\frac{\text{altlook}(n, TA) = \text{I n}(\overline{T} \overline{x})\{\text{return } e; \}}{\text{altlook}(m, TA \text{ with } \{ m @ n \}) = \text{I m}(\overline{T} \overline{x})\{\text{return } e; \}} \quad \text{(ATr-Ali)}
\]

(ATr-Inh) If the method \( m \) is not provided in the unaltered trait definition, then we look in the imported traits.

(ATr-Ali) 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 it if it exists, or the lookup fails.

The Function \textit{msig}. The function \textit{msig} looks up a trait alteration (similarly to \textit{altlook}) for a method signature \( S \), or fails (in the case where no signature is found, or the method has an available body). When a method \( m \) is \textit{required} by the trait alteration \( TA \), then \( \text{msig}(m, TA) \) returns the \textit{signature} with which \( m \) should be (later) implemented in a class. Note that \textit{msig} is not a function in general, but gets to be one in the case of well-typed programs. The most significant rules of \textit{msig} are

\[
\frac{\text{TT}(T) = \text{trait } T \text{ imports } \overline{TA} \{ \overline{M}; S \} \quad m \not\in \text{meth}(\overline{T})}{\exists TA \in \overline{TA}. \text{msig}(m, TA) = S} \quad m \not\in \text{meth}(\overline{S}) \quad \text{(MSig-Inh)}
\]

\[
\frac{\forall TA \in \overline{TA}. \text{msig}(m, TA) = \text{fail}}{\text{msig}(m, T) = \text{fail}} \quad \text{(MSig-End)}
\]

\[
\frac{m \neq n}{\text{altlook}(m, TA) = \text{I m}(\overline{T} \overline{x})\{\text{return } e; \}} \quad \text{(MSig-Ali)}
\]

\[
\frac{\text{msig}(m, TA \text{ with } \{ m @ n \}) = \text{I m}(\overline{T} \overline{x})}{\text{msig}(m, TA \text{ with } \{ m @ n \}) = \text{I m}(\overline{T} \overline{x})} \quad \text{(MSig-Ali)}
\]

(MSig-Inh) & (MSig-End) When looking up a required method signature in an unaltered trait, if the method is not spoken of in the trait, then we look in the imported traits. If none of them requires it, then the lookup fails. Note that we ensure that the method is not implemented in any of the imported traits.

(MSig-Ali) Those rules apply when looking up a required method \( m \) signature in a trait alteration where \( m \) is aliased to \( n \). We look whether or not \( m \) exists in
the head trait alteration with an implementation. If it does, then it becomes required (we do not change the type interface of the trait alteration even through alterations), otherwise the lookup fails.

**The Function \textit{mtype}.** As above, \textit{mtype} is a function only for well-typed programs. It fetches in a class or in a trait alteration the type of \textit{m} which can be either implemented or required. It is used in the typing expressions like \texttt{e.m(\ldots)}. Whether \textit{m} is required or implemented does not matter. Concrete objects (\texttt{new \ldots}) are instances of a class; type soundness ensures that classes implement all methods required by the traits they import. As far as the typing of expressions is concerned, traits are like interfaces. The most significant rules of \textit{mtype} are

\[
\frac{\text{CT(C)} = \text{class C extends D imports } \mathcal{TA} \{ \mathcal{J} \mathcal{F} ; \mathcal{K} \mathcal{N} \} }{\text{mtype}(m,C) = \mathbb{I} \to \mathbb{I} } \quad \text{(MTyp-Tr)}
\]

\[
\frac{\text{CT(C)} = \text{class C extends D imports } \mathcal{TA} \{ \mathcal{J} \mathcal{F} ; \mathcal{K} \mathcal{N} \} }{\text{m \notin \text{meth}(\mathcal{N}) \quad \forall \mathcal{TA} \in \mathcal{TA} \quad \text{mtype}(m,\mathcal{TA}) = \text{fail} \quad \text{mtype}(m,D) = \mathbb{I} \to \mathbb{I} }} \quad \text{(MTyp-Super)}
\]

\[
\text{altlook}(m,\mathcal{TA}) = \text{fail} \quad \text{msig}(m,\mathcal{TA}) = \mathbb{I} \quad \text{mtype}(m,\mathcal{TA}) = \mathbb{I} \to \mathbb{I} \quad \text{(MTyp-Virt)}
\]

\[
\text{(MTyp-Tr)} \quad \text{If a class has not declared a method explicitly, then we first lookup the method type inside the imported traits.}
\]

\[
\text{(MTyp-Super)} \quad \text{If the method is not declared in the imported traits (either implemented or required), then we lookup inside the superclass.}
\]

\[
\text{(MTyp-Virt)} \quad \text{This rule applies when looking up for a method type in a trait alteration. If the method is not implemented in the trait alteration, then we look whether it is required by the trait alteration, giving the appropriate type. Thus, traits behave more like interfaces than classes.}
\]

**The Functions \textit{fields} and \textit{mbody}.** Those functions are almost unmodified since \texttt{FJ}. The function \textit{fields} simply computes the set of the fields of a class (this includes those of the superclass) together with their types. \textit{mbody} is a function only for well-typed programs. The function \textit{mbody} performs the method body lookup: given a method \textit{m} and a class \textit{C}, it browses the inheritance tree of \textit{C} until it finds the body of \textit{m}. The most significant rules of \textit{mbody} are

\[
\text{CT(C)} = \text{class C extends D imports } \mathcal{TA} \{ \mathcal{J} \mathcal{F} ; \mathcal{K} \mathcal{N} \}
\]

\[
\frac{\text{m \notin \text{meth}(\mathcal{N}) \quad tlook(m,\mathcal{TA}) = \mathbb{I} \quad m(\mathcal{I} \mathcal{X})\{\text{return e;}\} }}{\text{mbody}(m,C) = (\mathbb{x},e) } \quad \text{(MBdy-Tr)}
\]

\[
\text{CT(C)} = \text{class C extends D imports } \mathcal{T} \{ \mathcal{J} \mathcal{F} ; \mathcal{K} \mathcal{N} \}
\]

\[
\frac{\text{m \notin \text{meth}(\mathcal{N}) \quad tlook(m,\mathcal{TA}) = \text{fail} \quad mbody(m,D) = (\mathbb{x},e)_{\bot} }}{\text{mbody}(m,C) = (\mathbb{x},e)_{\bot} } \quad \text{(MBdy-SCla)}
\]
(MBdy-Tr) If the method is not declared in the class, then we first look it up in the imported traits.

(MBdy-SCla) If the method is not in any of the imported traits, then we look it up in the superclass.

**Method Path relation** \( \preceq \). This relation is related to “diamond” (or “fork-join”) conflicts arising when a class/trait, that inherits from two classes/traits, would ostensibly have two distinct definitions for one method [19]. The set \( \cap \overline{TA} \) denotes methods defined in more than one trait; it is used to detect conflicts when importing traits. The set \( \circ \overline{TA} \) denotes methods that potentially determine a diamond when dealing with trait inheritance; such methods are expected to be “non-conflicting”, hence accepted by the type system. More precisely: the set \( \cap \overline{TA} \) detects every conflict in \( \overline{TA} \), while the set \( \circ \overline{TA} \) detects every diamond. A class declaration is well-formed only if the imported trait alterations imported by the class \( C \) satisfy the constraint \( \cap \overline{TA} \cap \circ \overline{TA} \subseteq \text{meth}(\overline{N}) \), ensuring that every conflict is resolved, *i.e.*, every new-born conflict \( (\cap \overline{TA}) \) which is not a diamond \( (\circ \overline{TA}) \) is being overridden.

\[
\cap \overline{TA} \overset{\text{def}}{=} \{ m \mid \exists \ TA_1 \neq TA_2 \in \overline{TA}. m \in \text{meth}(TA_1) \cap \text{meth}(TA_2) \} \\
\circ \overline{TA} \overset{\text{def}}{=} \{ m \mid \exists n, TA_1. \forall TA_2 \in \overline{TA}. m \in \text{meth}(TA_2) \implies m \text{ in } TA_2 \subseteq n \text{ in } TA_1 \}
\]

To compute \( \circ \overline{TA} \), we need a judgment of the form \( m \text{ in } TA_1 \triangleleft m \text{ in } TA_2 \) (read “m of TA_1 behaves exactly as n of TA_2”). The meaning is as follows: m is a method provided by trait TA_1, whose implementation is inherited from that of a method n provided by TA_2 through any number of trait declarations or alteration steps (paths). The most significant rules are

\[
\begin{array}{c}
\text{TT}(T)=\text{trait } T \text{ imports } \overline{TA} \{M; S\} \\
\text{TA} \in \overline{TA} \quad m \in \text{meth}(TA) \setminus \text{meth}(N) \\
\frac{\text{m in } T \triangleleft m \text{ in } TA}{p \text{ in } TA_1 \triangleleft n \text{ in } TA_2} \quad (\text{Path-Inh}) \\
\frac{\text{m in } TA_1 \triangleleft n \text{ in } TA_2 \quad m \neq p \quad m \neq q}{\text{m in } TA_1 \triangleleft n \text{ in } TA_2 \quad m \neq p} \quad (\text{Path-Ali1}) \\
\frac{\text{m in } TA_1 \triangleleft n \text{ in } TA_2 \quad \{p \circ q\} \subseteq n \text{ in } TA_2}{m \text{ in } TA_1 \text{ minus } \{p\} \triangleleft n \text{ in } TA_2} \quad (\text{Path-Exl})
\end{array}
\]

- (Path-Inh) If a trait T inherits a method \( m \) directly from a trait alteration \( \overline{TA} \) and does not override it, then \( m \) of T behaves exactly as \( m \) of TA.
- (Path-Ali1) If \( p \) of TA_1 behaves exactly as n of TA_2, then \( m \) of TA_1 with \( \{p \circ m\} \) behaves exactly as n of TA_2.
- (Path-Ali2) If \( m \neq p \) and \( m \neq q \), and \( m \) of TA_1 behaves exactly as n of TA_2, then \( m \) of TA_1 with \( \{p \circ q\} \) behaves exactly as n of TA_2.
- (Path-Exl) If \( m \neq p \), and \( m \) of TA_1 behaves exactly as n of TA_2, then \( m \) of TA_1 minus \( \{p\} \) behaves exactly as n of TA_2.

### 3.3 The Type System

We show the most important rules of \( \text{iFTJ} \) type system which allow to typecheck traits only once. The remaining rules are presented in Appendix A. The type system has three steps: first, expressions must be typed using
standard judgments of the form \( \Gamma \vdash e \in I \). Second, methods must be typed inside a class using judgments of the form \( M \text{ OK IN } C \) or inside a trait. Since a trait is essentially a Java-interface with some behavior inside, it has a type of its own. The associated judgment \( M \text{ OK IN } T \) (and typing rules) are similar. Judgments \( S \text{ OK IN } T \) and \( S \text{ OK IN } C \) hold to guarantee that signatures are compatible. Next, altered traits must be typed using judgments of the form \( TA \text{ OK requires } S \). The intuition behind the \text{requires } S part is that the implementation of the \( S \) methods must be available in the classes which import \( TA \) with the given signature. The implementation may be given either by an explicit declaration in the body of the class (or trait), or inherited by the superclass or by another trait. Signatures are considered equal modulo renaming of their arguments. Finally, trait and class typechecking are performed only once. Checking classes and traits is done via judgments of the form \( TL \text{ OK requires } S \) and \( CL \text{ OK} \) where the trait and class tables \( TT \) and \( CT \) are left implicit in the judgments. Like trait alterations, trait declaration checking gives also the signature of the abstract methods.

**Method typechecking** is defined as follows

\[
\begin{align*}
CT(C) &= \text{class } C \text{ extends } D \text{ imports } TA \{J, K, M\} \\
\varpi, \varpi, \text{this:C} &\vdash e \in J \quad J <: I \\
\text{override}(m, D, \varpi, \varpi, I) &\quad \text{override}(m, TA, \varpi, \varpi, I) \\
\hline
\text{I m(\varpi \varpi )\{return e;\} OK IN C} \tag{Mth-Ok-Cla}
\end{align*}
\]

\[
\begin{align*}
\varpi, \varpi, \text{this:T} &\vdash e \in J \quad J <: I \\
\text{override}(m, TA, \varpi, \varpi, I) &\\
\hline
\text{I m(\varpi \varpi )\{return e;\} OK IN T} \tag{Mth-Ok-Tr}
\end{align*}
\]

(Mth-Ok-Cla) We first ensure that the method body \( e \) is typable with a type compatible with its declared signature, \textit{i.e.} that if the method body has type \( J \), then \( J \) is smaller (possibly equal) than the declared type \( I \). We then check the two override conditions, \textit{i.e.} we check that if the method name \( m \) is used in any of the imported trait or in the superclass, then it is used with the same type as this method.

(Mth-Ok-Tr) This rule behaves as the previous (Mth-Ok-Cla) rule; it is interesting to remark that the type assigned to the pseudovariable \text{this} is the trait \( T \) itself, which is considered as a \textit{real type}.

Simpler rules apply to check method signatures in a trait or in a class.

**The trait alteration typing.** These rules derive judgments of the form \( TA \text{ OK requires } S \) which means that \( TA \) is well-typed where every method declared in the signature \( S \) must be implemented. The rationale is that every method occurring in the require part refers to a method that is not (or no more) implemented in the trait but is \textit{needed in order to complete the trait}. 

9
TA OK requires \( m \in \text{meth}(TA) \)
\[ n \not\in \text{meth}(TA) \cup \text{meth}(S) \]
\[ \text{ntype}(m, TA) = I \rightarrow I \]
\[(\text{Alias-Ok}_1)\]

TA with \( \{m \circ n\} \) OK requires \( \mathcal{S} \cup \{I \ m(I \ x)\} \)
\[ \mathcal{S} \subseteq \text{meth}(TA) \]
\[ n \not\in \text{meth}(TA) \cup \text{meth}(S) \]
\[ \text{ntype}(m, TA) = I \rightarrow I \]
\[ \{I \ m(I \ x)\} \]
\[(\text{Alias-Ok}_2)\]

TA OK requires \( \mathcal{S} \)
\[ m \in \text{meth}(TA) \]
\[ n \not\in \text{meth}(TA) \cup \text{meth}(S) \]
\[ \text{ntype}(m, TA) = I \rightarrow I \]
\[(\text{Exlude-Ok})\]

(Alias-Ok) This rule handles the typechecking of aliasing where the new name \( n \) is not a required method. It checks that the new method name \( n \) does not correspond to a defined method in \( TA \), and that the method name being aliased \( m \) exists. Then, simply adds the method \( m \) to the required methods.

(Alias-Ok\(_2\)) Behaves as (Alias-Ok\(_1\)), except that the aliased method name \( m \) takes the place of a required method and that the new method name \( n \) must be removed from the list (both must have the same type interface).

(Exlude-Ok) Adds the aliased method \( m \) to the required methods, in case another method calls \( m \).

The class and trait typing rules are defined as follows

\[ \cap \text{TA} \setminus \circ \text{TA} \subseteq \text{meth}(\mathcal{R}) \]
\[ \mathcal{M} \text{ OK IN } T \]
\[ \text{TA OK requires } \mathcal{S} \]
\[ \text{requires } (\text{sig}(\text{TA}) \cup \mathcal{S}) \text{ OK IN } T \]
\[ (\text{Tr-Ok}) \]

\[ K = C(J \ g, I \ f) \{ \text{super}(g); \text{this} = I; \} \]
\[ \text{fields}(D) = J \ g \]
\[ \cap \text{TA} \setminus \circ \text{TA} \subseteq \text{meth}(\mathcal{R}) \]
\[ \text{sig}(\text{TA}) \text{ OK IN } C \]
\[ \mathcal{R} \text{ OK IN } C \]
\[ \text{TA OK requires } \mathcal{S} \]
\[ \text{meth}(\mathcal{S}) \subseteq \text{meth}(C) \]
\[ (\text{Cla-Ok}) \]

Intuitively, those two rules check that all the components of the class and of the trait are well-typed, and that all conflicts are resolved; the trait rule also builds the list of methods that are required but not provided in the trait. In those rules, for \( \text{TT}(T) = \text{trait } T \text{ imports } \text{TA} \{\ldots\} \) we have a judgment \( \mathcal{S} \text{ OK IN } T \) basically meaning that the methods whose signature are in \( \mathcal{S} \) do not raise a typing conflict with \( T \). It is used to check that the imported traits are compatible (a class cannot import a trait with a method \( m \) returning an integer and another trait with a method \( m \) returning a string, for instance).

(Tr-Ok)/(Cla-Ok)

- (only in (Cla-Ok)) We fetch the constructor \( K \) and the fields \( g \).
- We typecheck the set of altered traits \( \text{TA} \) producing a set of required methods (the requires \( \mathcal{S} \) part).
We typecheck the methods $M$ inside $T/C$.

We check the key condition $\cap TA \setminus \sigma TA \subseteq \text{meth}(M)$ ensuring that every conflict is resolved, and guaranteeing that the lookup algorithm provides the correct conflict resolution.

We typecheck the method signatures in $\overline{TA}$ and $\overline{S}$. We check that all the traits $\overline{TA}$ and the signature $\overline{S}$ are compatible in the trait $T$ (resp. $\overline{TA}$ in the class $C$), i.e. they can be pairwise composed without any conflict in the method types (for instance, type conflicts may arise between an existing and an abstract method).

(only in (Tr-Ok)) The set of required methods in $T$, is then the set $\overline{S}'$ of all methods signature required by the imported trait alterations $\overline{TA}$ plus the set $\overline{S}$ declared in $T$ except all the methods that are defined (i.e. implemented) in $T$. It is worth noticing here that the user can define more method signatures in $\overline{S}$ than what is really needed; in other words: if a method has been declared in $\overline{S}$ but its behavior is already present inside an imported trait, then the system only checks that both the type of the defined method and of the required one are compatible.

(only in (Cla-Ok)) We check the condition $\text{meth}(\overline{S}) \subseteq \text{meth}(C)$. It means that classes have to provide all the necessary methods for the computation of its instances (objects of the form $\text{new } C(\ldots)$). Specifically, $\text{meth}(\overline{S})$ are being implemented either inside the class, in the superclass or inside some trait (other than the ones requiring it). As such, we ensure that every time a method $m$ is expected for $C$ then it is also implemented in $C$.

4 Properties

Once the type system for $iFTJ$ has been set up, the next step is to prove that $(i)$ the static semantics matches the dynamic one, i.e., 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 programs from the run-time $\text{message-not-understood}$ error. The proofs of these statements are not excessively more complicated than in $FJ$. The full proofs are provided in Appendix B.

Untypable programs are not necessarily deterministic, since the lookup rules do not give the priority to any of the imported trait (so that the order in which traits are importer does not change the semantics). Conflict Resolution Theorem states that typed programs have a deterministic lookup algorithm. Which means nothing more than saying that all conflicts have been resolved.

**Theorem 1 (Conflict Resolution)**

*If, for all $C_i \in CL$, we have $C_i \text{ OK}$, then both $mbody$ and $mtype$ are functions.*

Subject reduction follows easily.
Theorem 2 (Subject Reduction)
If $\Gamma \vdash e \in C$ and $e \rightarrow e'$, then $\Gamma \vdash e' \in D$, for some $D <: C$. $\square$

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 [9]).

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

1. If $e$ includes new $C(f).f$ as a subexpression, then \( \text{fields}(C) = T \) and $f \in T$;
2. If $e$ includes new $C(m).m(f)$ as a subexpression, then $\text{mbody}(m.C) = (x, e_0)$ and $\#(x) = \#(d)$.

In accordance with FJ, we define the notion of safe expression $e$ in $\Gamma$ if the type derivation of the underlying $(CT, TT)$ and $\Gamma \vdash e \in C$ contains no downcast or stupid cast (rules (Typ-DCast), and (Typ-SCast)). Then, soundness of safety and progress of safe programs follow.

Theorem 4 (Reduction preserves safety)
If $e$ is safe in $\Gamma$, and $e \rightarrow e'$, then $e'$ is safe in $\Gamma$. $\square$

Theorem 5 (Progress of safe programs)
Suppose $e$ is safe in $\Gamma$. If $e$ has (C)new $D(x)$ as a subexpression, then $D <: C$. $\square$

5 Example

We give a small example in Figure 2, using the Java class Integer enriched with some simple algebraic methods, e.g. mod and times. Here is a sum-up of this example. First we define a trait Convertible which is purely abstract (as an interface in Java, we define it only for typing purposes); it requires a method producing an integer. Then, we declare a trait Hashable that imports Convertible, and uses the to_Int method as an input for its hashing function. It allows then to define an extension $H_{Integer}$ of the class Integer with a method hash to get a hash value of the considered integer. Independently, we define a trait Convertible_Pair that imports Convertible (thus every Convertible_Pair is also Convertible) and define a method to_Int for a pair of two convertible object. Finally, we define our strings as lists of characters. The strings are indeed subclasses of a trait $My\_String$, which is an interface equivalent to Hashable (in real life it would be a strict subtype of Hashable though). As such, we have two classes that build strings, namely $Null\_String$, i.e. a single element class, and $Cons\_String$ which is intrinsically a pair of a character and a string. The class $Cons\_String$ imports Convertible_Pair to implement the method to_Int which is required by $My\_String$ (as a trait importing Hashable). Then, we can assume a class $Array$, which can be used as an array or an association table. Together with the hashing function of strings (or that of hashable integers) it may be used as a hash table, as suggested
trait Convertible {;Integer to_Int( )
trait Hashable imports Convertible {
    Integer hash(){... this.to_Int() ...};
} trait Convertible_Pair imports Convertible {
    Integer to_Int(){return this.fst().to_Int().
        plus(this snd().to_Int()).times(this.offset());};
    Convertible fst()
    Convertible snd()
    Integer offset()
} trait My_String imports Hashable { ; }

class H_Integer extends Integer imports Hashable {
    H_Integer() {super();}
    Integer to_Int() {return this;}
}
class My_Char extends Objects imports Convertible {
    int me;
    My_Char(Integer c) {super(); this.me=c.mod(new Integer(256));}
    Integer to_Int() {return this.me;}
}
class Cons_String extends Objects imports My_String Convertible_Pair{
    My_Char head
    My_String tail;
    Cons_String(My_Char c,My_String s) {
        super();
        this.head=c; this.tail=s;
    }
    Convertible fst() {return this.head;}
    Convertible snd() {return this.tail;}
    Integer offset() {return new Integer(256);}
}
class Null_String extends Objects imports My_String {
    Null_String() {
        super();
    }
    Integer to_Int() {return new Integer(0);}
}
class Array extends Objects imports H_Integers {
    Object index(Integer i) {...}
    Object assoc(Integer i) {...}
}

(Array)(new Array(...).index(new Cons_String(...).hash()))
    .assoc(new Cons_String(...))

Figure 2. Hashing Strings

in the example. Moreover, H_Integer, Cons_String, and Null_String do not only share the Hashable type, but they also have a common implementation of the method hash gotten through trait inheritance.

6 Related Work and Conclusions

Related Work. In the past few years, great interest was recorded around the possibility to use trait inheritance in statically typed class- and object-based languages [10,13,14,18]. Among the many propositions which arose in the literature (and apart our FTJ), we recall the following ones.
(TcoreMoby) adds statically typed trait inheritance to an object-based calculus with first-class functions of the ML family. Fisher and Reppy have the same interest in typed traits as we do, and historically this paper can be considered as the first attempt to typecheck traits statically. 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 need to be typechecked only once. The paper comes with the full set of proofs. Our iFTJ relaxes point (a) and features points (b) and (c).

(Chai) adds statically typed trait inheritance to a Java-like language; in fact there are three dialects defined: Chai₁,₂,₃. As for TcoreMoby, the key points in Chai are that (a) two traits can be combined only if they are disjoint, (b) one method can be overridden by another only if it has the same type interface, and (c) in Chai₂,₃ traits are typechecked only once, and that (d) in Chai₃ traits can be substituted for one another dynamically. The paper comes with proof sketches for the theorems of Chai₁, and soundness theorems for Chai₂,₃, whose proofs are not yet published. Our iFTJ can be compared with Chai₂: the bigger difference is that iFTJ relaxes point (a), by allowing conflicts to be resolved via overriding, and features points (b) and (c), making the type system more expressive than Chai₂. Moreover, iFTJ comes with a full metatheory.

(Scala) 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 [8], 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 website. A Featherweight Scala formal model with related meta-theory remains to be fleshed out (and a formal comparison of features also).

(Fortress) specification language by Allen, Chase, Luchangco, Maessen, Ryu, Steele, and Tobin-Hochstadt was published on SUN’s website 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.

However, our type system is deeply indebted to the work on incomplete objects by Bono, Bugliesi, Dezani, and Liquori [3]; this work presented a type system for the Lambda Calculus of Objects of Fisher, Honsell, and Mitchell [6], an untyped λ-calculus enriched with object primitives. The paper allowed objects to be typed independently of the order of their method additions. This flexibility arises from introducing the notion of completion, a complement to interface, to convey information on (the types of) the methods which are not available in the object, and yet are referenced by its the methods. Besides allowing a more flexible typing of methods (in particular, of mutually
recursive method definitions), this extension also allows methods to be invoked on incomplete objects, i.e. objects whose implementation (the set of their methods) is only partially specified. The paper conjectured that the concept underpinning the typing of incomplete objects may be exploited in modeling language constructs such as virtual methods and interfaces in class-based languages (exactly what our model of typed trait does). Ten year later, we found some evidence in our conjecture in designing a type system for iFTJ.

Conclusions. We have presented a formal development of iFTJ, a statically typed, functional, class-based language featuring classes, objects, trait inheritance where traits need to be typechecked only once in order to make the system compatible with separated compilation. Future directions will focus on:

- Add bounded polymorphic-types or even generic-types as in GJ [9]; this extension will greatly improve the usefulness of statically typed traits.
- Study 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.

Acknowledgments. (By the first author, in italian.) Cari Mario, Mariangiola e Simona, senza i vostri insegnamenti non saprei quello che so e non sarei quello che sono ...

References


A Dynamic and Static Semantics of iFTJ

**Head and Subtyping** (Sub-Tr) and (Sub-Cla-Tr) plus

head(T) = T  head(TA with {m : n}) = head(TA)  head(TA minus {m}) = head(TA)

\[ \frac{I_1 <: I_2 \quad I_2 <: I_3}{I_1 <: I_3} \]  (Sub-Ref)

\[ \frac{C <: I}{\text{class C extends D imports } TA \{ \ldots \}} \]  (Sub-Cla)

**Small-step semantics**

**fields(C) = \( I \Sigma \)** (Run-Field)

\[ (\text{new } C(\Sigma)).f_i \longrightarrow e_i \]  (Run-Field)

\[ C <: I \quad \frac{I(\text{new } C(\Sigma)) \longrightarrow \text{new } C(\Sigma)}{\text{Run-Cast}} \]

\[ mb(m, C) = (\Sigma, e_0) \]  (Run-Call)

\[ \text{new } C(\Sigma).m(\Sigma) \longrightarrow [I/\Sigma, \text{new } C(\Sigma)/\text{this}]e_0 \]

**Congruence**

\[ e \longrightarrow e' \]  (Cgr-Field)

\[ e.f \longrightarrow e'.f \]  (Cgr-Field)

\[ e \longrightarrow e' \]  (Cgr-Receiver)

\[ e.m(\Sigma) \longrightarrow e'.m(\Sigma) \]  (Cgr-Receiver)

\[ e \longrightarrow e' \]  (Cgr-Cast)

\[ (I)e \longrightarrow (I)e' \]  (Cgr-Cast)

\[ e_1 \longrightarrow e'_1 \]  (Cgr-Args)

\[ e.m(\ldots, e_i, \ldots) \longrightarrow e.m(\ldots, e'_i, \ldots) \]  (Cgr-Args)

\[ \text{new } C(\ldots, e_i, \ldots) \longrightarrow \text{new } C(\ldots, e'_i, \ldots) \]  (Cgr-New)

**Field lookup** exactly as in FJ

**Method body lookup** (MBdy-Tr) and (MBdy-SCla) plus

CT(C) = class C extends D imports TA \{ T T ; K M \}

\[ I(m(T x)\{ \text{return } e; \}) \in M \]

\[ mb(m, C) = (\Sigma, e) \]  (MBdy-Cla)

**Trait lookup**

\[ \exists TA \in TA. \text{altlook}(m, TA) = M \]  (Tr-Ok)

\[ t\text{look}(m, TA) = M \]  (Tr-Ok)

\[ \forall TA \in TA. \text{altlook}(m, TA) = \text{fail} \]  (Tr-Ko)

\[ t\text{look}(m, TA) = \text{fail} \]  (Tr-Ko)

**Trait alteration lookup** (ATr-Inh) and (ATr-Ali_1) plus

\[ \text{TT}(T) = \text{trait } T \text{ imports } TA \{ H, S \} \quad I(m(T x)\{ \text{return } e; \}) \in M \]

\[ \text{altlook}(m, TA) = M \]  (ATr-Inh)

\[ \text{altlook}(m, TA with \{ p \rightarrow q \}) = M \]  (ATr-Inh)

\[ m \neq p \quad m \neq q \quad \text{altlook}(m, TA) = M \]  (ATr-Ali_1)

\[ \text{altlook}(m, TA with \{ p \rightarrow q \}) = M \]  (ATr-Ali_1)
\[
\begin{align*}
&\text{Method names and signatures} \\
&\text{meth}(\text{I m(Î…)}) = \{m\} \\
&\text{TT} \cdot \text{T} = \text{trait T imports TA \{M; S\}} \\
&\text{meth(T)} = \text{meth(M)} \cup \text{meth(TA)} \\
&\text{sig}(\text{TA}) = \{S, I \text{ m(Î…)}\} \\
&\text{TT} \cdot \text{T} = \text{trait T imports TA \{M; S\}} \\
&\text{sig(T)} = S \cup \text{sig(M)} \cup \text{sig(TA)} \\
\end{align*}
\]

\[
\begin{align*}
&\text{Method paths in trait alterations (Path\{-\text{Inh, Alias1,2, Exc}\}) plus} \\
&\text{m in TA} \trianglelefteq m \text{ in TA} \\
&\text{m in TA} \trianglelefteq p \text{ in TA} \\
&\text{p in TA} \trianglelefteq n \text{ in TA} \\
&\text{m in TA} \trianglelefteq n \text{ in TA} \\
\end{align*}
\]

\[
\begin{align*}
&\text{Method type lookup} \\
&\text{I m(Î…)} \{\text{return e;}\} \in M \\
&\text{mtype(m, C)} = I \rightarrow I \\
\end{align*}
\]
\[
\text{altlook}(m,TA) = I \ m(\overline{T} x) \{ \text{return } e; \} \\
\]
\[
\text{ntype}(m,TA) = \overline{T} \rightarrow I \\
\]
\[
\text{altlook}(m,TA) = \text{fail} \quad \text{msig}(m,TA) = \text{fail} \\
\text{ntype}(m,TA) = \text{fail} \\
\]

\[\text{Signature lookup and overriding} \quad (\text{MSig}\{-\text{Inh, End, Ali}_3\}) \text{ plus} \]
\[
\text{TT}(T) = \text{trait } T \text{ imports } \overline{TA} \{ \overline{R}; \overline{S} \} \quad m \notin \text{meth}(T) \quad \text{I m}(\overline{T} x) \in \overline{S} \\
\text{msig}(m,T) = I m(\overline{T} x) \\
\]
\[
\text{TT}(T) = \text{trait } T \text{ imports } \overline{TA} \{ \overline{R}; \overline{S} \} \quad m \notin \text{meth}(T) \\
\forall \overline{TA} \in \overline{TA}, \text{msig}(m,TA) = \text{fail} \quad m \notin \text{meth}(\overline{S}) \\
\text{msig}(m,T) = \text{fail} \\
\]
\[
m \neq p \quad m \neq q \quad \text{msig}(m,TA) = S_\bot \\
\text{msig}(m,TA \text{ with } \{p \circ q\}) = S_\bot \\
\]
\[
m \neq n \\
\text{msig}(m,TA \text{ with } \{n \circ m\}) = \text{fail} \\
\text{msig}(m,TA \text{ minus } \{m\}) = \text{fail} \\
\]
\[
m \neq n \quad \text{msig}(m,TA \text{ minus } \{n\}) = S_\bot \\
\text{msig}(m,TA \text{ minus } \{m\}) = \text{fail} \\
\]
\[
\text{altlook}(m,TA) = I \ m(\overline{T} x) \{ \text{return } e; \} \\
\text{altlook}(m,TA) = \text{fail} \\
\text{msig}(m,TA) = \text{fail} \\
\]
\[
\text{Valid Method Overriding} \\
\text{ntype}(m,I) = J \rightarrow J_0 \text{ implies } \overline{T} = J \text{ and } I_0 = J_0 \\
\text{override}(m,I,\overline{T} \rightarrow I_0) \\
\]
\[
\text{ override}(m,\overline{TA},\overline{T} \rightarrow I) \\
\text{ msig}(m,TA) = \text{fail} \\
\text{override}(m,\overline{TA},\overline{T} \rightarrow I) \\
\]
\[\text{Basic expression typing} \text{ exactly as in FJ} \]
\[\text{Method typing} \quad (\text{Tr-Ok}) \text{ and } (\text{Cla-Ok}) \text{ plus} \]
\[
\text{TT}(T) = \text{trait } T \text{ imports } \overline{TA} \{ \overline{R}; \overline{S} \} \quad \text{override}(m,\overline{TA},\overline{T} \rightarrow I) \\
\text{I m}(\overline{T} x) \text{ OK IN } T \\
\]
\[
\text{CT(C)} = \text{class } C \text{ extends } D \text{ imports } \overline{TA} \{ \overline{T} \overline{F}; \overline{K} \overline{R} \} \\
\text{override}(m,D,\overline{T} \rightarrow I) \\
\text{I m}(\overline{T} x) \text{ OK IN } C \\
\]
B  The Full Proofs

We prove that a method path relation only designs paths for existing methods.

Lemma 1 (Non Virtual Paths)
If \( m \in TA_1 \trianglelefteq n \in TA_2 \) then, \( m \in \text{meth}(TA_1) \) and \( n \in \text{meth}(TA_2) \).

Proof  By induction on the derivation of \( m \in TA_1 \trianglelefteq n \in TA_2 \).

We show that \text{altlook} provides a method implementation with the proper name.

Lemma 2 (Naming Soundness)
\begin{itemize}
  \item If \( \text{altlook}(m,TA) = M \bot \), then either \( M \bot = \text{fail} \), or \( M \bot = \text{Im}(I;x)\{\ldots\} \).
  \item If \( \text{msg}(m,TA) = S \bot \), then either \( S \bot = \text{fail} \), or \( S \bot = \text{Im}(I;x)\{\text{return } e;\} \).
\end{itemize}

Proof  Straightforward induction on the derivation of \( \text{altlook}(m,TA) = M \bot \).

We prove that a method path relation preserves the body of the method. It is
the first step for proving determinism of well-typed programs.

Lemma 3 (Diamond Proto-Soundness)
If \( m \in TA_1 \trianglelefteq n \in TA_2 \), then \( \text{altlook}(n,TA_2) = \text{Im}(I;x)\{\text{return } e;\} \) implies \( \text{altlook}(m,TA_1) = \text{Im}(I;x)\{\text{return } e;\} \).

Proof  By induction on the derivation of \( m \in TA_1 \trianglelefteq n \in TA_2 \). Here are the most relevant cases
\begin{itemize}
  \item (Path·Inh)  Since \( m \notin \text{meth}(M) \), the rule (ATr·Inh) can apply to \( TA_1 \) which implies the result.
  \item (Path·Ali)  The rule (ATr·Ali) (or (ATr·Ali_4)) can apply to \( TA_1 \) which implies the result.
  \item (Path·Ali_2)  Since \( m \neq p \) and \( m \neq q \), the rule (ATr·Ali_2) can apply to \( TA_1 \) which implies the result.
  \item (Path·Exl)  Since \( m \neq p \), the rule (ATr·Exl_1) can apply to \( TA_1 \) which implies the result.
\end{itemize}

We prove that if a trait is well-typed, then \text{meth} refers to the set of methods
where \text{altlook} does not fail.

Lemma 4 (\text{meth} Soundness)
If \( TA \text{ OK requires } \overline{S}, \) then \( m \in \text{meth}(TA) \) if and only if \( \text{altlook}(m,TA) \neq \text{fail} \).

Proof  By induction on the derivation of \( \text{altlook}(m,TA) \). Here are the most relevant cases
\begin{itemize}
  \item (ATr·Found)  Then, \( TA = T \). Then, \( \text{altlook}(m,T) \neq \text{fail} \) and, from rule (Mth·Tr), we have \( m \in \text{meth}(T) \).
  \item (ATr·Inh)  Then, \( TA = T \), and \( TT(T) = \text{trait } T \text{ imports } TA \{\overline{M};\overline{S}\} \). Then,
\end{itemize}
Lemma 6 (Conflict Resolution in Trait Alterations)

If $\text{TA OK}$ requires $\overline{\text{S}}$, then $\text{altlook}(\cdot, \text{TA})$ is a function.

Proof. By induction on the derivation of $\text{altlook}$. Here are the most relevant cases:

- (ATr-Inh) If $\text{tlook}(\text{m}, \overline{\text{TA}}) = \text{fail}$, then the property obviously holds. Else, by induction hypothesis, for all $\text{TA}_1 \in \overline{\text{TA}}$, $\text{altlook}(\cdot, \text{TA}_1)$ is a function.
- If $\text{m} \not\in \cap \overline{\text{TA}}$, then there is an unique $\text{TA}_1 \in \overline{\text{TA}}$ where $\text{altlook}(\text{m}, \text{TA}_1) \neq \text{fail}$.
- If $\text{m} \in \cap \overline{\text{TA}}$, then, since $\text{TA}$ is well-typed, the rule (Tr-Ok) enforces that $\text{m} \in \cap \overline{\text{TA}}$. Then, for all $\text{TA}_1 \in \overline{\text{TA}}$, we have $\text{m} \in \text{meth}(\text{TA}_1)$, and $\text{m} \in \text{meth}(\text{TA}_1)$.

Moreover, we know that $\text{n} \in \text{meth}(\text{TA}_1)$, by Lemma 1, which means that there is at least one $\text{altlook}(\text{n}, \text{TA}_1) = \text{I} \text{n} (\overline{\text{T}} \text{x})\{\ldots\}$ which is derivable, by Lemma 4. Thus, $\text{altlook}(\text{m}, \text{TA}_1) = \text{I} \text{m} (\overline{\text{T}} \text{x})\{\ldots\}$ is derivable for all $\text{TA}_1$, such that $\text{m} \in \text{meth}(\text{TA}_1)$, by Lemma 3. To conclude, we know that $\text{altlook}(\cdot, \text{TA}_1)$ is a function which ensures they are all equal.

We prove that $\text{sig}$ returns the set of the signatures of all method (both implemented and required) of a typed trait.

Lemma 5 (Conflict Resolution in Trait Alterations)

If $\text{TA OK}$ requires $\overline{\text{S}}$, then $\text{altlook}(\cdot, \text{TA})$ is a function.

Proof. By induction on the derivation of $\text{altlook}$. Here are the most relevant cases:

- (ATr-Inh) If $\text{tlook}(\text{m}, \overline{\text{TA}}) = \text{fail}$, then the property obviously holds. Else, by induction hypothesis, for all $\text{TA}_1 \in \overline{\text{TA}}$, $\text{altlook}(\cdot, \text{TA}_1)$ is a function.
- If $\text{m} \not\in \cap \overline{\text{TA}}$, then there is an unique $\text{TA}_1 \in \overline{\text{TA}}$ where $\text{altlook}(\text{m}, \text{TA}_1) \neq \text{fail}$.
- If $\text{m} \in \cap \overline{\text{TA}}$, then, since $\text{TA}$ is well-typed, the rule (Tr-Ok) enforces that $\text{m} \in \cap \overline{\text{TA}}$. Then, for all $\text{TA}_1 \in \overline{\text{TA}}$, we have $\text{m} \in \text{meth}(\text{TA}_1)$, and $\text{m} \in \text{meth}(\text{TA}_1)$.

Moreover, we know that $\text{n} \in \text{meth}(\text{TA}_1)$, by Lemma 1, which means that there is at least one $\text{altlook}(\text{n}, \text{TA}_1) = \text{I} \text{n} (\overline{\text{T}} \text{x})\{\ldots\}$ which is derivable, by Lemma 4. Thus, $\text{altlook}(\text{m}, \text{TA}_1) = \text{I} \text{m} (\overline{\text{T}} \text{x})\{\ldots\}$ is derivable for all $\text{TA}_1$, such that $\text{m} \in \text{meth}(\text{TA}_1)$, by Lemma 3. To conclude, we know that $\text{altlook}(\cdot, \text{TA}_1)$ is a function which ensures they are all equal.

We prove that $\text{sig}$ returns the set of the signatures of all method (both implemented and required) of a typed trait.

Lemma 6 (sig Soundness)

If $\text{TA OK}$ requires $\overline{\text{S}}$, then $\text{mtype}(\text{m}, \text{TA}) = \overline{\text{I}} \rightarrow \text{I}$ if and only if $\text{I} \text{m} (\overline{\text{T}} \text{x}) \in \text{sig}(\text{TA})$.

Proof. We prove, first, that for any $\text{m} \in \text{meth}(\text{TA})$, we have $\text{mtype}(\text{m}, \text{TA}) = \overline{\text{I}} \rightarrow \text{I}$ for some $\overline{\text{I}}, \text{I}$ and for the same $\overline{\text{I}}, \text{I}$, we have $\text{I} \text{m} (\overline{\text{T}} \text{x}) \in \text{sig}(\text{TA})$. This reduces to the fact that $\text{altlook}(\text{m}, \text{TA}) = \text{I} \text{m} (\overline{\text{T}} \text{x})\{\ldots\}$ implies $\text{I} \text{m} (\overline{\text{T}} \text{x}) \in \text{sig}(\text{TA})$, by Lemma 4, and thanks to the rule (MTyp-Impl). We prove the latter result by straightforward induction on the derivation of $\text{altlook}(\text{m}, \text{TA}) = \text{I} \text{m} (\overline{\text{T}} \text{x})\{\ldots\}$. To conclude, we now need to establish the fact that if $\text{m} \not\in \text{meth}(\text{TA})$, then we have $\text{mtype}(\text{m}, \text{TA}) = \overline{\text{I}} \rightarrow \text{I}$ if and only if $\text{I} \text{m} (\overline{\text{T}} \text{x}) \in \text{sig}(\text{TA})$. This fact is equivalent to if $\text{m} \not\in \text{meth}(\text{TA})$, then $\text{msig}(\text{m}, \text{TA}) = \text{I} \text{m} (\overline{\text{T}} \text{x})$, $\text{msig}(\text{m}, \text{TA}) = \text{fail}$, we can prove both directions separately (and that either of them holds). We prove that if $\text{m} \not\in \text{meth}(\text{TA})$ and $\text{msig}(\text{m}, \text{TA}) = \text{I} \text{m} (\overline{\text{T}} \text{x})$, then $\text{I} \text{m} (\overline{\text{T}} \text{x}) \in \text{sig}(\text{TA})$, and that if $\text{m} \not\in \text{meth}(\text{TA})$ and $\text{msig}(\text{m}, \text{TA}) = \text{fail}$, we can prove both directions separately (and that either of them holds).
then \( I \ m(\overline{T} \ x) \notin \text{sig}(\overline{T}A) \). Both are proved by straightforward induction on the derivation of \( \text{msig}(m, TA) = S_1 \). We will emphasize one case of each, both being representative of the proofs.

- (MSig-Ali3) Since \( \text{altlook}(m, TA) = I \ m(\overline{T} x) \{\ldots\} \), then we know, thanks to the former part of the proof, that \( I \ m(\overline{T} x) \in \text{sig}(\overline{T}A) \). Then, by rule (Sig-Alias), we derive that \( I \ m(\overline{T} x) \in \text{sig}(\overline{T}A \text{ with } \{m \circ n\}) \). Hence the result.
- (MSig-Ali4) Thanks to typechecking of \( TA \) with \( \{m \circ n\} \), we know that \( m \in \text{meth}(\overline{T}A) \). By Lemma 4, we thus know that \( \text{altlook}(m, TA) \neq \text{fail} \). This case can’t raise. \( \square \)

We prove that when the program typechecks, \( \text{msig} \) is a function. It is necessary to ensure soundness of typing.

**Lemma 7 (Conflict Resolution in Abstract Signatures)**

If \( TA \ OK \) requires \( S \), then \( \text{msig}(\cdot, TA) \) is a function (modulo renaming of the formal parameters).

**Proof** By induction on a derivation of \( \text{msig} \). All cases are obviously disjoint, except for \((\text{MSig-Ex1}) / (\text{MSig-Ex2})\), and \((\text{MSig-Ali3}) / (\text{MSig-Ali4})\) which apply to the same terms. Remember however that by Lemma 5, \( \text{altlook}(\cdot, TA) \) is a function, thus ensuring that those two pairs of rules are indeed disjoint. Knowing this and the fact that \( \text{altlook}(\cdot, TA) \) is a function, every rule of \( \text{msig} \) is straightforward, except from (MSig-Inh) treated below.

- (MSig-Inh) Then, \( T \) has been typechecked through rule (Tr-Ok), and \( TT(T) = \text{trait } T \text{ imports } T \overline{A} \{\overline{R}, \overline{S}\} \). We have, incidentally, that all \( TA \in \overline{T}A \) do typecheck also. Thus, by Lemma 5, \( \text{altlook}(\cdot, TA) \) is a function for all \( TA \in \overline{T}A \) and, by induction hypothesis, \( \text{msig}(\cdot, TA) \) is a function for all \( TA \in \overline{T}A \). We also have that \( m \notin \text{meth}(\overline{T}A) \), by Lemma 4, and this means that \( \text{altlook}(m, TA) = \text{fail} \) for all \( TA \in \overline{T}A \) (and it can be nothing else since \( \text{altlook}(\cdot, TA) \) is a function). We can then conclude that \( \text{mtype}(m, TA) \) is inferred through the (MTyp-Virt) rule for each \( TA_i \in \overline{T}A \). Now let’s assume that there are \( TA_1, TA_2 \in \overline{T}A \) such that \( \text{msig}(m, TA_1) \neq \text{fail} \) and \( \text{msig}(m, TA_2) \neq \text{fail} \). Then, \( \text{msig}(m, TA_1) = I \ m(\overline{T} x) \) and \( \text{msig}(m, TA_2) = J \ m(\overline{J} y) \), and we can then deduce that \( \text{mtype}(m, TA_1) = T \rightarrow I \) and \( \text{mtype}(m, TA_2) = J \rightarrow J \) are derivable. The judgment \( (\text{sig}(\overline{T}A) \cup S) \text{ OK in } T \) in (Tr-Ok), ensures then that \( \overline{T} = \overline{J} \) and \( I = J \), by Lemma 6. Hence the result. \( \square \)

The system is kept non-deterministic 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 1 (Conflict Resolution)**

If for all \( C_i \in \text{CL} \), we have \( C_i \ OK \) and for all \( T_i \in \text{TT} \), we have \( T_i \ OK \) requires \( S \), then both \( \text{mbody}(\cdot, \cdot) \) and \( \text{mtype}(\cdot, \cdot) \) are functions.

**Proof** We prove that \( \text{mbody}(\cdot, \cdot) \) is a function by induction on the derivation of \( \text{mbody}(m, C_i) \), the proof for \( \text{mtype}(\cdot, \cdot, \cdot) \) being similar (note however that...
there are two extra cases to deal with for mtype, handled directly by Lemmas 5 and 7).

- (MBdy-Cla) Direct.
- (MBdy-SCla) Straightforward by induction hypothesis.
- (MBdy-Tr) For all $\mathcal{T}_{A_1} \in \mathcal{T}_{\mathcal{A}}$, allook($\cdot$, $\mathcal{T}_{A_1}$) is a function by Lemma 5.
  - If $m \not\in \cap \mathcal{T}_{\mathcal{A}}$, then there is an unique $\mathcal{T}_{A_1} \in \mathcal{T}_{\mathcal{A}}$ where allook($m$, $\mathcal{T}_{A_1}$) $\neq$ fail.
  - If $m \in \cap \mathcal{T}_{\mathcal{A}}$, then, since $C_1$ is well-typed, the rule (Cla-Ok) enforces that $m \in \circ \mathcal{T}_{\mathcal{A}}$. Then, for all $\mathcal{T}_{A_1} \in \mathcal{T}_{\mathcal{A}}$, we have $m \in \operatorname{meth}(\mathcal{T}_{A_1}) \Rightarrow m$ in $\mathcal{T}_{A_1} \not\subseteq n$ in $\mathcal{T}_{A_1}$. Moreover, we know that $n \in \operatorname{meth}(\mathcal{T}_{A_1})$, by Lemma 1, which means that there is at least one allook($n$, $\mathcal{T}_{A_1}$) = $I$ n ($I$ x){...} which is derivable, by Lemma 4. Thus, allook($m$, $\mathcal{T}_{A_1}$) = $I$ m ($I$ x){...} is derivable, for all $\mathcal{T}_{A_1}$ such that $m \in \operatorname{meth}(\mathcal{T}_{A_1})$, by Lemma 3. To conclude, we know that allook($\cdot$, $\mathcal{T}_{A_1}$) is a function. Which ensures they are all equal. □

In the following, we suppose that Theorem 1 holds, and we can address mbody and mtype as mathematical functions. We prove that $msig$ has the same semantics than the requires set of trait checking. Both are expected to give the signatures of the methods which are required in a trait alteration.

**Lemma 8 (Require Soundness)**

If $\mathcal{T}_{A} \mathit{Ok}$ requires $\mathcal{S}$, then $msig(m, \mathcal{T}_{A}) = \mathcal{S}$ if and only if $\mathcal{S} \in \mathcal{S}$.

**Proof** Straightforward induction on the derivation of $msig(m, \mathcal{T}_{A}) = \mathcal{S}$. Here are the most relevant cases:

- (MSig-Al1) In both rules (Alias-Ok1) and (Alias-Ok2), it is easy to observe that the new name of the method is not in the requires list (the role of (Alias-Ok2) is actually to remove it from the requires list if it appears in the previous step). Hence the result.
- (MSig-Al13) In both rules (Alias-Ok1) and (Alias-Ok2), $n \in \operatorname{meth}(\mathcal{T}_{A_1})$ is a precondition. By Lemma 4, it contradicts the precondition of rule (MSig-Al13). This case never occurs when a trait alteration typechecks. □

We prove that trait alterations do not alter the type interface of a trait.

**Lemma 9 (head Soundness)**

If $\mathcal{T}_{A} \mathit{Ok}$ requires $\mathcal{S}$ and mtype($m$, head($\mathcal{T}_{A}$)) = $I$ $\rightarrow$ $I$, then mtype($m$, $\mathcal{T}_{A}$) = $I$ $\rightarrow$ $I$.

**Proof** By induction on $\mathcal{T}_{A}$. Let us prove the most significant cases

- (TA = $\mathcal{T}_{A_1}$ minus {p}), with $p \neq m$. Then, mtype($m$, $\mathcal{T}_{A}$) is inferred from either allook($m$, $\mathcal{T}_{A}$) or $msig(m, \mathcal{T}_{A})$ (thanks to (MTyp-Impl) or (MTyp-Virt), respectively). In both cases the result is straightforward.
- (TA = $\mathcal{T}_{A_1}$ minus {m}). Since $\mathcal{T}_{A}$ typechecks, then allook($m$, $\mathcal{T}_{A_1}$) $\neq$ fail, by Lemma 4. We also have, by definition of allook, that allook($m$, $\mathcal{T}_{A_1}$) = fail. By definition of mtype, we have that mtype($m$, $\mathcal{T}_{A}$) is inferred from $msig(m, \mathcal{T}_{A})$ (rule (MTyp-Virt)), and that mtype($m$, $\mathcal{T}_{A_1}$) is inferred from allook($m$, $\mathcal{T}_{A_1}$) (rule (MTyp-Impl)). By rule (MSig-Ex1), we have mtype($m$, $\mathcal{T}_{A_1}$) = $I$ $\rightarrow$ $I$.
implies $\text{mtype}(m, \text{TA}) = \Gamma \rightarrow I$. By induction hypothesis we have the result.

- (TA = TA$_1$ with \{m \in A\}). Since TA typechecks, we can derive that TA$_1$ typechecks and altlookup(m,TA$_1$) = fail. Then, we know that if mtype(m,TA$_1$) = \( I \rightarrow J \), then \text{msg}(m,TA$_1$) = I m(\( \Gamma \rightarrow I \)). By Lemma 8, it follows that I m(\( \Gamma \rightarrow I \)) \in \mathcal{F}$ (where TA$_1$ OK requires \( \mathcal{F} \)). We can deduce that TA is typechecked through the (Alias-Ok$_2$) rule. It is then obvious that mtype(m,TA$_2$) = \( I \rightarrow I \) implies mtype(m,TA) = \( I \rightarrow I \) (it is enforced directly by the (Alias-Ok$_2$) rule). By induction hypothesis, we have the result. □

From now on the lemma and theorem sequence is the same as in FTJ and FTJ.

**Lemma 10 (mtype Soundness)**

If mtype(m, J) = \( \Gamma \rightarrow L \), then mtype(m, I) = \( \Gamma \rightarrow L \), for all I \( <: J \).

**Proof** Straightforward induction on the derivation of I \( <: J \), using Lemma 9 for the cases (Sub-Tr) and (Sub-Cla-Tr). We show the most difficult cases

- (Sub-Cla) Let’s assume that mtype(m, D) = \( \Gamma \rightarrow L \). We want to prove that mtype(m, C) = \( \Gamma \rightarrow L \). It can be achieved by several rules.
  - (MTyp-Self) Then, the result is obtained thanks to the rule (Mth-Ok-Cla); in particular, thanks to the override clause in it.
  - (MTyp-Tr) Let us suppose that the type of m is obtained from the trait TA$_i$. We then know that sig(TA$_i$) OK IN C. In particular, for any \( I m(\Gamma \rightarrow I) \in \text{sig(TA$_i$)} \) holds. We can rewrite it thanks to Lemma 6 to if mtype(m, D) = \( \Gamma \rightarrow I \), then mtype(m, TA$_i$) = \( \Gamma \rightarrow I \).
  - (MTyp-Super) This case is direct. □

**Lemma 11 (Substitution lemma)**

If \( \Gamma, x : \mathcal{E} \vdash e \in J \) and \( \Gamma \vdash \exists \in I \), where \( \exists \rightarrow J \), then \( \Gamma \vdash \exists[e/x] \in I \) for some I \( <: J \).

**Proof** By induction on the derivation of \( \Gamma, x : \mathcal{E} \vdash e \in J \). We will show only two cases the other ones are straightforward

- (Typ-Call). By induction hypothesis, we have \( \Gamma \vdash \exists[e/x] \in J_0 \) and \( J_0 <: I_0 \), and \( \Gamma \vdash \exists[e/x] \in I \), and \( I \rightarrow J \). Then, by Lemma 10, we have that mtype(m, J$_0$) = mtype(m, I$_0$) = \( J \rightarrow I \). By transitivity we also have \( I \rightarrow J \). Then, we conclude that \( \Gamma \vdash e_0, m(\exists) \in I \) (and obviously I \( <: I \)).
- (Typ-Field). By induction hypothesis, we have \( \Gamma \vdash \exists[e/x] \in I_0 \) for some I$_0$ \( <: C_0 \). An easy induction on the derivation of I$_0$ \( <: C_0 \) shows that I$_0$ = D$_0$ for some D$_0$. Then, it is easy to show that D$_0$ \( <: C_0 \) implies fields(D$_0$) \( \subseteq \) fields(C$_0$). Hence the result. □

**Lemma 12 (Weakening)**

If \( \Gamma \vdash e \in C \), then \( \Gamma, x : \mathcal{D} \vdash e \in C \).

**Proof** Straightforward induction. □

**Lemma 13 (Method Body Type)**

If mtype(m, C) = \( J \rightarrow J \) and mbody(m, C) = (\( \mathcal{X}, e \)), then for some I$_0$ with C \( <: I_0 \),
there exists $I <: J$ such that $\exists J, \text{this} : I_0 \vdash e \in I$.

**Proof** We prove the following statement by induction on the derivation of $\text{mbody}(m, C) = (\overline{x}, e)$: if $\text{mbody}(m, C) = (\overline{x}, e)$, then $J \vdash (J \overline{x})\{\text{return } e;\} \text{ OK IN } I_0$ for some $I_0$ with $C <: I_0$ and some $J \rightarrow J$. Then, by Lemma 10, $J \rightarrow J = \text{mtype}(m, C)$ holds.

- (MBdy-Cla) Immediate from (Cla-Ok) rule.
- (MBdy-Tr) Straightforward induction on the derivation of $\text{altlook}(m, TA) = J \vdash (J \overline{x})\{\text{return } e;\}$.
- (MBdy-SCla) Induction case. □

We are ready to prove the main theorems.

**Theorem 2** (Subject Reduction)
If $\Gamma \vdash e \in J$ and $e \rightarrow e'$, then $\Gamma \vdash e' \in I$, for some $I <: J$.

**Proof** We prove it by a straightforward induction on the derivation of $e \rightarrow e'$. The base case (reduction of the head redex) is done by a straightforward case analysis on the reduction rule used. This proof has no difficult content, however if a reader is interested, a comprehensive account of the details can be found in the original FJ paper [9]. We provided all the lemmas used in the proof, so it works also for ifFTJ (the key lemma being Lemma 13). □

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

1. If $e$ includes $\text{new } C(\overline{f}).f$ as a subexpression, then $\text{fields}(C) = \overline{f}$ and $f \in \overline{f}$.
2. If $e$ includes $\text{new } C(\overline{m}).m(\overline{f})$ as a subexpression, then $\text{mbody}(m, C) = (\overline{x}, e_0)$ and $\#(\overline{x}) = \#(\overline{d})$.

**Proof** The proof is straightforward: subexpression are well-typed, thus we can assume that the subexpression appears at the head of $e$. Then, the result is deduced directly from the typing rules. Theorem 1 is essential for this proof. □

**Theorem 4** (Reduction preserves safety)
If $e$ is safe in $\Gamma$, and $e \rightarrow e'$, then $e'$ is safe in $\Gamma$.

**Proof** This proof is just similar to the Subject Reduction proof. □

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

**Proof** The result is straightforward from the definition of safety. Indeed the only rule that can be applied to derive the type of $(C)\text{new } C_0(\overline{f})$ is $(\text{Typ-UCast})$. The result follows directly. □