Implementing type-safe software product lines using parametric traits

Implementing Type-Safe Software Product Lines using Records and Traits☆

Lorenzo Bettini[a], Ferruccio Damiani[*,a], Ina Schaefer[b]

[a]*Dipartimento di Informatica, Università di Torino*
[b]*Institute for Software Systems Engineering, Technische Universität Braunschweig*

Abstract

A *software product line (SPL)* is a set of related software systems with well-defined commonality and variability that are developed by reusing common artifacts. In this paper, we present a novel technique for implementing SPLs by exploiting mechanisms for fine-grained reuse which are orthogonal to class-based inheritance. We formalize our proposal by means of FEATHERWEIGHT RECORD-TRAIT JAVA (FRTJ), a minimal core calculus where units of product functionality are modeled by *traits*, a construct for fine-grained behavior reuse, and by *records*, a construct that complements traits to model the variability of state. Records and traits are assembled in classes that are used to build products. The composition of product functionality is realized by explicit operators of the calculus, allowing code manipulations for modeling product variability. The FRTJ type system ensures that the products in the SPL are type-safe by type-checking the records, traits and classes shared by different products only once. Moreover, type-safety of an extension of a (type-safe) SPL can be guaranteed by checking only the newly added parts.

Key words: Featherweight Java, Feature Model, Software Product Line, Trait, Type System

1. Introduction

A *software product line (SPL)* is a set of software systems with well-defined commonality and variability [14, 30]. SPL engineering aims at developing these systems by managed reuse. Products of a SPL are commonly described in terms of *features* [18], where a feature is a unit of product functionality. Feature-based product variability has to be captured in the product line artifacts that are reused to realize the single products. On the implementation level, reuse mechanisms for product implementations have to be flexible enough to express the desired product variability. Additionally, they should provide static guarantees that the resulting products are type-safe. In order to be of effective use, the type-checking has to facilitate the analysis of newly added parts, if the product line evolves, without re-checking unmodified, already existing parts.

Today, most product implementations of SPLs are carried out within the object-oriented paradigm. Although class-based inheritance in object-oriented languages provides means for code reuse with static guarantees, the rigid structure of class-based inheritance puts limitations on the effective modeling of product variability and on the reuse of code (in particular, code reuse can be exploited only from within a class hierarchy) [27, 16]. *Feature-oriented programming (FOP)* [6] allows to flexibly implement product lines within the object-oriented paradigm by complementing class-based inheritance by class refinement. In FOP, a product implementation for a particular feature configuration is obtained by composing *feature modules* for the respective features. A feature module contains class definitions and class *refinements*. A class refinement can modify an existing class by adding new fields/methods, by wrapping code around existing methods or by changing the superclass. *Delta-oriented programming (DOP)* [32] extends FOP by the possibility to remove code from an existing product. In DOP, a product implementation is obtained by applying

☆This work has been partially supported by MIUR DISCO - Distribution, Interaction, Specification, Composition for Object Systems. The authors of this work have been partially supported by the Ateneo Italo-Tedesco / Deutsch-Italienisches Hochschulzentrum (Vigoni project "Language constructs and type systems for object oriented program components"), the Deutsche Forschungsgemeinschaft (DFG) and the European FET-IP HATS.

*Corresponding Author.

modifications specified in *delta modules* to existing products. Both FOP [4, 15] and DOP [33] are equipped with type systems that allow establishing the type-safety of the implemented products.

In this paper, we explore another approach to structure the implementation of SPLs in which flexible code reuse is combined with static guarantees. Instead of implementing products by specifying code modifications, products are realized by exploiting *record and trait composition*. The term *trait* has been used by Ungar et al. [41], in the context of the dynamically-typed prototype-based language SELF, to refer to a parent object to which an object may delegate some of its behavior. Subsequently, Schärli et al. [35, 16] introduced traits in the context of the dynamically-typed class-based language SQUEAK/SMALLTALK, as means for fine-grained code reuse to overcome the limitations of class-based inheritance. A *trait* is a set of methods, completely independent from any class hierarchy. *Records* [11] have been introduced to represent the counterpart of traits with respect to the state. In the original proposals of traits in SQUEAK/SMALLTALK [35, 16] (and in most of the subsequent formulation of traits within a JAVA-like nominal type system [37, 28, 31, 24]) trait composition and class-based inheritance live together. However, class-based inheritance introduces an obstacle for flexibly implementing product lines since it limits the possibilities of reusing code. Therefore, in our approach, class-based inheritance is ruled out. Classes are assembled only by composition of code artifacts (traits, interfaces and records) that are suitable for reuse in different product implementations.

We formalize our approach in FEATHERWEIGHT RECORD-TRAIT JAVA (FRTJ), a minimal core calculus (in the spirit of FJ [17]) for interfaces, records, traits and classes. In FRTJ, the concepts of type, state, behavior are separated into different and orthogonal linguistic concepts: interfaces, records and traits, respectively. FRTJ is an extension of the trait-based calculus presented in [12] with an enhanced version of the *record* construct introduced in [11], in order to model variability in the state part of products explicitly. The type system of FRTJ provides static guarantees on safe and consistent class assembly from records, traits and interfaces. FRTJ programs may look more verbose than standard class-based programs; however, the degree of reuse provided by records and traits is higher than the reuse potential of standard static class-based hierarchies. The intent of this paper is not to present the calculus FRTJ in itself, but to formalize the implementation of SPLs using linguistic constructs for fine-grained code reuse. SPLs in FRTJ are implemented in three layers. First, the FRTJ language is used for programming records, traits and interfaces which are assembled into classes. Second, a product is specified by the classes it uses. Third, a SPL is described by its products and its *artifact base*, consisting of the records, traits, interfaces and classes used to build the products of the SPL. The type system of FRTJ ensures that a SPL is type-safe by type-checking the artifacts in the artifact base only once. Type-safety of an extension of a (type-safe) product line can be guaranteed by analyzing only the newly added parts.

The main differences of the trait-oriented approach for implementing SPL presented in this paper with respect to the approaches based on FOP and DOP relying on class-based inheritance, such as [20, 36, 6, 4, 15, 32], are the following:

- The modeling of SPL variability and the associated code reuse are only achieved by trait and record composition operations (for creating new traits/records by removing, aliasing, and renaming members from already defined traits/records), without introducing feature/delta modules specifying class refinements/modifications.
- The classes, interfaces, records and traits of all the products coexist in the artifact base. Generation of a single product just amounts to selecting a subset of these artifacts. Therefore, a class/interface/trait/record name refers to the same definition entity in all the products.

A preliminary version of the results presented in this paper has been presented in [8].

Organization of the Paper. In Section 2, we illustrate traits and records. In Section 3, we show how to use them to implement the products of a software product line with an example. In Section 4, we present the FRTJ calculus and state its type soundness. In Section 5, we use FRTJ to formalize software product lines and their type-checking. Related work is discussed in Section 6. We conclude by outlining some directions for future work. The appendices contains proofs omitted from the main text.

2. Introducing Records and Traits

Traits have been introduced in the dynamically-typed class-based language SQUEAK/SMALLTALK to play the role of *units for behavior fine-grained reuse*: the common behavior (that is, the common methods) of a set of classes can

```
interface IAccount { void update(int x); }
record RAccount is { int balance; /* provided field */ }
trait TAccount is {
  int balance; /* required field */
  void update(int x) { balance = balance + x; } /* provided method */ }
---
class Account implements IAccount by RAccount and TAccount
```

Listing 1: Artifacts for the ACCOUNT product

be factored into a trait [35, 16]. Various formulations of traits in a JAVA-like setting can be found in the literature (see, e.g., [37, 28, 11, 31, 12, 24]). The programming language FORTRESS [1] (which does not contain class-based inheritance) incorporates a trait construct, while the "trait" construct of SCALA [29] is indeed a form of mixin.

In this paper, we use the concept of trait as described in [12]. A trait can consist of *provided methods*, that implement behavior, of *required methods*, that parametrize the behavior itself, and of *required fields*, that can be directly accessed in the body of the provided methods. Traits are the building blocks to compose classes or other, more complex, traits. A suite of trait composition operations allows the programmer to build classes and composite traits. A distinguished characteristic of traits is that the composite unit (class or trait) has complete control over conflicts that may arise during composition and must solve them explicitly. Traits do not specify any state, therefore a class composed by using traits has to provide the required fields (by means of records). The trait composition operations considered in this paper are as follows:

- A basic trait defines a set of methods and declares the required fields and the required methods.
- The *symmetric sum* operation, +, merges two traits to form a new trait. It requires that the summed traits must be disjoint (that is, they must not provide identically named methods).
- The operation `exclude` forms a new trait by removing a method from an existing trait.
- The operation `aliasAs` forms a new trait by giving a new name to an existing method.
- The operation `renameTo` creates a new trait by renaming all the occurrences of a required field name or of a required/provided method name from an existing trait.

Note that the actual names of the methods defined in a trait (and also the names of the required methods and fields) are irrelevant, since they can be changed by the `renameTo` operation.

A record is a set of fields, completely independent from any class hierarchy. Records have been recently proposed in [11] as the counterpart of traits with respect to state to play the role of *units for state fine-grained reuse*. The common state (i.e., the common fields) of a set of classes can be factored into a record. Records are building blocks to compose classes or other, more complex, records by means of operations analogous to the ones described above for traits. The record construct considered in this paper enhances the original one [11] by providing a richer set of composition operations.

In the following, we illustrate the trait and record constructs by an example implementation of bank accounts (cf. [15]). We use a JAVA-like notation and a more general syntax (including. e.g., the types `void` and `boolean`, the assignment operator, etc.) than the one of the FRTJ calculus presented in Section 4. We omit the class constructors in the examples. All constructors are assumed to be of the form

C(I1 f1,..., In fn) {**this**.f1 = f1;...;**this**.fn=fn;}

where `f1,...,fn` are all the fields of the class `C`. We consider the implementation of a class `Account` providing the basic functionality to update the balance of an account with the interface:

interface IAccount { **void** update(); }

In a language with traits and records, the fields and the methods of the class can be defined independently from the class itself, as illustrated by the code at the top of Listing 1. The class `Account` is composed as shown at the bottom of Listing 1.

A class `SyncAccount` implementing a variant of the basic bank account that guarantees synchronized access can be developed by introducing a record `RSync` that provides a field for a lock and a trait `TSync` that provides a method

```
record RSync is { Lock lock; /* provided field */ }
trait TSync is {
  Lock lock; /* required field */
  void m(int x); /* required method */
  void sync_m(int x) { lock.lock(); m(x); lock.unlock(); } /* provided method */ }

record RSyncAccount is RSync + RAccount
trait TSyncAccount is TAccount[update renameTo unsyncUpdate]
    + TSync[m renameTo unsyncUpdate, sync_m renameTo update]

---

class SyncAccount implements IAccount by RSyncAccount and TSyncAccount
```

Listing 2: Artifacts for the SYNC_ACCOUNT product

that wraps the code for synchronization around a non-synchronized method. Based on these and the record `RAccount` and trait `TAccount` for the basic account, a record `RSyncAccount` and a trait `TSyncAccount` can be defined providing the fields and methods of the class `SyncAccount`. The corresponding code is shown in Listing 2.

The record `RSync` and the trait `TSync` are completely independent from the code for the basic account. Because of the trait and record operations to rename methods and fields, they can be reused for synchronizing any method (provided the signature is the same as in `TSync`) or several methods on the same lock (as we will see in Section 3). FRTJ extends method reusability of traits to state reusability of records, and fosters a programming style relying on small components that are easy to reuse.

Traits/records satisfy the so called *flattening principle* [28] (see also [23, 22]), that is, the semantics of a method-/field introduced in a class by a trait/record is identical to the semantics of the same method/field defined directly within the class. For instance, the semantics of the class `SyncAccount` in Listing 2 is identical to the semantics of the JAVA class:

```
class SyncAccount implements IAccount
{  int balance;
   Lock lock;
   void unsyncUpdate(int x) { balance = balance + x; }
   void update(int x) { lock.lock(); unsyncUpdate(x); lock.unlock(); } }
```

3. Implementing Software Product Lines

As a running example to demonstrate how product line variability is implemented in our trait-based approach, we use the SPL of bank accounts considered in [15]. The products of a SPL are defined by their features. A feature is a designated characteristic of a product and represents a unit of product functionality. Figure 1 shows the feature model [18] of the bank account SPL determining the different products by possible combinations of features. The mandatory Base feature represents the basic functionality of any bank account allowing to store the current balance and to update it. This functionality can be extended by the optional Sync(hronized) feature guaranteeing synchronized access to the account. The features Retirement and Investment that provide the possibility to store an additional bonus for the account are optional and mutually exclusive. The optional feature With Holder adds a reference to the holder of the account and requires the presence of either the Retirement or the Investment feature.

Products in a SPL are constructed from a common artifact base. In our approach, the artifact base for a SPL consists of records, traits, interfaces and the classes assembled thereof. Products use different classes depending on the features they provide. The record, trait, interface and class that capture the functionality of the account providing the Base feature are given in Listing 1. The product `ACCOUNT` (providing the mandatory feature Base) is specified by the declaration

```
product ACCOUNT uses Account  // 1st product
```

A product providing several features can be realized by composing and/or modifying records, traits and interfaces contained in the artifact base. Listing 2 contains the records, traits and class required to implement the Sync feature. The product `SYNC_ACCOUNT` (providing the features Base and Sync) is specified by the declaration

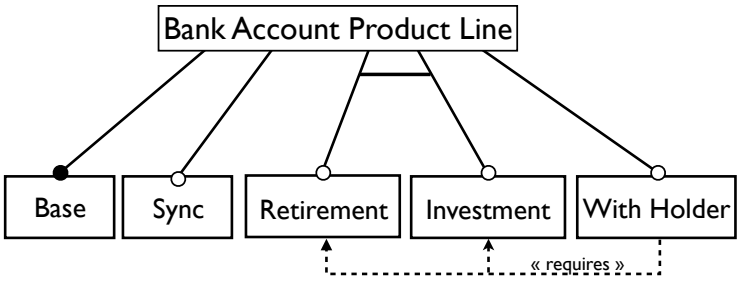

Figure 1: Feature Model for Bank Account Product Line

```
interface IBonusAccount extends IAccount {void addBonus(int b);}
record RBonus is RAccount[balance renameTo 401balance]
trait TBonus is TAccount[balance renameTo 401balance, update renameTo addBonus]

interface IRetAccount extends IBonusAccount { }
record RRetAccount is RBonus
trait TRetAccount is TBonus + TAccount[balance renameTo 401kbalance]
---
class RetAccount implements IRetAccount by RRetAccount and TRetAccount
```

Listing 3: Artifacts for the RET_ACCOUNT product

```
product SYNC_ACCOUNT uses SyncAccount // 2nd product
```

The Retirement feature is implemented by the code artifacts in Listing 3. An account with the Retirement feature contains a `401kbalance` field that is incremented by the usual `update` method and by an additional `addBonus` method. We introduce the interface `IBonusAccount` extending the `IAccount` interface to provide uniform access to all variants of a basic bank account containing the `addBonus` method. The record `RBonus` provides the `401kbalance` field by renaming the `balance` field of the record `RAccount`. Trait `TBonus` provides the `addBonus` method by renaming the `balance` field and the `update` method of trait `TAccount` such that `TBonus` provides the functionality to increment the `401kbalance` field by the `addBonus` method. In order to implement the `RetAccount` class, we use the record `RBonus` (the `balance` field of `RAccount` is not required) and the traits `TBonus` and `TAccount` (where the `balance` field is renamed to `401kbalance`). The product `RET_ACCOUNT` (providing the features Base and Retirement) is specified by the declaration

```
product RET_ACCOUNT uses RetAccount // 3rd product
```

Listing 4 contains the code artifacts to implement the Investment feature. The `InvAccount` class implements a variant of the basic bank account which has a `401kbalance` field in addition to the usual balance of the account. When

```
trait TInv is TBonus + {
    int 401balance; /* required field */
    void originalUpdate(int x); /* required method */
    void update(int x) { x = x/2; originalUpdate(x); 401balance += x; } /*provided method*/ }

interface IInvAccount extends IBonusAccount { }
record RInvAccount is RBonus + RAccount
trait TInvAccount is TInv + TAccount[update renameTo originalUpdate]
---
class InvAccount implements IInvAccount by RInvAccount and TInvAccount
```

Listing 4: Artifacts for the INV_ACCOUNT product

```
interface IClient { void payday(int x, int bonus); }
record RClient is { IBonusAccount a; /*provided field*/}
trait TClient is {
    IBonusAccount a; /* required field */
    void payday(int x, int bonus) { a.addBonus(bonus); a.update(x); } /*provided method*/ }
---
class Client implements IClient by RClient and TClient
```

Listing 5: Artifacts for the `*_ACCOUNT_WH` products

```
trait TSync2 is TSync + TSync[m renameTo m1, sync_m renameTo synch_m1]

trait TSyncBonusAccount is TSync2[m renameTo unsyncUpdate,
    sync_m renameTo update, m1 renameTo unsyncAddBonus, sync_m1 renameTo addBonus]

record RSyncRetAccount is RSync + RRetAccount
trait TSyncRetAccount is TSyncBonusAccount
    + TRetAccount[update renameTo unsyncUpdate, addBonus renameTo unsyncAddBonus]
---
class SyncRetAccount implements IRetAccount by RSyncRetAccount and TSyncRetAccount
```

Listing 6: Artifacts of the `SYNC_RET_ACCOUNT` product

the balance is updated by the `update` method, the input is split between the basic `balance` field and the `401kbalance` field. This is realized in the trait `TInv`. The `addBonus` method increments the `401kbalance` field directly. The interface `IInvAccount`, the record `RInvAccount` and the trait `TInvAccount` provide the public methods, the fields and the methods of the class `InvAccount`. The record `RInvAccount` is composed from the records `RBonus` and `RAccount`. The trait `TInvAccount` is built by composing the trait `TInv` and the trait `TAccount` where the `update` method is renamed to `originalUpdate` to work with the `TInv` trait. The product `INV_ACCOUNT` (providing the features Base and Investment) is specified by the declaration

```
product INV_ACCOUNT uses InvAccount // 4th product
```

The With Holder feature is implemented by adding a class `Client`, representing the owner of an account in a field `a` of type `IBonusAccount`. The owner can access his account via the methods `update` and `addBonus` of the `IBonusAccount` interface. The `payday` method in the `TClient` trait increments both the `balance` and `401kbalance` fields by the input amount. The corresponding artifacts are given in Listing 5. The WithHolder feature requires the presence of a feature that implements the `IBonusAccount` interface, i.e., either Retirement or Investment. The corresponding products `INV_ACCOUNT_WH` and `RET_ACCOUNT_WH` are specified by the declarations

```
product INV_ACCOUNT_WH uses InvAccount, Client // 5th product
product RET_ACCOUNT_WH uses RetAccount, Client // 6th product
```

The product `SYNC_RET_ACCOUNT` (providing the features Base, Sync and Retirement) implements an account where all public methods are synchronized (cf. Listing 6). First, we introduce the trait `TSync2` that synchronizes two methods on the same lock. In trait `TSync2`, trait `TSync` is duplicated, and in the second version the required method `m`

```
record RSyncInvAccount is RSync + RInvAccount
trait TSyncInvAccount is TSyncBonusAccount
    + TInvAccount[update renameTo unsyncUpdate, addBonus renameTo unsyncAddBonus]
---
class SyncInvAccount implements IInvAccount by RSyncInvAccount and TSyncInvAccount
```

Listing 7: Artifacts of the `SYNC_INV_ACCOUNT` product

ID	::=	`interface I extends` $\bar{\texttt{I}}$ { $\bar{\texttt{S}}$; }	interfaces
S	::=	`I m (`$\bar{\texttt{I}}$ $\bar{\texttt{x}}$`)`	method headers
RD	::=	`record R is RE`	records
RE	::=	{ $\bar{\texttt{F}}$; } | R | RE + RE | RE[`exclude f`] | RE[`f renameTo f`]	record expressions
F	::=	`I f`	fields
TD	::=	`trait T is TE`	traits
TE	::=	{ $\bar{\texttt{F}}$; $\bar{\texttt{S}}$; $\bar{\texttt{M}}$ } | T | TE + TE | TE[`exclude m`] | TE[`m aliasAs m`] | TE[`m renameTo m`] | TE[`f renameTo f`]	trait expressions
M	::=	S { `return e;` }	methods
e	::=	x | e.f | e.m($\bar{\texttt{e}}$) | `new C(`$\bar{\texttt{e}}$`)` | (I)e	expressions
CD	::=	`class C implements` $\bar{\texttt{I}}$ `by RE and TE`	classes

Figure 2: FRTJ: Syntax

is renamed to `m1` and the provided method `sync_m` is renamed to `sync_m1`. The trait `TSyncBonusAccount` customizes the trait `TSync2` to synchronize the `update` and `addBonus` methods.

The class `SyncRetAccount` is realized by the interface `IBonusAccount`, the record `RSyncRetAccount` composed from the records `RSync` (providing the lock) and `RRetAccount` and the trait `TSyncRetAccount`. The trait `TSyncRetAccount` is composed from the trait `TSyncBonusAccount` and the trait `TRetAccount` in which the provided methods are renamed so that they can be synchronized by `TSync2`. The product is specified by the declaration

product SYNC_RET_ACCOUNT **uses** SyncRetAccount *// 7th product*

The product `SYNC_INV_ACCOUNT` (providing the features Base, Sync and Investment) is implemented in a similar way by the code artifacts in Listing 7. The product is specified by the declaration

product SYNC_INV_ACCOUNT **uses** SyncInvAccount *// 8th product*

The last two products of the SPL, obtained by adding the Sync feature to the 5th and 6th product, respectively, are specified by the declarations

// 9th and 10th products
product SYNC_INV_ACCOUNT_WH **uses** SyncInvAccount, Client
product SYNC_RET_ACCOUNT_WH **uses** SyncRetAccount, Client

This example shows that the proposed approach can be used to flexibly model product line variability without limitations by a class hierarchy. The composition operators on records and traits support the fine-grained reuse of artifacts, e.g., to express different features accessing the same fields, features removing fields that are no longer required, or different features redefining the same methods.

4. The FRTJ Calculus

In this section, we describe the syntax, type system and operational semantics of the FRTJ calculus, a minimal core calculus (in the spirit of FJ [17]) for interfaces, records, traits and classes that we use for implementing SPLs.

4.1. Syntax

The syntax of FRTJ is given in Figure 2. We use the overbar sequence notation according to [17]. For instance, the pair "$\bar{\texttt{I}}\,\bar{\texttt{x}}$" stands for "$\texttt{I}_1\,\texttt{x}_1,...,\texttt{I}_n\,\texttt{x}_n$", and "$\bar{\texttt{I}}\,\bar{\texttt{f}}$;" stands for "$\texttt{I}_1\,\texttt{f}_1;...;\texttt{I}_n\,\texttt{f}_n$;". The empty sequence is denoted by "$\bullet$" and the length of a sequence $\bar{\texttt{S}}$ is denoted by "#$\bar{\texttt{S}}$". In the FRTJ calculus, trait, record and class names are not types. The only user defined types are interface names. As pointed out in [12], using trait names as types limits the reuse potential of traits, because method exclusion and renaming operations would break the type system. Moreover, if class names are not used as types, interface and record declarations are independent from classes, and the dependencies of trait declarations on classes are restricted to object creation. By only using interfaces as types, the reuse potential of traits and records is increased to appropriately capture product line variability.

$$
\begin{array}{rcll}
\mathit{mSig}(\mathtt{I\ m}\ (\bar{\mathtt{I}}\ \bar{\mathtt{x}})) & = & \mathtt{I\ m}\ (\bar{\mathtt{I}}) & \\
\mathit{mSig}(\mathtt{S}_1;\ldots;\mathtt{S}_n;) & = & \mathit{mSig}(\mathtt{S}_1)\cdot\ldots\cdot\mathit{mSig}(\mathtt{S}_n) & \\
\mathit{mSig}(\mathtt{I}) & = & \mathit{mSig}(\bar{\mathtt{I}})\cup\mathit{mSig}(\bar{\mathtt{S}};) & \textbf{if } \mathtt{IT}(\mathtt{I}) = \mathtt{interface\ I\ extends}\ \bar{\mathtt{I}}\ \{\bar{\mathtt{S}};\} \\
\mathit{mSig}(\mathtt{I}_1,\ldots,\mathtt{I}_n) & = & \mathit{mSig}(\mathtt{I}_1)\cup\ldots\cup\mathit{mSig}(\mathtt{I}_n) & \\
\mathit{mSig}(\mathtt{S}\ \{\mathtt{return\ e};\}) & = & \mathit{mSig}(\mathtt{S}) & \\
\mathit{mSig}(\mathtt{M}_1\ldots\mathtt{M}_n) & = & \mathit{mSig}(\mathtt{M}_1)\cdot\ldots\cdot\mathit{mSig}(\mathtt{M}_n) & \\
\mathit{mSig}(\mathtt{C}) & = & \mathit{mSig}(\bar{\mathtt{I}}) & \textbf{if } \mathtt{CT}(\mathtt{C}) = \mathtt{class\ C\ implements}\ \bar{\mathtt{I}}\ \mathtt{by}\ \cdots
\end{array}
$$

Figure 3: FRTJ: Function *mSig*

In FRTJ, there are no constructor declarations. Like in FJ, the syntax of the constructor of a class is fixed with respect to the field order, types and names: in every class C, we assume the implicit constructor $\mathtt{C}(\bar{\mathtt{I}}\,\bar{\mathtt{f}})\{\mathtt{this}.\bar{\mathtt{f}} = \bar{\mathtt{f}};\}$, where $\bar{\mathtt{I}}\,\bar{\mathtt{f}}$ are the fields of C.

A class table CT is a map from class names to class declarations. Similarly, an interface table IT, a record table RT and a trait table TT map interface, record and trait names to interface, record and trait declarations, respectively. A FRTJ program is a 5-tuple $(\mathtt{IT},\mathtt{RT},\mathtt{TT},\mathtt{CT},\mathtt{e})$, where e is the expression to be executed. For the type system and the operational semantics, we assume fixed, global tables IT, RT, TT, and CT. We also assume that these tables are *well-formed*, i.e., they contain an entry for each interface/record/trait/class mentioned in the program, and that the interface subtyping and record/trait reuse graphs are acyclic.

Convention 4.1 (On Sequences of Named Elements). *A sequence of named elements (e.g., interface declarations, method headers,...) is* well-formed *if it does not contain two (or more) elements with the same name. Sequences of named elements are in general assumed to be well-formed. The fact that a sequence of named elements $\bar{\mathtt{S}}$ is well formed can be emphasized by writing "$\bar{\mathtt{S}}$ **wf**", e.g., in the premise of some typing rules. The sequence of names of the elements of $\bar{\mathtt{S}}$ is denoted by $\mathit{names}(\bar{\mathtt{S}})$, the subsequence of the elements of $\bar{\mathtt{S}}$ with the names $\bar{\mathtt{n}}$ is denoted by $\mathit{choose}(\bar{\mathtt{S}},\bar{\mathtt{n}})$, and $\mathit{exclude}(\bar{\mathtt{S}},\bar{\mathtt{n}})$ denotes the sequence obtained from $\bar{\mathtt{S}}$ by removing the elements with the names $\bar{\mathtt{n}}$. According to [17], a set-based notation for operators over sequences of named elements is used. In the union and in the intersection of sequences, denoted by $\bar{\mathtt{S}}\cup\bar{\mathtt{Z}}$ and $\bar{\mathtt{S}}\cap\bar{\mathtt{Z}}$, respectively, it is assumed that if $\mathtt{n}\in\mathit{names}(\bar{\mathtt{S}})$ and $\mathtt{n}\in\mathit{names}(\bar{\mathtt{Z}})$ then $\mathit{choose}(\bar{\mathtt{S}},\mathtt{n}) = \mathit{choose}(\bar{\mathtt{Z}},\mathtt{n})$. In the disjoint union of sequences, denoted by $\bar{\mathtt{S}}\cdot\bar{\mathtt{Z}}$, it is assumed that $\mathit{names}(\bar{\mathtt{S}})\cap\mathit{names}(\bar{\mathtt{Z}}) = \emptyset$.*

4.2. Typing

The FRTJ type system analyzes each trait definition in isolation from the classes or traits that use it. Some of the typing rules use the auxiliary function *mSig*, given in Fig. 3. The function *mSig* returns *method signatures*, ranged over by σ and ζ, i.e., method headers deprived of parameter names. For instance, the signature associated to the header $\mathtt{I\,m}(\mathtt{I}_1\,\mathtt{x}_1,\ldots,\mathtt{I}_n\,\mathtt{x}_n)$ is $\mathtt{I\,m}(\mathtt{I}_1,\ldots,\mathtt{I}_n)$.

For each method definition in a trait, the constraints on the use of this within the method body are collected by the type system. Within a basic trait expression $\{\bar{\mathtt{F}};\ \bar{\mathtt{S}};\ \bar{\mathtt{M}}\}$, each method $\mathtt{M}\in\bar{\mathtt{M}}$ is type-checked by assuming for this the structural type $\langle\ \bar{\mathtt{F}}\ \shortmid\ \overline{\sigma}\ \rangle$, where $\bar{\mathtt{F}}$ are the required fields and $\overline{\sigma} = \mathit{mSig}(\bar{\mathtt{S}})\cdot\mathit{mSig}(\bar{\mathtt{M}})$ is the (disjoint) union of the signatures of the required methods and of the provided methods of the basic trait expression, respectively. The typing judgment for method definitions is $\boxed{\mathtt{this}:\langle\ \bar{\mathtt{F}}\ \shortmid\ \overline{\sigma}\ \rangle\vdash\mathtt{M}:\mu}$ to be read: "under the assumption that this has fields $\bar{\mathtt{F}}$ and methods $\overline{\sigma}$, the method declaration M is well-typed with type μ", where $\mu = \mathtt{I\,m}\ (\bar{\mathtt{I}})\ \shortmid\ \langle\ \bar{\mathtt{F}}'\ \shortmid\ \overline{\sigma}'\ \shortmid\ \bar{\mathtt{I}}'\ \rangle$ is such that

1. $\mathtt{I\,m}\ (\bar{\mathtt{I}})$ is the signature of the method;
2. $\bar{\mathtt{F}}'$ and $\overline{\sigma}'$ specify that the body of the method (the expression e) selects the fields $\bar{\mathtt{F}}'$ ($\subseteq\bar{\mathtt{F}}$) and the methods with signatures $\overline{\sigma}'$ ($\subseteq\overline{\sigma}$) on this, respectively; and
3. $\bar{\mathtt{I}}'$ specifies that the use of this in the body of the method assumes that the class of the this object implements the interfaces $\bar{\mathtt{I}}'$.

The triples $\langle\ \bar{\mathtt{F}}\ \shortmid\ \overline{\sigma}\ \shortmid\ \bar{\mathtt{I}}\ \rangle$, ranged over by γ, represent the constraints on this inferred from the body of the method. The typing rule for classes will check that a class satisfies the constraints inferred for the bodies of the methods provided by the composing traits.

Interface declaration typing:

$$\frac{mSig(\mathtt{I})\ \mathbf{wf}}{\vdash\ \mathtt{interface\ I\ extends\ \bar{J}\ \{\ \bar{S}\ \}}}\ (\text{I-OK})$$

Record declaration and trait declaration typing:

$$\frac{\vdash \mathtt{RE} : \bar{\mathtt{F}}}{\vdash \mathtt{record\ R\ is\ RE} : \bar{\mathtt{F}}}\ (\text{R-OK}) \qquad \frac{\vdash \mathtt{TE} : \bar{\mu}}{\vdash \mathtt{trait\ T\ is\ TE} : \bar{\mu}}\ (\text{T-OK})$$

Class declaration typing:

$$\frac{\begin{array}{c}\vdash \mathtt{RE} : \bar{\mathtt{G}} \qquad \vdash \mathtt{TE} : \mu_1 ... \mu_p \qquad p \geq 0 \qquad \forall i \in 1..p,\ \mu_i = \zeta_i \ \mid\ \langle\ \bar{\mathtt{F}}^{(i)} \ \mid\ \overline{\sigma}^{(i)} \ \mid\ \bar{\mathtt{I}}^{(i)}\ \rangle \\ \cup_{i\in 1..p}\bar{\mathtt{F}}^{(i)} = \bar{\mathtt{G}} \qquad ((\cup_{i\in 1..p}\overline{\sigma}^{(i)}) \cup mSig(\bar{\mathtt{I}})) \subseteq \zeta_1 ... \zeta_p \qquad \forall \mathtt{I}' \in \cup_{i\in 1..p}\bar{\mathtt{I}}^{(i)},\ \exists \mathtt{I} \in \bar{\mathtt{I}},\quad \mathtt{I} <: \mathtt{I}'\end{array}}{\vdash\ \mathtt{class\ C\ implements\ \bar{I}\ by\ RE\ and\ TE}}\ (\text{C-OK})$$

Basic record expression and basic trait expression typing:

$$\vdash \{\,\bar{\mathtt{F}};\,\} : \bar{\mathtt{F}} \qquad (\text{T-REBAS})$$

$$\frac{\begin{array}{c} mSig(\bar{\mathtt{S}}) = \overline{\sigma} \qquad mSig(\mathtt{M}_1 ... \mathtt{M}_p) = \zeta_1 ... \zeta_p \\ p \geq 0 \qquad \forall i \in 1..p,\ \mathtt{this} : \langle\ \bar{\mathtt{F}} \ \mid\ \overline{\sigma} \cdot \zeta_1 ... \zeta_p\ \rangle \vdash \mathtt{M}_i : \mu_i,\ \mu_i = \zeta_i \ \mid\ \langle\ \bar{\mathtt{F}}^{(i)} \ \mid\ \overline{\zeta}^{(i)} \ \mid\ \bar{\mathtt{I}}^{(i)}\ \rangle \\ \bar{\mathtt{F}} = \cup_{i\in 1..p}\bar{\mathtt{F}}^{(i)} \qquad \overline{\sigma} = exclude((\cup_{i\in 1..p}\overline{\zeta}^{(i)}), names(\zeta_1 ... \zeta_p)) \qquad (\overline{\sigma} \cdot \zeta_1 ... \zeta_p) \cup mSig(\cup_{i\in 1..p}\bar{\mathtt{I}}^{(i)})\ \mathbf{wf}\end{array}}{\vdash \{\,\bar{\mathtt{F}};\ \bar{\mathtt{S}};\ \mathtt{M}_1 ... \mathtt{M}_p\,\} : \mu_1 ... \mu_p}\ (\text{T-TEBAS})$$

Method declaration typing:

$$\frac{\begin{array}{c}\mathtt{this} : \langle\ \bar{\mathtt{F}} \ \mid\ \overline{\sigma}\ \rangle, \bar{\mathtt{x}} : \bar{\mathtt{J}} \vdash \mathtt{e} : \theta \ \mid\ \langle\ \bar{\mathtt{F}}' \ \mid\ \overline{\sigma}' \ \mid\ \bar{\mathtt{I}}\ \rangle \\ \theta = \langle\ \bar{\mathtt{F}} \ \mid\ \overline{\sigma}\ \rangle\ \mathbf{implies}\ \bar{\mathtt{I}}' = \bar{\mathtt{I}} \cup \bar{\mathtt{J}} \qquad \theta \neq \langle\ \bar{\mathtt{F}} \ \mid\ \overline{\sigma}\ \rangle\ \mathbf{implies}\ (\ \theta <: \mathtt{J}\ \mathbf{and}\ \bar{\mathtt{I}}' = \bar{\mathtt{I}}\ )\end{array}}{\mathtt{this} : \langle\ \bar{\mathtt{F}} \ \mid\ \overline{\sigma}\ \rangle \vdash \mathtt{J\ m\ (\bar{J}\ \bar{x})\{return\ e;\}} : \mathtt{J\ m\ (\bar{J})} \ \mid\ \langle\ \bar{\mathtt{F}}' \ \mid\ \overline{\sigma}' \ \mid\ \bar{\mathtt{I}}'\ \rangle}\ (\text{M-OK})$$

Figure 4: FRTJ: Typing rules for interfaces, records, traits, classes, basic record expressions, basic trait expressions and methods

Nominal types, ranged over by η, are either class names or interface names. The syntax of *nominal types* is as follows: $\boxed{\eta\ ::=\ \mathtt{C}\ \mid\ \mathtt{I}}$. A nominal type is either a class name or an interface name. The syntax of *types for expressions* is as follows: $\boxed{\theta\ ::=\ \langle\ \bar{\mathtt{F}} \ \mid\ \overline{\sigma}\ \rangle\ \mid\ \eta}$. The type of the expression `this` is a pair $\langle\ \bar{\mathtt{F}} \ \mid\ \overline{\sigma}\ \rangle$, specifying that `this` has the fields $\bar{\mathtt{F}}$ and methods with signatures $\overline{\sigma}$. The type of an object creation expression $\mathtt{new\,C}(\cdots)$ is the class `C`. The type of any other expression `e` is an interface name.

The (nominal) subtyping relation is the reflexive and transitive closure of the union of the immediate implements relation and extends relation declared by the `implements` clauses in the class table `CT` and by the `extends` clauses in the interface table `IT`, respectively. We will write $\boxed{\eta_1 <: \eta_2}$ to mean that η_1 is a subtype of η_2.

An environment Γ is either a finite mapping from variable names (including `this`) to types, written "$\bar{\mathtt{x}} : \bar{\mathtt{I}}$, $\mathtt{this} : \langle\ \bar{\mathtt{F}} \ \mid\ \overline{\sigma}\ \rangle$", or the empty mapping, written "$\bullet$". We write $\vdash (\mathtt{IT}, \mathtt{RT}, \mathtt{TT}, \mathtt{CT}, \mathtt{e}) : \eta$, to be read: "the program $(\mathtt{IT}, \mathtt{RT}, \mathtt{TT}, \mathtt{CT}, \mathtt{e})$ is well-typed with type η", to mean that the interfaces in `IT`, the records in `RT`, the traits in `TT` and the classes in `CT` are well-typed, and the expression `e` is well-typed with type η.

4.2.1. Typing Rules for Interfaces, Records, Traits, Classes, Basic Record and Trait Expressions, and Methods

The typing rules for interface, record, trait and class definitions, for basic record and trait expressions, and for methods definitions are given in Fig. 4.

The typing judgement for interface definitions is $\boxed{\vdash \mathtt{interface\ I\ extends\ \bar{I}\ \{\bar{S};\}}}$, to be read: "the declaration of the interface `I` is well-typed". The associated typing rule checks that the sequence of method signatures belonging to the interface is well-formed. The typing judgement for record definitions is $\boxed{\vdash \mathtt{record\ R\ is\ RE} : \bar{\mathtt{F}}}$, to be read:

"the declaration of record R is well-typed with type $\bar{\mathtt{F}}$". The associated typing rule assigns to the record declaration the type of the record expression RE. Since the record reuse graph is acyclic, no use of R may be encountered when typing RE. The typing judgment for trait definitions is $\boxed{\vdash \mathtt{trait\ T\ is\ TE} : \bar{\mu}}$, to be read: "the declaration of trait T is well-typed with type $\bar{\mu}$". The associated typing rule assigns to the trait declaration the type of the trait expression TE. Since the trait reuse graph is acyclic no use of T may be encountered when typing TE. The typing judgment for class declarations is $\boxed{\vdash \mathtt{class\ C\ implements\ \bar{I}\ by\ RE\ and\ TE\ \ OK}}$, to be read: "the declaration of the class C is well-typed". The associated typing rule checks that the constrains in the types of the methods provided by TE are satisfied. Namely, that the class C provides the fields and the methods required by TE, that TE provides all the methods of the interfaces implemented by C, and that C is a subtype of each interface required by the methods provided by TE.

The typing judgement for record expressions is $\boxed{\vdash \mathtt{RE} : \bar{\mathtt{F}}}$, to be read: "the record expression RE is well-typed with type $\bar{\mathtt{F}}$", i.e., RE provides the fields $\bar{\mathtt{F}}$. The typing judgement for trait expressions is $\boxed{\vdash \mathtt{TE} : \bar{\mu}}$ where $\bar{\mu} = \mu_1 ... \mu_n \quad (n \geq 1)$, to be read: "the trait expression TE is well-typed with type $\bar{\mu}$", i.e., TE provides n methods with constrain-based types $\mu_1, ..., \mu_n$, respectively. The typing judgment for method definitions is described at the beginning of Section 4.2.

4.2.2. Typing Rules for Non-Basic Record Expressions and Non-Basic Trait Expressions

The typing rules for non-basic record and trait expressions are given in Fig. 5. Both, the typing rule for record names (T-RE) and the typing rule for trait names (T-TE), lookup the typing of the corresponding record and trait definition, respectively. The typing rules for non-basic record expressions are straightforward (remember that, by Convention 4.1, in the disjoint union of sequences, $\bar{\mathtt{S}} \cdot \bar{\mathtt{Z}}$, it is assumed that $names(\bar{\mathtt{S}}) \cap names(\bar{\mathtt{Z}}) = \emptyset$).

The rules for typing non-basic trait expressions are as follows. The rule for method exclusion, (T-TEEX), simply removes the type of the excluded method. The rule for symmetric sum of traits, (T-TESUM), assigns to the composed trait the type resulting from the concatenation of the types of the summed traits. Thus, it has to be checked that there are no conflicts among the methods provided by the summed traits. Moreover, in the two **wf** statements, it is ensured that there are no conflicts among the fields required by the summed traits and among the provided methods ($\zeta_1 ... \zeta_{p+q}$), the required methods ($\cup_{i \in 1..p+q} \overline{\sigma}^{(i)}$) and the method signatures of the interfaces that this must implement ($mSig(\cup_{i \in 1..p+q} \bar{\mathtt{I}}^{(i)})$). The rule for method aliasing, (T-TEAL), besides ensuring that the method to be aliased exists, it also checks that the new name does not create conflicts. The type of the alias method is added to the final type. The rule for method renaming, (T-TEREM), is similar, but since renameTo renames also the recursive occurrences of method names, it must perform the substitution also on the signatures of required/provided methods (and also on the resulting type). Rule (T-TEREF) is similar, but it acts on field names.

4.2.3. Typing Rules for Expressions

The typing rules for expressions are given in Fig. 6. The rules are syntax directed, with one rule for each term, except that there are two different rules for casts (to distinguish between *upcasts* and *downcasts*). The typing judgment is $\boxed{\Gamma \vdash \mathtt{e} : \theta \ | \ \gamma}$, to be read: "under the assumption in Γ, the expression e is well-typed with type θ and constraints γ". The meaning of the constraints γ has been illustrated at the beginning of Section 4.2.

The rule for variables, (T-VAR), is standard; no constraints on this have to be collected (even when x is this). The rule for field selection, (T-FIELD), looks up the type of this in Γ, extracts the type I of f and collects the constraint that this must have a field f of type I. When the expression e to be checked is a method body, $\Gamma(\mathtt{this})$ contains the required fields declaration of a basic trait expression $\{\bar{\mathtt{F}}; \bar{\mathtt{S}}; \bar{\mathtt{M}}\}$ which comprises the method definition. The constraints collected by means of rule (T-FIELD) are a subset of the assumptions $\bar{\mathtt{F}}$ provided by $\Gamma(\mathtt{this})$: they describe the fields that are selected on this by the checked expression. Collecting this information is crucial in order to be able to check trait expressions TE by considering only the constraints enforced by the methods provided by the trait expressions. Such constraints, due to the presence of the method exclusion operation, are more liberal than the constraints corresponding to the required fields declarations of the basic trait expressions used by TE.

In the rule for method invocation, (T-INVK), the actual parameters $(\mathtt{e}_1, ..., \mathtt{e}_n)$ are checked and the inferred constraints are collected in the conclusion of the rule. It has to be distinguished whether the receiver is this or different from this. In the former case, the signature $\mathtt{Im}(\mathtt{I}_1, ..., \mathtt{I}_n)$ of m is extracted from the sequence of signatures $\overline{\sigma}$ in the type assumed for this, and the constraint that this must have a method m with signature $\mathtt{Im}(\mathtt{I}_1, ..., \mathtt{I}_n)$ is collected in the conclusion of the rule (since $\overline{\zeta} = \mathtt{Im}(\mathtt{I}_1, ..., \mathtt{I}_n)$) . For the types of the actual parameters that are different from

Non-basic record expression typing:

$$\frac{\vdash \texttt{record R} \cdots : \bar{\texttt{F}}}{\vdash \texttt{R} : \bar{\texttt{F}}}\ (\text{T-RE}) \qquad \frac{\vdash \texttt{RE} : \bar{\texttt{F}} \qquad \vdash \texttt{RE}' : \bar{\texttt{F}}'}{\vdash \texttt{RE} + \texttt{RE}' : \bar{\texttt{F}} \cdot \bar{\texttt{F}}'}\ (\text{T-RESUM})$$

$$\frac{\vdash \texttt{RE} : \bar{\texttt{F}} \cdot \texttt{F} \cdot \bar{\texttt{G}} \qquad \mathit{names}(\texttt{F}) = \texttt{f}}{\vdash \texttt{RE}[\texttt{exclude f}] : \bar{\texttt{F}} \cdot \bar{\texttt{G}}}\ (\text{T-REEX}) \qquad \frac{\vdash \texttt{RE} : \bar{\texttt{F}} \quad \texttt{f} \in \mathit{names}(\bar{\texttt{F}}) \quad \texttt{f}' \notin \mathit{names}(\bar{\texttt{F}}) \quad \bar{\texttt{G}} = \bar{\texttt{F}}[\texttt{f}'/\texttt{f}]}{\vdash \texttt{RE}[\texttt{f renameTo f}'] : \bar{\texttt{G}}}\ (\text{T-REREF})$$

Non-basic trait expression typing:

$$\frac{\vdash \texttt{trait T} \cdots : \bar{\mu}}{\vdash \texttt{T} : \bar{\mu}}\ (\text{T-TE}) \qquad \frac{\vdash \texttt{TE} : \bar{\mu} \cdot \mu \cdot \bar{\mu}' \qquad \mathit{names}(\mu) = \texttt{m}}{\vdash \texttt{TE}[\texttt{exclude m}] : \bar{\mu} \cdot \bar{\mu}'}\ (\text{T-TEEX})$$

$$\frac{\begin{array}{c}\vdash \texttt{TE}_1 : \mu_1 \ldots \mu_p \qquad \vdash \texttt{TE}_2 : \mu_{p+1} \ldots \mu_{p+q} \\ \forall i \in 1..p+q, \quad \mu_i = \zeta_i \ \triangleright\ \langle\ \bar{\texttt{F}}^{(i)} \ \triangleright\ \overline{\sigma}^{(i)} \ \triangleright\ \bar{\texttt{I}}^{(i)}\ \rangle \qquad \cup_{i \in 1..p+q} \bar{\texttt{F}}^{(i)}\ \textbf{wf} \\ \zeta_1 \ldots \zeta_{p+q} \cup (\cup_{i \in 1..p+q} \overline{\sigma}^{(i)}) \cup \mathit{mSig}(\cup_{i \in 1..p+q} \bar{\texttt{I}}^{(i)})\ \textbf{wf}\end{array}}{\vdash \texttt{TE}_1 + \texttt{TE}_2 : \mu_1 \ldots \mu_{p+q}}\ (\text{T-TESUM})$$

$$\frac{\begin{array}{c}\vdash \texttt{TE} : \mu_1 \ldots \mu_n \qquad n \geq p \geq 1 \qquad \forall i \in 1..n, \quad \mu_i = \zeta_i \ \triangleright\ \langle\ \bar{\texttt{F}}^{(i)} \ \triangleright\ \overline{\sigma}^{(i)} \ \triangleright\ \bar{\texttt{I}}^{(i)}\ \rangle \\ \mathit{names}(\zeta_p) = \texttt{m} \qquad \texttt{m}' \notin \mathit{names}(\zeta_1 \ldots \zeta_n) \qquad \zeta_p[\texttt{m}'/\texttt{m}] \cup (\cup_{i \in 1..n} \overline{\sigma}^{(i)})\ \textbf{wf} \\ \mu = \zeta_p[\texttt{m}'/\texttt{m}] \ \triangleright\ \langle\ \bar{\texttt{F}}^{(p)} \ \triangleright\ \overline{\sigma}^{(p)} \ \triangleright\ \bar{\texttt{I}}^{(p)}\ \rangle\end{array}}{\vdash \texttt{TE}[\texttt{m aliasAs m}'] : \mu_1 \ldots \mu_n \mu}\ (\text{T-TEAL})$$

$$\frac{\begin{array}{c}(\texttt{m} \in \mathit{names}(\overline{\zeta} \cup \overline{\sigma})\ \textbf{implies}\ \texttt{m}' \notin \mathit{names}(\overline{\zeta} \cup \overline{\sigma})) \\ \vdash \texttt{TE} : \mu_1 \ldots \mu_n \qquad \forall i \in 1..n, \quad \mu_i = \zeta_i \ \triangleright\ \langle\ \bar{\texttt{F}}^{(i)} \ \triangleright\ \overline{\sigma}^{(i)} \ \triangleright\ \bar{\texttt{I}}^{(i)}\ \rangle \qquad \overline{\sigma} = \zeta_1 \ldots \zeta_n \cup \overline{\sigma}^{(1)} \cup \ldots \cup \overline{\sigma}^{(n)} \\ \texttt{m} \in \mathit{names}(\overline{\sigma}) \qquad (\overline{\sigma}[\texttt{m}/\texttt{m}'] \cup \mathit{mSig}(\cup_{i \in 1..n} \bar{\texttt{I}}^{(i)}))\ \textbf{wf} \\ \forall i \in 1..n, \quad \mu_i' = \zeta_i[\texttt{m}'/\texttt{m}] \ \triangleright\ \langle\ \bar{\texttt{F}}^{(i)} \ \triangleright\ \overline{\sigma}^{(i)}[\texttt{m}'/\texttt{m}] \ \triangleright\ \bar{\texttt{I}}^{(i)}\ \rangle\end{array}}{\vdash \texttt{TE}[\texttt{m renameTo m}'] : \mu_1' \ldots \mu_n'}\ (\text{T-TEREM})$$

$$\frac{\begin{array}{c}\vdash \texttt{TE} : \mu_1 \ldots \mu_n \qquad \forall i \in 1..n, \quad \mu_i = \zeta_i \ \triangleright\ \langle\ \bar{\texttt{F}}^{(i)} \ \triangleright\ \overline{\sigma}^{(i)} \ \triangleright\ \bar{\texttt{I}}^{(i)}\ \rangle \\ \bar{\texttt{F}} = \bar{\texttt{F}}^{(1)} \cup \ldots \cup \bar{\texttt{F}}^{(n)} \qquad \texttt{f} \in \mathit{names}(\bar{\texttt{F}}) \qquad \bar{\texttt{F}}[\texttt{f}'/\texttt{f}]\ \textbf{wf} \qquad \forall i \in 1..n, \quad \mu_i' = \zeta_i \ \triangleright\ \langle\ \bar{\texttt{F}}^{(i)}[\texttt{f}'/\texttt{f}] \ \triangleright\ \overline{\sigma}^{(i)} \ \triangleright\ \bar{\texttt{I}}^{(i)}\ \rangle\end{array}}{\vdash \texttt{TE}[\texttt{f renameTo f}'] : \mu_1' \ldots \mu_n'}\ (\text{T-TEREF})$$

Figure 5: FRTJ: Typing rules for non-basic record expressions and non-basic trait expressions

`this` (θ_i such that $i \notin \mathscr{T}$), the subtyping check between actual and formal parameter types is performed ($\theta_i <: \texttt{I}_i$). The check is not performed when the actual parameter is `this` (θ_i such that $i \in \mathscr{T}$); instead, the types of the corresponding actual parameters ($\texttt{I}_i$ such that $i \in \mathscr{T}$) are included in the required interfaces for `this` collected in the conclusion of the rule. The case when the receiver is different from `this` is similar. The only difference is that the signature of `m` is extracted from the type θ of the receiver, that is, either from the signatures of θ (if θ is an interface) or from the signatures of the interfaces implemented by θ (if θ is a class). Moreover, $\overline{\zeta}$ is set to the empty sequence.

The rule (T-NEW) for object creation is similar. It uses the field declarations in the class `C` to check the type of the arguments of the constructor. The rules for cast (T-UCAST) and (T-DCAST) are a straightforward adaptation of the corresponding rules proposed in [17]. Note that expressions like `(C)e` where the type of `e` is not a subtype of `C` (called *stupid casts* in [17]) are ill-typed.

4.3. Operational Semantics

Following a standard approach in the literature on traits [16] (see also [28, 23]), we specify the semantics of FRTJ by defining a "flattening" translation (that provides a canonical semantics for records and traits by compiling them

Expression typing:

$$\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x}) \mid \langle \bullet \mid \bullet \mid \bullet \rangle \ \text{(T-VAR)} \qquad \frac{\Gamma \vdash \mathtt{this} : \langle\, \bar{\mathtt{F}} \mid \ldots \,\rangle \mid \langle \bullet \mid \bullet \mid \bullet \rangle \qquad \mathit{choose}(\bar{\mathtt{F}}, \mathtt{f}) = \mathtt{I}\,\mathtt{f}}{\Gamma \vdash \mathtt{this.f} : \mathtt{I} \mid \langle\, \mathtt{I}\,\mathtt{f} \mid \bullet \mid \bullet \,\rangle} \ \text{(T-FIELD)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{e} : \theta \mid \langle\, \bar{\mathtt{F}}^{(0)} \mid \overline{\sigma}^{(0)} \mid \bar{\mathtt{I}}^{(0)} \,\rangle \qquad \forall i \in 1..n, \quad \Gamma \vdash \mathtt{e}_i : \theta_i \mid \langle\, \bar{\mathtt{F}}^{(i)} \mid \overline{\sigma}^{(i)} \mid \bar{\mathtt{I}}^{(i)} \,\rangle \\ \theta = \Gamma(\mathtt{this}) = \langle \ldots \mid \overline{\sigma} \,\rangle \ \textbf{implies} \ (\, \mathit{choose}(\overline{\sigma}, \mathtt{m}) = \mathtt{I}\,\mathtt{m}(\mathtt{I}_1, \ldots, \mathtt{I}_n) \ \textbf{and} \ \overline{\zeta} = \mathtt{I}\,\mathtt{m}(\mathtt{I}_1, \ldots, \mathtt{I}_n) \,) \\ \theta \neq \Gamma(\mathtt{this}) \ \textbf{implies} \ (\, \mathit{choose}(\mathit{mSig}(\theta), \mathtt{m}) = \mathtt{I}\,\mathtt{m}(\mathtt{I}_1, \ldots, \mathtt{I}_n) \ \textbf{and} \ \overline{\zeta} = \bullet \,) \\ \mathscr{T} = \{ i \mid i \in 1..n \ \textbf{and} \ \theta_i = \Gamma(\mathtt{this}) \} \qquad \forall i \in 1..n - \mathscr{T}, \quad \theta_i <: \mathtt{I}_i \end{array}}{\Gamma \vdash \mathtt{e.m}(\mathtt{e}_1, \ldots, \mathtt{e}_n) : \mathtt{I} \mid \langle\, \cup_{i \in 0..n} \bar{\mathtt{F}}^{(i)} \mid (\cup_{i \in 0..n} \overline{\sigma}^{(i)}) \cup \overline{\zeta} \mid (\cup_{i \in 0..n} \bar{\mathtt{I}}^{(i)}) \cup (\cup_{i \in \mathscr{T}} \mathtt{I}_i \,\rangle)} \ \text{(T-INVK)}$$

$$\frac{\begin{array}{c}\mathit{fields}(\mathtt{C}) = \mathtt{I}_1\,\mathtt{f}_1; \ldots; \mathtt{I}_n\,\mathtt{f}_n; \qquad \forall i \in 1..n, \quad \Gamma \vdash \mathtt{e}_i : \theta_i \mid \langle\, \bar{\mathtt{F}}^{(i)} \mid \overline{\sigma}^{(i)} \mid \bar{\mathtt{I}}^{(i)} \,\rangle \\ \mathscr{T} = \{ i \mid i \in 1..n \ \textbf{and} \ \theta_i = \Gamma(\mathtt{this}) = \langle \ldots \mid \overline{\sigma} \,\rangle \} \qquad \forall i \in 1..n - \mathscr{T}, \quad \theta_i <: \mathtt{I}_i \end{array}}{\Gamma \vdash \mathtt{new}\,\mathtt{C}(\mathtt{e}_1, \ldots, \mathtt{e}_n) : \mathtt{C} \mid \langle\, \cup_{i \in 1..n} \bar{\mathtt{F}}^{(i)} \mid \cup_{i \in 1..n} \overline{\sigma}^{(i)} \mid (\cup_{i \in 1..n} \bar{\mathtt{I}}^{(i)}) \cup (\cup_{i \in \mathscr{T}} \mathtt{I}_i \,\rangle)} \ \text{(T-NEW)}$$

$$\frac{\Gamma \vdash \mathtt{e} : \eta \mid \gamma \qquad \eta <: \mathtt{I}}{\Gamma \vdash (\mathtt{I})\mathtt{e} : \mathtt{I} \mid \gamma} \ \text{(T-UCAST)} \qquad \frac{\Gamma \vdash \mathtt{e} : \mathtt{J} \mid \gamma \qquad \mathtt{I} <: \mathtt{J} \qquad \mathtt{I} \neq \mathtt{J}}{\Gamma \vdash (\mathtt{I})\mathtt{e} : \mathtt{I} \mid \gamma} \ \text{(T-DCAST)}$$

Figure 6: FRTJ: Typing rules for expressions

away) and by providing a semantics for the subset of the language without records and traits. To this aim, we introduce FFRTJ (FLAT FRTJ), the subset of FRTJ where there are no record and trait declarations and the syntax of record and trait expressions is simplified as follows:

$$\begin{array}{lcl} \mathtt{RE} & ::= & \{\bar{\mathtt{F}};\} \\ \mathtt{TE} & ::= & \{\bar{\mathtt{F}};\ \bullet;\ \bar{\mathtt{M}}\} \end{array}$$

A FFRTJ program is a FRTJ program with empty record and trait tables. The translation removes the record and trait tables and replaces the class table with a class table containing only FFRTJ classes.

The translation is specified by the function $[\![\cdot]\!]$, given in Figure 7, that maps a FRTJ class declaration to a FFRTJ class declaration, a record expression to a sequence of field declarations, and a trait expression to a sequence of method declarations. We write $[\![\mathtt{CT}]\!]$ to denote the class table containing the translation of all the classes in CT. The clauses in Figure 7 are self-explanatory. Note that, in the translation of trait expressions, the clause for field renaming is simpler than the clause for method renaming (which uses the auxiliary function *mR*); this is due to the fact that fields can be accessed only on this.

A FFRTJ program is a 5-tuple $(\mathtt{IT}, \bullet, \bullet, \mathtt{CT}, \mathtt{e})$. Following FJ [17], the semantics of FFRTJ is given by means of a reduction relation of the form $\mathtt{e} \to \mathtt{e}'$, to be read "expression e reduces to expression $\mathtt{e}'$ in one step". We write $\to^\star$ to denote the reflexive and transitive of $\to$. Values are defined by the following syntax: $\boxed{\mathtt{v} ::= \mathtt{new}\,\mathtt{C}(\bar{\mathtt{v}})}$. The FFRTJ lookup functions, evaluation contexts/redexes and reduction rules are given in Figure 8.

4.4. FRTJ *Type Soundness*

The proof of the type soundness result for FRTJ consists of two steps. First, we show that the flattening translation preserves types. Second, we prove type soundness for FRTJ.

Theorem 4.2. *(Flattening Preserves the Type of Programs)*
If $\vdash (\mathtt{IT}, \mathtt{RT}, \mathtt{TT}, \mathtt{CT}, \mathtt{e}) : \eta$, *then* $\vdash (\mathtt{IT}, \bullet, \bullet, [\![\mathtt{CT}]\!], \mathtt{e}) : \eta$.

PROOF. See Appendix A. □

Theorem 4.3. *(*FRTJ *Type Soundness) If* $\vdash (\mathtt{IT}, \mathtt{RT}, \mathtt{TT}, \mathtt{CT}, \mathtt{e}) : \eta$ *and* $\mathtt{e} \to^\star \mathtt{e}'$ *with* $\mathtt{e}'$ *a normal form, then* $\mathtt{e}'$ *is: either a value* v *of type* C *and* $\mathtt{C} <: \eta$*; or an expression containing* $(\mathtt{I})\mathtt{new}\ \mathtt{C}(\bar{\mathtt{v}})$ *where* $\mathtt{C} \not<: \mathtt{I}$.

PROOF. See Appendix B. □

$$[\![\texttt{class C implements } \bar{\texttt{I}} \texttt{ by RE and TE}]\!] = \texttt{class C implements } \bar{\texttt{I}} \texttt{ by } \{\bar{\texttt{F}};\} \texttt{ and } \{\bar{\texttt{F}};\ \bullet;\ [\![\texttt{TE}]\!]\} \qquad \textbf{if } [\![\texttt{RE}]\!] = \bar{\texttt{F}}$$

$$[\![\{\bar{\texttt{F}};\}]\!] = \bar{\texttt{F}}$$
$$[\![\texttt{R}]\!] = [\![\texttt{RE}]\!] \qquad \textbf{if } \texttt{RT(R)} = \texttt{record R is RE}$$
$$[\![\texttt{RE}_1 + \texttt{RE}_2]\!] = [\![\texttt{RE}_1]\!] \cdot [\![\texttt{RE}_2]\!]$$
$$[\![\texttt{RE[exclude f]}]\!] = \mathit{exclude}([\![\texttt{RE}]\!], \texttt{f})$$
$$[\![\texttt{RE[f renameTo f}']]\!] = [\![\texttt{RE}]\!][\texttt{f}'/\texttt{f}]$$

$$[\![\{\bar{\texttt{F}};\ \bar{\texttt{S}};\ \bar{\texttt{M}}\}]\!] = \bar{\texttt{M}}$$
$$[\![\texttt{T}]\!] = [\![\texttt{TE}]\!] \qquad \textbf{if } \texttt{TT(T)} = \texttt{trait T is TE}$$
$$[\![\texttt{TE}_1 + \texttt{TE}_2]\!] = [\![\texttt{TE}_1]\!] \cdot [\![\texttt{TE}_2]\!]$$
$$[\![\texttt{TE[exclude m]}]\!] = \mathit{exclude}([\![\texttt{TE}]\!], \texttt{m})$$
$$[\![\texttt{TE[m aliasAs m}']]\!] = \bar{\texttt{M}} \cdot (\texttt{I m}'(\bar{\texttt{I}}\ \bar{\texttt{x}})\{\texttt{return e;}\}) \qquad \textbf{if } [\![\texttt{TE}]\!] = \bar{\texttt{M}} \textbf{ and } \texttt{I m}(\bar{\texttt{I}}\ \bar{\texttt{x}})\{\texttt{return e;}\} \in \bar{\texttt{M}}$$
$$[\![\texttt{TE[f renameTo f}']]\!] = [\![\texttt{TE}]\!][\texttt{f}'/\texttt{f}]$$
$$[\![\texttt{TE[m renameTo m}']]\!] = \mathit{mR}([\![\texttt{TE}]\!], \texttt{m}, \texttt{m}')$$

$$\mathit{mR}(\texttt{I n}(\bar{\texttt{I}}\ \bar{\texttt{x}})\{\texttt{return e;}\}, \texttt{m}, \texttt{m}') = \texttt{I n}[\texttt{m}'/\texttt{m}](\bar{\texttt{I}}\ \bar{\texttt{x}})\{\texttt{return e}[\texttt{this.m}'/\texttt{this.m}];\}$$
$$\mathit{mR}(\texttt{M}_1 \cdot \ldots \cdot \texttt{M}_n, \texttt{m}, \texttt{m}') = (\mathit{mR}(\texttt{M}_1, \texttt{m}, \texttt{m}')) \cdot \ldots \cdot (\mathit{mR}(\texttt{M}_n, \texttt{m}, \texttt{m}'))$$

Figure 7: Flattening FRTJ to FFRTJ

Fields, methods and interfaces lookup functions

$$\begin{array}{ll} \mathit{fields}(\texttt{C}) = \bar{\texttt{F}} & \textbf{if } \texttt{CT(C)} = \texttt{class C} \cdots \texttt{ by } \{\bar{\texttt{F}};\} \texttt{ and } \cdots \\ \mathit{methods}(\texttt{C}) = \bar{\texttt{M}} & \textbf{if } \texttt{CT(C)} = \texttt{class C} \cdots \texttt{ by } \cdots \texttt{ and } \{\cdots;\ \bullet;\ \bar{\texttt{M}}\} \\ \mathit{interfaces}(\texttt{C}) = \bar{\texttt{I}} & \textbf{if } \texttt{CT(C)} = \texttt{class C implements } \bar{\texttt{I}} \texttt{ by } \cdots \end{array}$$

Evaluation contexts and redexes:

$$\begin{array}{lcl} E & ::= & [\,] \mid E.\texttt{f} \mid E.\texttt{m}(\bar{\texttt{e}}) \mid \texttt{v.m}(\bar{\texttt{v}}, E, \bar{\texttt{e}}) \mid (\texttt{I})E \mid \texttt{new C}(\bar{\texttt{v}}, E, \bar{\texttt{e}}) \\ r & ::= & (\texttt{new C}(\bar{\texttt{v}})).\texttt{f} \mid (\texttt{new C}(\bar{\texttt{v}})).\texttt{m}(\bar{\texttt{v}}) \mid (\texttt{I})(\texttt{new C}(\bar{\texttt{v}})) \end{array}$$

Reduction rules:

$$\frac{\mathit{fields}(\texttt{C}) = \texttt{I}_1\,\texttt{f}_1; \ldots; \texttt{I}_n\,\texttt{f}_n}{E[(\texttt{new C}(\texttt{v}_1, \ldots, \texttt{v}_n)).\texttt{f}_i] \to E[\texttt{v}_i]} \qquad (\textsc{R-Field})$$

$$\frac{\texttt{I m}(\bar{\texttt{I}}\ \bar{\texttt{x}})\ \{\texttt{return e;}\} \in \mathit{methods}(\texttt{C})}{E[(\texttt{new C}(\bar{\texttt{v}})).\texttt{m}(\bar{\texttt{u}})] \to E[\texttt{e}[\bar{\texttt{u}}/\bar{\texttt{x}}, \texttt{new C}(\bar{\texttt{v}})/\texttt{this}]]} \qquad (\textsc{R-Invk})$$

$$\frac{\exists \texttt{J} \in \mathit{interfaces}(\texttt{C}), \quad \texttt{J} <: \texttt{I}}{E[(\texttt{I})(\texttt{new C}(\bar{\texttt{v}}))] \to E[\texttt{new C}(\bar{\texttt{v}})]} \qquad (\textsc{R-Cast})$$

Figure 8: FFRTJ: Operational Semantics

5. Implementing Type-safe Software Product Lines in FRTJ

In this section, the methodology how to implement type-safe SPLs in FRTJ is presented. Following [40], we say that a SPL is *type-safe* if all its product are well-typed programs (according to the type system of the language in which they are implemented). The implementation of SPLs in FRTJ relies on the fact that the FRTJ type system supports the type-checking of traits in isolation from the traits and classes that use them.

Theorem 5.1. *(*FRTJ *Type-checking)*
An FRTJ *program can be type-checked by type-checking only once its interfaces, record, traits, and classes.*

PROOF. Recall that the interface subtyping and record/trait reuse graphs are acyclic and observe that the FRTJ typing rules are such that:

- each interface can be type-checked in isolation from the interfaces, records, traits and classes that use it (by relying on the typing of the interfaces that it extends and on the information contained in the interface table);
- each record can be type-checked in isolation from the records and classes that use it (by relying on the typing of the records that it uses and on the information contained in the interface and record tables);
- each trait can be type-checked in isolation from the code of the traits and classes that use it (by relying on the typing of the traits that it uses and on the information contained in the interface, trait and class table tables);
- each class can be type-checked in isolation from the code of the other classes (by relying on the typing of the records, traits and interfaces that it uses and on the information contained in the interface, record, trait and class tables).

□

5.1. Formalizing Software Product Lines in FRTJ

A SPL consists of a set of products that are constructed from common artifacts in the SPL artifact base. A *product* is specified by the set of classes it uses. A *product specification* $\mathtt{PS}$ is a declaration

$$\mathtt{product\ P\ uses}\ \bar{\mathtt{C}}$$

where $\mathtt{P}$ is the name of the product and $\bar{\mathtt{C}}$ is a sequence of class names. The set of products contained in the SPL is captured in its product table. A *product table* $\mathtt{PT}$ is a map from product names $\mathtt{P}$ to product specifications $\mathtt{PS}$. A SPL $\mathtt{L}$ is a 5-tuple $(\mathtt{IT},\mathtt{RT},\mathtt{TT},\mathtt{CT},\mathtt{PT})$ where $\mathtt{PT}$ is the product table of the SPL and $(\mathtt{IT},\mathtt{RT},\mathtt{TT},\mathtt{CT})$ represents the artifact base. The artifact base contains the interfaces, records, traits and classes used to specify products. We assume that the tables $\mathtt{IT}$, $\mathtt{RT}$, $\mathtt{TT}$ and $\mathtt{CT}$ are *well-formed*, i.e., the tables $\mathtt{IT}/\mathtt{RT}/\mathtt{TT}/\mathtt{CT}$ contain an entry for each interface/record/trait/class used in the SPL and the interface subtyping and record/trait reuse graphs are acyclic.

The *code* of a product $\mathtt{P}$ is the 4-tuple $(\mathtt{IT_P},\mathtt{RT_P},\mathtt{TT_P},\mathtt{CT_P})$, where $\mathtt{IT_P}$, $\mathtt{RT_P}$, $\mathtt{TT_P}$ and $\mathtt{CT_P}$ are the subtables of $\mathtt{IT}$, $\mathtt{RT}$, $\mathtt{TT}$ and $\mathtt{CT}$ containing exactly the entries for the interfaces, records, traits and classes reachable from the classes contained in the product $\mathtt{P}$. We assume that each product specification $\mathtt{product\ P\ uses}\ \bar{\mathtt{C}}$ of $\mathtt{PT}$ is *well-formed*, that is: $\bar{\mathtt{C}}$ contains exactly the classes in $\mathtt{CT_P}$.

Figure 9 depicts the relations between a SPL artifact base, the products and the SPL. On the lowest level, the traits $\mathsf{T}_1,\mathsf{T}_2,\mathsf{T}_3$, the records $\mathsf{R}_1,\mathsf{R}_2,\mathsf{R}_3$ and interfaces $\mathsf{I}_1,\mathsf{I}_2,\mathsf{I}_3$ are used to built classes $\mathsf{C}_1,\mathsf{C}_2,\mathsf{C}_3$ constituting the SPL artifact base. The classes are used to built the products $\mathsf{P}_1,\mathsf{P}_2$ and P_3. which are contained in the SPLs L_1 and L_2.

Example 5.2. *Consider the bank account SPL introduced in Section 3. The SPL* BankLine*, described by the feature model in Figure 1, where the feature* Sync *is removed, is formalized as a 5-tuple* $(\mathtt{IT},\mathtt{RT},\mathtt{TT},\mathtt{CT},\mathtt{PT})$ *containing the entries for all interfaces/records/traits/classes given in Listings 1, 3, 4 and 5 and the five product specifications*

product ACCOUNT **uses** Account
product INV_ACCOUNT **uses** InvAccount
product RET_ACCOUNT **uses** RetAccount
product INV_ACCOUNT_WH **uses** InvAccount, Client
product RET_ACCOUNT_WH **uses** RetAccount, Client

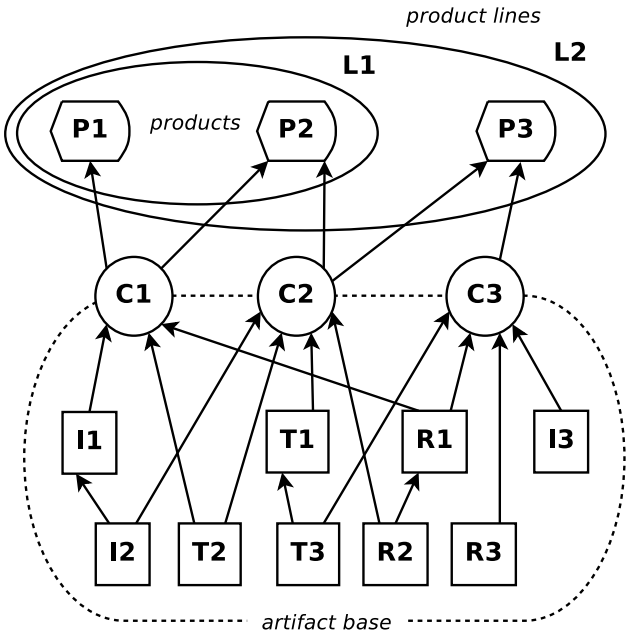

Figure 9: Relation between Artifacts, Products and SPL

The type system of FRTJ supports the development of type-safe product lines. We write $\vdash \mathtt{L}$, to be read: "the SPL L is well-typed", i.e., the code of every product P in L is well-typed. As a consequence of Theorem 5.1 ,the type safety of a FRTJ SPL can be verified without type-checking all its products individually, since it is enough to type-check each artifact in the artifact base only once.

Theorem 5.3. *(*FRTJ *SPL Type-checking)*
A SPL L *can be type-checked by type-checking only once its interfaces, records, traits and classes.*

PROOF. Immediate from Theorem 5.1.

□

5.2. Extending FRTJ *Software Product Lines*

The proposed formalization of SPL in the FRTJ calculus easily allows extending a SPL with further products. These products can also use traits, records, interfaces and classes that are not contained in the original artifact base. The SPL $\mathtt{L}' = (\mathtt{IT}', \mathtt{RT}', \mathtt{TT}', \mathtt{CT}', \mathtt{PT}')$ is an *extension* of the SPL $\mathtt{L} = (\mathtt{IT}, \mathtt{RT}, \mathtt{TT}, \mathtt{CT}, \mathtt{PT})$ if L has been obtained from L by adding interfaces, records, traits, classes and products (that is if $\mathtt{IT} \subseteq \mathtt{IT}'$, $\mathtt{RT} \subseteq \mathtt{RT}'$, $\mathtt{TT} \subseteq \mathtt{TT}'$, $\mathtt{CT} \subseteq \mathtt{CT}'$ and $\mathtt{PT} \subseteq \mathtt{PT}'$ hold). In Figure 9, the SPL L_2 is an extension of SPL L_1.

Example 5.4. *The SPL* BankLine′*, described by the feature model in Figure 1* ***with*** *the* Sync *feature, extends the SPL* BankLine *of Example 5.2. It can be formalized by adding to the SPL* BankLine *the code in Listings 2, 6 and 7 and the five product specifications*

product SYNC_ACCOUNT **uses** SyncAccount
product SYNC_INV_ACCOUNT **uses** SyncInvAccount
product SYNC_RET_ACCOUNT **uses** SyncRetAccount
product SYNC_INV_ACCOUNT_WH **uses** SyncInvAccount, Client
product SYNC_RET_ACCOUNT_WH **uses** SyncRetAccount, Client

A further consequence of Theorem 5.1 is that for ensuring the type safety of the extended SPL BankLine′ only the newly added records, traits, interfaces, classes, and products must be type-checked.

Theorem 5.5. *(*FRTJ *SPL Extension Type-checking) Let the SPL* $\mathtt{L}' = (\mathtt{IT}', \mathtt{RT}', \mathtt{TT}', \mathtt{CT}', \mathtt{PT}')$ *be an extension of the SPL* $\mathtt{L} = (\mathtt{IT}, \mathtt{RT}, \mathtt{TT}, \mathtt{CT}, \mathtt{PT})$*. If* L *has already been type-checked (so that the typings of all its artifacts are available), then the products in* $\mathtt{PT}' - \mathtt{PT}$ *can be type-checked without type-checking the artifacts of* L *and by type-checking only once the interfaces, records, traits, classes in* $\mathtt{IT}' - \mathtt{IT}$, $\mathtt{RT}' - \mathtt{RT}$, $\mathtt{TT}' - \mathtt{TT}$, $\mathtt{CT}' - \mathtt{CT}$*, respectively.*

PROOF. Immediate from Theorem 5.1.

□

6. Related Work

In object-oriented programming languages, class-based inheritance allows reusing code by subclassing. However, inheritance is often too restrictive to implement feature-based variability of SPLs since it allows code reuse only within the class hierarchy. Furthermore, inheritance does not support the removal of product functionality. Hence, there are several approaches providing other linguistic constructs than inheritance for flexibly implementing the variability of SPLs in the object-oriented paradigm. The approaches to implementing SPLs can be classified into two main directions [21]. First, *annotative approaches*, such as conditional compilation, frames [5] and COLORED FEATHERWEIGHT JAVA (CFJ) [19], mark the source code of the whole SPL with respect to product features and remove marked code depending on the feature configuration. Second, *compositional approaches* (like the calculus FRTJ presented in this paper) assemble products from artifacts in a common artifact base.

Compositional implementations of SPLs in the object-oriented paradigm use a variety of program modularization mechanisms, such as aspects [20], framed aspects [26], mixins [36], or hyperslices [39]. In these approaches, feature-based variability is restricted to the expressivity of the underlying programming paradigm. For instance, aspects only allow to add code at defined pointcuts, but cannot remove code. In [25], product line variability is implemented in SCALA [29] using mixin-based inheritance. While SCALA provides means to modularize classes and to extend them by adding classes, fields and methods via mixins (called "traits" in SCALA), the specification of the desired composition is less flexible than in FRTJ.

Traits are well suited for designing libraries and enable clean design and reuse which has been shown using SMALLTALK/SQUEAK (see, e.g., [10, 13]). Recently, [7] pointed out limitations of the trait model caused by the fact that methods provided by a trait can only access state by accessor methods (which become required methods of the trait). To avoid this, traits are made *stateful* (in a SMALLTALK/SQUEAK-like setting) by adding private fields that can be accessed from the clients possibly under a new name or merged with other variables. In FRTJ traits are stateless. By their required fields, however, it is possible to directly access state within the methods provided by a trait. Moreover, the names of required fields (in traits) and provided fields (in records) are unimportant because of the field rename operation. Since field renaming works synergically with method renaming, exclusion and aliasing, FRTJ has more reuse potential than stateful traits.

In feature-oriented programming (FOP) [6], the implementation of a product line is modularized into feature modules, each referring to one product feature. Feature modules can define new classes and refine existing classes. In order to realize a particular feature configuration, the respective feature modules are composed. The calculus LIGHTWEIGHT FEATURE JAVA (LFJ) [15], based on LJ (LIGHTWEIGHT JAVA) [38] provides a formalization of FOP together with a constraint-based type system (similar to the one in [2]) that supports the type-checking of feature modules in isolation. For each feature module, a set of constraints is inferred that are imposed by the introduction and refinement operations of the feature modules. The type safety of a SPL in LFJ can be verified by checking the validity of a generated propositional formula expressing the type safety of all products that can be derived according to the constraints of the feature model. The FEATHERWEIGHT FEATURE JAVA for Product Lines (FFJ_{PL}) calculus [3] proposes an independently developed type checking approach for feature-oriented product lines. FFJ_{PL} relies on FFJ [4], a calculus for stepwise-refinement, that is not explicitly bound to implementing SPLs. In FFJ_{PL}, feature-oriented mechanisms, such as class/method refinements, are modeled directly by the dynamic semantics of the language instead of by a translation into JAVA code. The FFJ_{PL} typing rules do not generate constraints, but directly consult the feature model. Modular type-checking is not supported in FFJ_{PL} since each feature module is analyzed by relying on information of the complete product line.

Delta-oriented programming (DOP) [32, 34] is an extension of feature-oriented programming. The implementation of a SPL in DOP is split into delta modules which extend feature modules by including removal of classes,

methods and fields. A delta module refers to a set of feature configurations for which the specified modifications have to be carried out. In order to generate a particular product, the modifications of the applicable delta modules are applied in an ordering that is compatible with an explicitly specified application ordering. In [32], product generation starts from a designated core product, while in [34], the product line is derived by applying delta modules to the empty product which makes DOP a true generalization of FOP. In [33], a compositional type system for IFΔJ, a core calculus for delta-oriented product lines of JAVA programs based on IFJ (IMPERATIVE FEATHERWEIGHT JAVA), an imperative version of FJ [17], is presented. Similar to LFJ, it is equipped with a constraint-based type system that infers constraints for each delta module in isolation. Instead of generating one propositional formula for all derivable products, an abstraction representation of delta modules and a corresponding abstract product generation is defined such that the type safety of products can be established by only considering the delta module constraints and the product abstractions.

Both LFJ [15] and IFΔJ [33] are equipped with a constraint-based type system that supports compositional type-checking by analyzing in isolation each feature module or delta module, respectively. This makes it possible to type-check extensions of a SPL with new products by analyzing only the code of the newly added feature or delta modules. Also FRTJ supports efficiently type-checking extensions of a SPL. If new artifacts are added to the artifact base, the existing class/trait/record/interface tables do not have to be considered, but it suffices to type-check the code of the new classes/traits/records/interfaces. FOP and DOP techniques for implementing SPLs provide explicit mechanisms for relating product features and code artifacts implementing them. The idea of replacing class-based inheritance by the more flexible trait and record composition (which is at the basis of the calculus presented in this paper) and the idea of complementing class-based inheritance by class refinement/modification (which is at the basis of FOP/DOP) are indeed orthogonal. In particular, we believe that adding a notion similar to feature/delta modules on top of FRTJ represents an interesting direction for future work.

7. Conclusions and Future Work

In this paper, we presented a novel approach to implement product line variability by trait and record composition. FRTJ programs may look more verbose than standard class-based programs; however, the degree of reuse provided by records and traits is higher than the reuse potential of standard static class-based hierarchies. The FRTJ type system is able to ensure type-safety of a SPL by type-checking its artifacts only once and to ensure type-safety of an extension of a (type-safe) SPL by checking only the newly added parts. A prototypical implementation of a language based on the FRTJ calculus, called SWRTJ (SUGARED WELTERWEIGHT RECORD-TRAIT JAVA) [9], containing some syntactic sugar not present in the underlying calculus, is available at `http://swrtj.sourceforge.net`.

For future work, we plan to investigate the possibility of adding a feature module [15] or a delta module [34] construct to FRTJ in order to explicitly represent the link between product features and code artifacts implementing them. Additionally, we aim at developing a process for building up an artifact base supporting as much code reuse as possible for implementing a particular SPL and evaluate this at larger case examples. The process will include guidelines on how features in a feature model can be represented best by traits, records and interfaces and how the resulting classes should be assembled. An IDE that allows viewing the different code artifacts from the perspective of the feature model of the SPL has to be developed assisting the programmer in managing the created artifacts.

References

[1] E. Allen, D. Chase, J. Hallett, V. Luchangco, G.-W.Maessen, S. Ryu, G. Steele, and S. Tobin-Hochstad. The Fortress Language Specification, V. 1.0, 2008.

[2] D. Ancona, F. Damiani, S. Drossopoulou, and E. Zucca. Polymorphic Bytecode: Compositional Compilation for Java-like Languages. In *POPL*, pages 26–37. ACM, 2005.

[3] S. Apel, C. Kästner, A. Grösslinger, and C. Lengauer. Type safety for feature-oriented product lines. *Automated Software Engineering*, 17(3):251–300, 2010.

[4] S. Apel, C. Kästner, and C. Lengauer. Feature Featherweight Java: A Calculus for Feature-Oriented Programming and Stepwise Refinement. In *GPCE*, pages 101–112. ACM, 2008.

[5] P. G. Bassett. *Framing software reuse: lessons from the real world*. Prentice-Hall, Inc., 1997.

[6] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *ICSE*, pages 187–197, 2003.

[7] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Stateful traits and their formalization. *Computer Languages, Systems & Structures*, 34(2-3):83–108, 2008.

[8] L. Bettini, F. Damiani, and I. Schaefer. Implementing Software Product Lines using Traits. In *Proc. of SAC, OOPS Track*, pages 2096 –2102. ACM, 2010.
[9] L. Bettini, F. Damiani, I. Schaefer, and F. Strocco. A Prototypical Java-like Language with Records and Traits. In *PPPJ*, pages 2096–2102. ACM, 2010.
[10] A. P. Black, N. Schärli, and S. Ducasse. Applying traits to the smalltalk collection classes. In *OOPSLA*, pages 47–64. ACM, 2003.
[11] V. Bono, F. Damiani, and E. Giachino. Separating Type, Behavior, and State to Achieve Very Fine-grained Reuse. In *Electronic proceedings of FTfJP*, 2007.
[12] V. Bono, F. Damiani, and E. Giachino. On Traits and Types in a Java-like setting. In *TCS (Track B)*, volume 273 of *IFIP*, pages 367–382. Springer, 2008.
[13] D. Cassou, S. Ducasse, and R. Wuyts. Redesigning with traits: the nile stream trait-based library. In *ICDL*, pages 50–75. ACM, 2007.
[14] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley Longman, 2001.
[15] B. Delaware, W. Cook, and D. Batory. A Machine-Checked Model of Safe Composition. In *FOAL*, pages 31–35. ACM, 2009.
[16] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *ACM TOPLAS*, 28(2):331–388, 2006.
[17] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
[18] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie Mellon Software Engineering Institute, 1990.
[19] C. Kästner and S. Apel. Type-Checking Software Product Lines - A Formal Approach. In *ASE*, pages 258–267. IEEE, 2008.
[20] C. Kästner, S. Apel, and D. S. Batory. A Case Study Implementing Features Using AspectJ. In *SPLC*, pages 223–232, 2007.
[21] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *ICSE*, pages 311–320. ACM, 2008.
[22] G. Lagorio, M. Servetto, and E. Zucca. Featherweight Jigsaw - A minimal core calculus for modular composition of classes. In *ECOOP*, volume 5653 of *LNCS*, pages 244–268. Springer, 2009.
[23] G. Lagorio, M. Servetto, and E. Zucca. Flattening versus direct semantics for Featherweight Jigsaw. In *FOOL*, 2009.
[24] L. Liquori and A. Spiwack. FeatherTrait: A Modest Extension of Featherweight Java. *ACM TOPLAS*, 30(2), 2008.
[25] R. E. Lopez-Herrejon, D. S. Batory, and W. R. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *ECOOP*, pages 169–194, 2005.
[26] N. Loughran and A. Rashid. Framed aspects: Supporting variability and configurability for aop. In *ICSR*, volume 3107 of *LNCS*, pages 127–140. Springer, 2004.
[27] L. Mikhajlov and E. Sekerinski. A Study of the Fragile Base Class Problem. In *ECOOP*, number 1445 in LNCS, pages 355–383. Springer, 1998.
[28] O. Nierstrasz, S. Ducasse, and N. Schärli. Flattening traits. *JOT (www.jot.fm)*, 5(4):129–148, 2006.
[29] M. Odersky. The Scala Language Specification, version 2.4. Technical report, Programming Methods Laboratory, EPFL, 2007.
[30] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering - Foundations, Principles, and Techniques*. Springer, 2005.
[31] J. Reppy and A. Turon. Metaprogramming with traits. In *ECOOP*, volume 4609 of *LNCS*, pages 373–398. Springer, 2007.
[32] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented Programming of Software Product Lines. In *SPLC*, volume 6287 of *LNCS*, pages 77–91. Springer, 2010.
[33] I. Schaefer, L. Bettini, and F. Damiani. Compositional type-checking of delta-oriented programming. In *AODS 2011*. ACM, 2011. To appear (available from `http://www.di.unito.it/~damiani/AOSD11.pdf`).
[34] I. Schaefer and F. Damiani. Pure Delta-oriented Programming. In *FOSD 2010*, 2010. Available from `http://www.cse.chalmers.se/~schaefer/FOSD10.pdf`.
[35] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *ECOOP*, volume 2743 of *LNCS*, pages 248–274. Springer, 2003.
[36] Y. Smaragdakis and D. S. Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2):215–255, 2002.
[37] C. Smith and S. Drossopoulou. *Chai*: Traits for Java-like languages. In *ECOOP*, volume 3586 of *LNCS*, pages 453–478. Springer, 2005.
[38] R. Strniša, P. Sewell, and M. Parkinson. The Java module system: core design and semantic definition. In *OOPSLA*, pages 499–514. ACM, 2007.
[39] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N degrees of separation: multi-dimensional separation of concerns. In *ICSE*, pages 107–119, 1999.
[40] S. Thaker, D. S. Batory, D. Kitchin, and W. R. Cook. Safe Composition of Product Lines. In *GPCE*, pages 95–104. ACM, 2007.
[41] D. Ungar, C. Chambers, B.-W. Chang, and U. Hölzle. Organizing Programs Without Classes. *Lisp and Symbolic Computation*, 4(3):223–242, July 1991.

A. Proof of Theorem 4.2 (Flattening Preserves the Type of Programs)

Notation A.1. *The sequence of the field names and the sequence of the method names selected on* `this` *in the expressions* `e` *are denoted by fN*(`e`) *and mN*(`e`)*, respectively. The sequence of the field names and the sequence of the method names selected on* `this` *in the method declaration* $\texttt{M} = \texttt{Im}(\bar{\texttt{I}}\,\texttt{x})\{\texttt{return e}\}$ *are given by fN*(`M`) $=$ *fN*(`e`) *and mN*(`M`) $=$ *mN*(`e`)*. The definitions of fN and mN naturally extend to sequences of expressions and sequences of method definitions.*

Recall that the flattening $[\![\texttt{TE}]\!]$ of a trait expression `TE` yields a sequence of methods and that the flattening $[\![\texttt{RE}]\!]$ of a record expression `RE` yields a sequence of fields.

Notation A.2. *Given a sequence of methods* $\bar{\mathtt{M}}$*, we will write "*$\mathtt{this} : \langle\ \bar{\mathtt{F}}\ \mid\ \overline{\sigma}\ \rangle \vdash \mathtt{M}_1 ... \mathtt{M}_n : \mu_1 ... \mu_n$*" as short for "*$\mathtt{this} : \langle\ \bar{\mathtt{F}}\ \mid\ \overline{\sigma}\ \rangle \vdash \mathtt{M}_1 : \mu_1$*, ...,* $\mathtt{this} : \langle\ \bar{\mathtt{F}}\ \mid\ \overline{\sigma}\ \rangle \vdash \mathtt{M}_n : \mu_n$*.*

Lemma A.3. *Let* $\mathtt{this} : \langle\ \bar{\mathtt{F}}\ \mid\ \overline{\sigma}\ \rangle \vdash \mathtt{M} : \mu$*, where* $\mu = \mathtt{I}\ \mathtt{m}\ (\bar{\mathtt{I}})\ \mid\ \langle\ \bar{\mathtt{F}}'\ \mid\ \overline{\sigma}'\ \mid\ \bar{\mathtt{I}}'\ \rangle$*. Then* $\mathtt{this} : \langle\ \bar{\mathtt{F}}''\ \mid\ \overline{\sigma}''\ \rangle \vdash \mathtt{M} : \mu$ *for all* $\bar{\mathtt{F}}'' \supseteq \bar{\mathtt{F}}'$ *and* $\overline{\sigma}'' \supseteq \overline{\sigma}'$*.*

PROOF. By structural induction on typing derivations.

□

Lemma A.4. *Let* $\vdash \mathtt{TE} : \mu_1 \cdots \mu_n$*, where* $\mu_i = \zeta_i\ \mid\ \langle\ \bar{\mathtt{F}}^{(i)}\ \mid\ \overline{\sigma}^{(i)}\ \mid\ \bar{\mathtt{I}}^{(i)}\ \rangle$ *(for all* $i \in 1..n$*). Then* $\mathtt{this} : \langle\ \bar{\mathtt{F}}\ \mid\ \overline{\zeta}\ \rangle \vdash$ *names*$(\llbracket \mathtt{TE} \rrbracket) : \mu_1 \cdots \mu_n$*, where* $\bar{\mathtt{F}} = \bar{\mathtt{F}}^{(1)} \cup \cdots \cup \bar{\mathtt{F}}^{(n)}$ *and* $\overline{\zeta} = \zeta_1 \cdots \zeta_n \cup \overline{\sigma}^{(1)} \cup \cdots \cup \overline{\sigma}^{(n)}$*.*

PROOF. By case induction on the flattening translation for trait expressions defined in Fig. 7.

Case $\llbracket \{\bar{\mathtt{F}};\ \bar{\mathtt{S}};\ \bar{\mathtt{M}}\} \rrbracket$**.** This is the base case of the induction. The fact that $\bar{\mathtt{M}}$ is well-typed follows directly from the fact that $\{\bar{\mathtt{F}};\ \bar{\mathtt{S}};\ \bar{\mathtt{M}}\}$ is well-typed, in particular $\bar{\mathtt{M}}$ is well-typed under the assumption $\mathtt{this} : \langle\ \bar{\mathtt{F}}\ \mid\ \mathit{mSig}(\bar{\mathtt{S}}) \cup \mathit{mSig}(\bar{\mathtt{M}})\ \rangle$.

Case $\llbracket \mathtt{T} \rrbracket$**.** Straightforward, by induction hypothesis.

Case $\llbracket \mathtt{TE}_1 + \mathtt{TE}_2 \rrbracket$**.** By induction hypothesis we have that $\mathtt{this} : \langle\ \bar{\mathtt{F}}'\ \mid\ \overline{\sigma}'\ \rangle \vdash \llbracket \mathtt{TE}_1 \rrbracket : \bar{\mu}_1$ and $\mathtt{this} : \langle\ \bar{\mathtt{F}}''\ \mid\ \overline{\sigma}''\ \rangle \vdash \llbracket \mathtt{TE}_2 \rrbracket : \bar{\mu}_2$. Then the result follows by Lemma A.3.

Case $\llbracket \mathtt{TE}[\mathtt{exclude}\ \mathtt{m}] \rrbracket$**.** By induction hypothesis we have that $\mathtt{this} : \langle\ \bar{\mathtt{F}}\ \mid\ \overline{\sigma}\ \rangle \vdash \llbracket \mathtt{TE} \rrbracket : \bar{\mu}$. Then we have two possible cases for the type of `this` in the typing of the sequence of methods $\llbracket \mathtt{TE}[\mathtt{exclude}\ \mathtt{m}] \rrbracket$: (*i*) $\mathtt{m} \notin \mathit{mN}(\llbracket \mathtt{TE}[\mathtt{exclude}\ \mathtt{m}] \rrbracket)$. Then for each method $\mathtt{n} \neq \mathtt{m}$ in $\mathit{names}(\llbracket \mathtt{TE}[\mathtt{exclude}\ \mathtt{m}] \rrbracket)$ we have that $\mathtt{this} : \langle\ \bar{\mathtt{F}}\ \mid\ \mathit{exclude}(\overline{\sigma}, \mathtt{m})\ \rangle \vdash \mathtt{I}\ \mathtt{n}\ (\bar{\mathtt{I}}\ \bar{\mathtt{x}})\{\mathtt{return}\ \mathtt{e};\} : \mu$, where $\mathtt{I}\ \mathtt{n}\ (\bar{\mathtt{I}}\ \bar{\mathtt{x}})\{\mathtt{return}\ \mathtt{e};\} = \mathit{choose}(\llbracket \mathtt{TE}[\mathtt{exclude}\ \mathtt{m}] \rrbracket, \mathtt{n})$ and $\mu = \mathit{choose}(\bar{\mu}, \mathtt{n})$, by Lemma A.3. So $\mathtt{this} : \langle\ \bar{\mathtt{F}}\ \mid\ \mathit{exclude}(\overline{\sigma}, \mathtt{m})\ \rangle \vdash \llbracket \mathtt{TE}[\mathtt{exclude}\ \mathtt{m}] \rrbracket : \mathit{exclude}(\bar{\mu}, \mathtt{m})$. (*ii*) $\mathtt{m} \in \mathit{mN}(\llbracket \mathtt{TE}[\mathtt{exclude}\ \mathtt{m}] \rrbracket)$. Then we have that $\mathtt{this} : \langle\ \bar{\mathtt{F}}\ \mid\ \overline{\sigma}\ \rangle \vdash \llbracket \mathtt{TE}[\mathtt{exclude}\ \mathtt{m}] \rrbracket : \mathit{exclude}(\bar{\mu}, \mathtt{m})$.

Case $\llbracket \mathtt{TE}[\mathtt{m}\ \mathtt{aliasAs}\ \mathtt{m}'] \rrbracket$**.** This case is similar to the case $\llbracket \mathtt{TE}_1 + \mathtt{TE}_2 \rrbracket$.

Case $\llbracket \mathtt{TE}[\mathtt{m}\ \mathtt{renameTo}\ \mathtt{m}'] \rrbracket$**.** By induction hypothesis we have that $\mathtt{this} : \langle\ \bar{\mathtt{F}}\ \mid\ \overline{\sigma}\ \rangle \vdash \llbracket \mathtt{TE} \rrbracket : \bar{\mu}$. Then we have two possible cases for the type of `this` in the typing of the sequence of methods $\mathit{mR}(\llbracket \mathtt{TE} \rrbracket, \mathtt{m}, \mathtt{m}')$: (*i*) $\langle\ \bar{\mathtt{F}}\ \mid\ \mathit{exclude}(\overline{\sigma}, \mathtt{m}) \cup (\mathit{choose}(\overline{\sigma}, \mathtt{m})[{}^{\mathtt{m}'}/_{\mathtt{m}}])\ \rangle$, if $\mathtt{m}'$ is fresh; (*ii*) $\langle\ \bar{\mathtt{F}}\ \mid\ \mathit{exclude}(\overline{\sigma}, \mathtt{m})\ \rangle$, if $\mathtt{m}' \in \mathit{names}(\llbracket \mathtt{TE} \rrbracket)$. Note that case (*ii*) can happen only if `m` and the occurrence of $\mathtt{m}'$ already in `TE` have the same signature, and it is not the case that both `m` and $\mathtt{m}'$ are supplied (they can be both required or one of them required), otherwise $\mathtt{TE}[\mathtt{m}\ \mathtt{renameTo}\ \mathtt{m}']$ would have not been well-typed, which contradicts the hypothesis. In both cases the result can be proved straightforwardly by induction on typing derivations.

Case $\llbracket \mathtt{TE}[\mathtt{f}\ \mathtt{renameTo}\ \mathtt{f}'] \rrbracket$**.** By induction hypothesis we have that $\mathtt{this} : \langle\ \bar{\mathtt{F}}\ \mid\ \overline{\sigma}\ \rangle \vdash \llbracket \mathtt{TE} \rrbracket : \bar{\mu}$. Then we have two possible cases for the type of `this` in the typing of the sequence of methods $\llbracket \mathtt{TE} \rrbracket[{}^{\mathtt{f}'}/_{\mathtt{f}}]$: (*i*) $\langle\ \mathit{exclude}(\bar{\mathtt{F}}, \mathtt{f}) \cup (\mathit{choose}(\bar{\mathtt{F}}, \mathtt{f})[{}^{\mathtt{f}'}/_{\mathtt{f}}])\ \mid\ \overline{\sigma}\ \rangle$, if $\mathtt{f}'$ is fresh; (*ii*) $\langle\ \mathit{exclude}(\bar{\mathtt{F}}, \mathtt{f})\ \mid\ \overline{\sigma}\ \rangle$, if $\mathtt{f}' \in \mathit{fN}(\llbracket \mathtt{TE} \rrbracket)$. Note that case (*ii*) can happen only if `f` and the occurrence of $\mathtt{f}'$ already in `TE` have the same type, otherwise $\mathtt{TE}[\mathtt{f}\ \mathtt{renameTo}\ \mathtt{f}']$ would have not been well-typed, which contradicts the hypothesis. In both cases the result can be proved straightforwardly by induction on typing derivations.

□

Lemma A.5. *If the class declaration* $\mathtt{class}\ \mathtt{C}\ \mathtt{by}\ \bar{\mathtt{J}}\ \mathtt{by}\ \mathtt{RE}\ \mathtt{and}\ \mathtt{TE}$ *is well-typed, then its translation* $\llbracket \mathtt{class}\ \mathtt{C}\ \mathtt{by}\ \bar{\mathtt{J}}\ \mathtt{by}\ \mathtt{RE}\ \mathtt{and}\ \mathtt{TE} \rrbracket$ *is well-typed.*

PROOF. We have $\llbracket \mathtt{class}\ \mathtt{C}\ \mathtt{by}\ \bar{\mathtt{J}}\ \mathtt{by}\ \mathtt{RE}\ \mathtt{and}\ \mathtt{TE} \rrbracket = \mathtt{class}\ \mathtt{C}\ \mathtt{implements}\ \bar{\mathtt{I}}\ \mathtt{by}\ \{\bar{\mathtt{F}};\}\ \mathtt{and}\ \{\bar{\mathtt{F}};\ \bullet;\ \llbracket \mathtt{TE} \rrbracket\}$, where $\bar{\mathtt{F}} = \llbracket \mathtt{RE} \rrbracket$. By Lemma A.4, the sequence of methods $\llbracket \mathtt{TE} \rrbracket$ is well-typed. The proof follows then by observing that all method requirements present in a trait expression `TE` are enforced by the rule (C-OK), therefore the basic trait $\{\bar{\mathtt{F}};\ \bullet;\ \llbracket \mathtt{TE} \rrbracket\}$ is well-typed, and so it is the translation.

□

$$\Delta \vdash' \mathtt{x} : \Delta(\mathtt{x}) \ (\text{T-VAR}') \qquad \frac{\Delta \vdash' \mathtt{e} : \mathtt{C} \qquad \mathit{fields}(\mathtt{C}) = \bar{\mathtt{F}} \qquad \mathit{choose}(\bar{\mathtt{F}}, \mathtt{f}) = \mathtt{I}\,\mathtt{f}}{\Delta \vdash' \mathtt{e}.\mathtt{f} : \mathtt{I}} \ (\text{T-FIELD}')$$

$$\frac{\Delta \vdash' \mathtt{e} : \eta \qquad \mathit{mSig}'(\eta) = \overline{\sigma} \qquad \mathit{choose}(\overline{\sigma}, \mathtt{m}) = \mathtt{I}\,\mathtt{m}(\mathtt{I}_1, ..., \mathtt{I}_n) \qquad \forall i \in 1..n, \quad \Delta \vdash' \mathtt{e}_i : \eta_i \qquad \eta_i <: \mathtt{I}_i}{\Delta \vdash' \mathtt{e}.\mathtt{m}(\mathtt{e}_1, ..., \mathtt{e}_n) : \mathtt{I}} \ (\text{T-INVK}')$$

$$\frac{\mathit{fields}(\mathtt{C}) = \mathtt{I}_1\,\mathtt{f}_1, ..., \mathtt{I}_n\,\mathtt{f}_n \qquad \forall i \in 1..n, \quad \Delta \vdash' \mathtt{e}_i : \eta_i \qquad \eta_i <: \mathtt{I}_i}{\Delta \vdash' \mathtt{new}\,\mathtt{C}(\mathtt{e}_1, ..., \mathtt{e}_n) : \mathtt{C}} \ (\text{T-NEW}')$$

$$\frac{\Delta \vdash' \mathtt{e} : \eta \qquad \eta <: \mathtt{I}}{\Delta \vdash' (\mathtt{I})\mathtt{e} : \mathtt{I}} \ (\text{T-UCAST}') \qquad \frac{\Delta \vdash' \mathtt{e} : \mathtt{J} \qquad \mathtt{I} <: \mathtt{J} \qquad \mathtt{I} \neq \mathtt{J}}{\Delta \vdash' (\mathtt{I})\mathtt{e} : \mathtt{I}} \ (\text{T-DCAST}')$$

$$\frac{\Delta \vdash' \mathtt{e} : \eta \qquad \eta \not<: \mathtt{I} \qquad \mathtt{I} \not<: \eta \qquad \textbf{stupid warning}}{\Delta \vdash' (\mathtt{I})\mathtt{e} : \mathtt{I}} \ (\text{T-SCAST}')$$

Figure 10: FFRTJ: Typing rules for runtime expressions

Lemma A.6. *If a class table* CT *is well-typed, then its translation by flattening* $[\![\mathtt{CT}]\!]$ *is well-typed.*

PROOF. This follows from Lemma A.5.

□

Lemma A.7. *Let* e *be an expression. If there exists an interface table* IT*, a record table* RT*, a trait table* TT*, and a class table* CT *for which the judgement* $\Gamma \vdash \mathtt{e} : \theta \mid \gamma$ *holds, for a basis* Γ*, a type* θ*, and constraints* γ*, then, by using the interface table* IT *and the class table* $[\![\mathtt{CT}]\!]$*, there exists a basis* Γ' *such that:*

1. *if* $\mathtt{e} \neq \mathtt{this}$*: the judgement* $\Gamma' \vdash \mathtt{e} : \theta \mid \gamma$ *holds;*
2. *if* $\mathtt{e} = \mathtt{this}$*: the judgement* $\Gamma' \vdash \mathtt{this} : \Gamma'(\mathtt{this}) \mid \langle \bullet \mid \bullet \mid \bullet \rangle$ *holds.*

PROOF. By induction on the typing derivation $\Gamma \vdash e : \theta \mid \gamma$ using Lemma A.6. Notice that the basis Γ' is constructed as shown in Lemma A.4.

□

PROOF OF THEOREM 4.2 (Flattening Preserves the Type of Programs). This is a straightforward corollary of Lemma A.7.

□

B. Proof of Theorem 4.3 (FRTJ Type Soundness)

To prove the type soundness result, we need to consider a suitable notion of typing for runtime expressions. As for FJ [17], the syntax of runtime expressions is the same as the one for expressions. Constraints are not required to prove the type soundness for FFRTJ, therefore, the typing for runtime expressions does not consider constraints. An environment for runtime expressions Δ is either a finite mapping from variable names (including this) to types, written "$\bar{\mathtt{x}} : \bar{\mathtt{I}}$, this : C", or the empty mapping, written "$\bullet$". The typing judgement for runtime expressions is $\boxed{\Delta \vdash' \mathtt{e} : \eta}$ to be read: "under the assumption in Δ, the runtime expression e is well-typed with type η". The typing rules for runtime expressions are given in Fig. 10.

The expression e in rule (T-FIELD′) of Fig. 10 can be either this or new...(...); this is enforced by the shape of the premises of the rule itself. The lookup function *fields* used in rules (T-FIELD′) and (T-NEW′) is the one defined in Figure 8, and the lookup function mSig' used in rule (T-INVK′) is defined by

$$\mathit{mSig}'(\eta) = \begin{cases} \mathit{mSig}(\mathit{methods}(\mathtt{C})) & \textbf{if } \eta \text{ is a class } \mathtt{C} \\ \mathit{mSig}(\eta) & \text{otherwise} \end{cases}$$

where *methods* is the lookup function defined in Figure 8 and *mSig* is the lookup function defined in Figure 3. Note that, following FJ [17], we introduce a rule for dealing with *stupid casts*, that is, expression of the form $(\mathtt{I})\mathtt{e}$ where the type of $\mathtt{e}$ is not related with the type $\mathtt{I}$. In fact, although not present in well-typed programs, stupid casts may be introduced during reduction. The relation between the FFRTJ type system and the type system for runtime expressions is stated by the following theorem.

Theorem B.1. *(Well-typed expressions are well-typed at runtime) If* $\bullet \vdash \mathtt{e} : \eta \mid \langle \bullet \mid \bullet \mid \bullet \rangle$, *then* $\bullet \vdash' \mathtt{e} : \eta$.

PROOF. By induction on expressions. We show only the case $\mathtt{new\,C}(\bar{\mathtt{e}})$. The other cases are straightforward. Since $\bullet \vdash \mathtt{new\,C}(\bar{\mathtt{e}}) : \mathtt{C} \mid \gamma$, then by rule (T-NEW) for flattened expressions (that is the same as rule (T-NEW) in Fig. 6 but with a translated class table) we have that

$$\mathit{fields}(\mathtt{C}) = \mathtt{I}_1\,\mathtt{f}_1;\ldots;\mathtt{I}_n\,\mathtt{f}_n; \qquad \forall i \in 1..n, \qquad \bullet \vdash \mathtt{e}_i : \eta_i \mid \langle \bullet \mid \bullet \mid \bullet \rangle \quad \eta_i <: \mathtt{I}_i.$$

Notice that, since the environment is $\bullet$, the set $\mathscr{T}$ defined in rule (T-NEW) is empty. By induction hypothesis on $\mathtt{e}_i$ we have that $\forall i \in 1..n$, $\bullet \vdash' \mathtt{e}_i : \eta_i$. Therefore we can apply runtime rule (T-NEW′) and we have that $\bullet \vdash' \mathtt{new\,C}(\mathtt{e}_1,\ldots,\mathtt{e}_n) : \mathtt{C}$. □

Type soundness is proved by using the standard technique of subject reduction and progress theorems.

Theorem B.2. *(Subject Reduction) If* $\Delta \vdash' \mathtt{e} : \eta$ *and* $\mathtt{e} \to \mathtt{e}'$, *then* $\Delta \vdash' \mathtt{e}' : \eta'$ *and* $\eta' <: \eta$.

PROOF. See Appendix C.

□

Theorem B.3. *(Progress) Suppose* $\mathtt{e}$ *is a well-typed expression.*

1. *If* $\mathtt{e} = E[(\mathtt{new\,C}(\bar{\mathtt{v}}).\mathtt{f}]$, *then* $\mathit{fields}(\mathtt{C}) = \bar{\mathtt{I}}\,\bar{\mathtt{f}}$ *and* $\mathtt{f} \in \bar{\mathtt{f}}$ *for some* $\bar{\mathtt{I}}$ *and* $\bar{\mathtt{f}}$.
2. *If* $\mathtt{e} = E[(\mathtt{new\,C}(\bar{\mathtt{v}}).\mathtt{m}(\bar{\mathtt{u}})]$, *then*
   $\mathtt{class\,C} \cdots \{ \cdots \mathtt{m}(\mathtt{I}_1\,\mathtt{x}_1,\ldots,\mathtt{I}_n\,\mathtt{x}_n)\ \{\ \mathtt{return}\ \mathtt{e}_0;\ \} \cdots \}$ *and* $\#\bar{\mathtt{u}} = \#\bar{\mathtt{x}}$, *for some* $\bar{\mathtt{I}}$, $\bar{\mathtt{x}}$ *and* $\mathtt{e}_0$.

PROOF. See Appendix C.

□

Theorem B.4. *(*FFRTJ *Type Soundness)*
If $\bullet \vdash' \mathtt{e} : \eta$ *and* $\mathtt{e} \to^\star \mathtt{e}'$ *with* $\mathtt{e}'$ *a normal form, then* $\mathtt{e}'$ *is* (i) *either a value* $\mathtt{v}$ *with* $\bullet \vdash' \mathtt{v} : \mathtt{C}$ *and* $\mathtt{C} <: \eta$, *or* (ii) *an expression containing* $(\mathtt{I})\mathtt{new\,C}(\bar{\mathtt{e}})$ *where* $\mathtt{C} \not<: \mathtt{I}$.

PROOF. Immediate from Theorems B.2 and B.3.

□

PROOF OF THEOREM 4.3 (FRTJ Type Soundness). Immediate from Theorems 4.2, B.1 and B.4.

□

C. Proof of Theorems B.2 (Subject Reduction) and B.3 (Progress)

Lemma C.1. *(Unique Decomposition) Suppose that the runtime expression* $\mathtt{e}$ *is not a value and is such that* $\Delta \vdash' \mathtt{e} : \eta$ *for some* Δ *and* η. *There is a unique evaluation context* E *such that* $\mathtt{e} = E[r]$ *for some redex* r.

PROOF.

Case $\mathtt{e.f}$**.** If $\mathtt{e}$ is a value then the evaluation context is $[\,]$. Otherwise we can apply the induction hypothesis to $\mathtt{e}$ deriving that there is a unique evaluation context E such that $\mathtt{e} = E[r]$ for some redex r. Therefore, $E.\mathtt{f}$ is the unique evaluation context for $\mathtt{e.f}$.

Case $\mathtt{e}'.\mathtt{m}(\bar{\mathtt{e}})$**.** Either $\mathtt{e}' = \mathtt{v}$ or we can apply the induction hypothesis to $\mathtt{e}'$ deriving that the unique evaluation context for $\mathtt{e}'.\mathtt{m}(\bar{\mathtt{e}})$ is $E.\mathtt{m}(\bar{\mathtt{e}})$. If $\mathtt{e}'$ is a value, either for all $\mathtt{e} \in \bar{\mathtt{e}}$ we have that $\mathtt{e} = \mathtt{v}$ for some $\mathtt{v}$ in which case the expression is a redex and the evaluation context is $[]$, or there is a minimum j such that $\mathtt{e}_j$ is not a value and for all $k < j$ the expression $\mathtt{e}_k$ is a value. In this case we apply the induction hypothesis to $\mathtt{e}_j$ deriving that $\mathtt{v}.\mathtt{m}(\mathtt{v}_1, \ldots, \mathtt{v}_{j-1}, E, \mathtt{e}_{j+1}, \ldots)$ is the unique evaluation context for $\mathtt{e}'.\mathtt{m}(\bar{\mathtt{e}})$.

Cases $\mathtt{new}\,\mathtt{C}(\bar{\mathtt{e}})$ **and** $(\mathtt{I})\mathtt{e}$**.** Similar to the previous ones.

□

Lemma C.2. *If* $\Delta \vdash' E[r] : \eta$*, then for some* η' *we have that* $\Delta \vdash' r : \eta'$*.*

PROOF. Straightforward induction on the derivation of $\Delta \vdash' E[r] : \eta$.

□

Lemma C.3. *Let* $\Delta \vdash' E[r] : \eta$ *where* $\Delta \vdash' r : \eta_1$*, and let* $\mathtt{e}$ *be such that* $\Delta \vdash' \mathtt{e} : \eta_1'$ *for some* η_1' *such that* $\eta_1' <: \eta_1$*. Then* $\Delta \vdash' E[\mathtt{e}] : \eta'$ *for some* η' *such that* $\eta' <: \eta$*.*

PROOF. By induction on evaluation contexts E.

Case $[]$**.** Immediate.

Case $E.\mathtt{f}$**.** Let $\Delta \vdash' E[r].\mathtt{f} : \eta$ where $\Delta \vdash' r : \eta_1$, and let $\mathtt{e}$ be such that $\Delta \vdash' \mathtt{e} : \eta_1'$ for some η_1' such that $\eta_1' <: \eta_1$. From $\Delta \vdash' E[r].\mathtt{f} : \eta$, by rule (T-FIELD′), we have that

$$\Delta \vdash' E[r] : \mathtt{C} \qquad \mathit{fields}(\mathtt{C}) = \bar{\mathtt{F}} \qquad \mathit{choose}(\bar{\mathtt{F}}, \mathtt{f}) = \mathtt{I}\,\mathtt{f} \qquad \eta = \mathtt{I}.$$

By induction hypothesis on E we have that $\Delta \vdash' E[\mathtt{e}] : \mathtt{C}$, since $\mathtt{C} <: \mathtt{C}$. Applying rule (T-FIELD′) we get that $\Delta \vdash' E[\mathtt{e}].\mathtt{f} : \eta$ which proves the result.

Case $E.\mathtt{m}(\bar{\mathtt{e}})$**.** Similar to the previous one.

Case $\mathtt{v}.\mathtt{m}(\bar{\mathtt{v}}, E, \bar{\mathtt{e}})$**.** Let $\Delta \vdash' \mathtt{v}.\mathtt{m}(\bar{\mathtt{v}}, E[r], \bar{\mathtt{e}}) : \eta$, where $\Delta \vdash' r : \eta_1$, and let $\mathtt{e}$ be such that $\Delta \vdash' \mathtt{e} : \eta_1'$ for some η_1' such that $\eta_1' <: \eta_1$. From $\Delta \vdash' \mathtt{v}.\mathtt{m}(\bar{\mathtt{v}}, E[r], \bar{\mathtt{e}}) : \eta$, by rule (T-INVK′), we have that $\Delta \vdash' E[r] : \eta_2$. By induction hypothesis on E we have that $\Delta \vdash' E[\mathtt{e}] : \eta_2'$ for some $\eta_2' <: \eta_2$. Then, applying rule (T-INVK′) (and for the transitivity of subtyping) we get that $\Delta \vdash' \mathtt{v}.\mathtt{m}(\bar{\mathtt{v}}, E[\mathtt{e}], \bar{\mathtt{e}}) : \eta$ which proves the result.

Case $\mathtt{new}\,\mathtt{C}(\bar{\mathtt{v}}, E, \bar{\mathtt{e}})$**.** Similar to the previous one.

Case $(\mathtt{I})E$**.** Let $\Delta \vdash' (\mathtt{I})E[r] : \mathtt{I}$ where $\Delta \vdash' r : \eta_1$, and let $\mathtt{e}$ be such that $\Delta \vdash' \mathtt{e} : \eta_1'$ for some η_1' such that $\eta_1' <: \eta_1$. From $\Delta \vdash' (\mathtt{I})E[r] : \mathtt{I}$, by rules (T-UCAST′), (T-DCAST′) and (T-SCAST′), we have that $\Delta \vdash' E[r] : \eta$. By induction hypothesis on E we have that $\Delta \vdash' E[\mathtt{e}] : \eta'$ for some $\eta' <: \eta$. Applying the one of the rules (T-UCAST′), (T-DCAST′) or (T-SCAST′), depending on the subtyping relation between η and $\mathtt{I}$, we get $\Delta \vdash' (\mathtt{I})E[\mathtt{e}] : \mathtt{I}$.

□

Lemma C.4. *(Weakening) If* $\Delta \vdash' \mathtt{e} : \eta$*, then* $\Delta, \mathtt{x} : \mathtt{I} \vdash' \mathtt{e} : \eta$*.*

PROOF. Straightforward induction on the derivation of $\Delta \vdash' \mathtt{e} : \eta$.

□

Lemma C.5. *(Substitution) If* $\Delta, \bar{\mathtt{x}} : \bar{\eta}' \vdash' \mathtt{e} : \eta$ *and* $\Delta \vdash' \bar{\mathtt{v}} : \overline{\eta}$*, where* $\overline{\eta} <: \bar{\eta}'$*, then* $\Delta \vdash' \mathtt{e}[\bar{\mathtt{v}}/\bar{\mathtt{x}}] : \eta'$*, for some* $\eta' <: \eta$*.*

PROOF. By induction on expressions.

Case $\mathtt{x}$**.** Since $\Delta, \bar{\mathtt{x}} : \bar{\eta}' \vdash' \mathtt{x} : \eta$ for some η, we have that $\mathtt{x} \in \bar{\mathtt{x}}$ that is $\mathtt{x} = \mathtt{x}_j$ for some j. Therefore, $\Delta, \bar{\mathtt{x}} : \bar{\eta}' \vdash' \mathtt{x} : \eta_j'$ and $\mathtt{x}[\bar{\mathtt{v}}/\bar{\mathtt{x}}] = \mathtt{v}_j$. From $\Delta \vdash' \mathtt{v}_j : \eta_j$ where $\eta_j <: \eta_j'$ we derive the result.

Case $\mathtt{e.f}$**.** Since $\Delta, \bar{\mathtt{x}} : \bar{\eta}' \vdash' \mathtt{e.f} : \eta$ for some η, we have that

$$\Delta, \bar{\mathtt{x}} : \bar{\eta}' \vdash' \mathtt{e} : \mathtt{C} \qquad \mathit{fields}(\mathtt{C}) = \bar{\mathtt{F}} \qquad \mathit{choose}(\bar{\mathtt{F}}, \mathtt{f}) = \mathtt{I}\,\mathtt{f} \qquad \eta = \mathtt{I}$$

for some $\mathtt{I}$, $\bar{\mathtt{F}}$ and $\mathtt{C}$. By induction hypothesis on $\mathtt{e}$ we have that $\Delta \vdash' \mathtt{e}[\bar{\mathtt{v}}/\bar{\mathtt{x}}] : \mathtt{C}$ since $\mathtt{C} <: \mathtt{C}$. Applying rule (T-FIELD′) we get that $\Delta \vdash' (\mathtt{e}[\bar{\mathtt{v}}/\bar{\mathtt{x}}]).\mathtt{f} : \eta$ which proves the result.

Case $\mathtt{e.m}(\bar{\mathtt{e}})$**.** Since $\Delta, \bar{\mathtt{x}} : \bar{\eta}' \vdash' \mathtt{e.m}(\bar{\mathtt{e}}) : \eta$ for some η, we have that

$$\Delta, \bar{\mathtt{x}} : \bar{\eta}' \vdash' \mathtt{e} : \eta' \qquad \mathit{choose}(\mathit{mSig}'(\eta'), \mathtt{m}) = \mathtt{I}\,\mathtt{m}(\mathtt{I}'_1, ..., \mathtt{I}'_n) \qquad \forall i \in 1..n, \quad \Delta, \bar{\mathtt{x}} : \bar{\eta}' \vdash' \mathtt{e}_i : \eta_i \quad \eta_i <: \mathtt{I}'_i$$

for some $\bar{\eta}$, and where $\eta = \mathtt{I}$. By induction hypothesis on $\mathtt{e}$ we have that $\Delta \vdash' \mathtt{e}[\bar{\mathtt{v}}/\bar{\mathtt{x}}] : \eta''$ for some η'' such that $\eta'' <: \eta'$. By rules (I-OK) and (I-OK) we have that $\mathit{choose}(\mathit{mSig}'(\eta''), \mathtt{m}) = \mathit{choose}(\mathit{mSig}'(\eta'), \mathtt{m})$. By induction on $\mathtt{e}_i \in \bar{\mathtt{e}}$ we have that $\Delta \vdash' \mathtt{e}_i[\bar{\mathtt{v}}/\bar{\mathtt{x}}] : \eta''_i$ for some η''_i such that $\eta''_i <: \eta_i <: \mathtt{I}'_i$. Applying rule (T-INVK′) we get that $\Delta \vdash' (\mathtt{e.m}(\bar{\mathtt{e}}))[\bar{\mathtt{v}}/\bar{\mathtt{x}}] : \eta$ which proves the result.

Case $\mathtt{new}\ \mathtt{C}(\bar{\mathtt{e}})$**.** Similar to the previous one.

Case $(\mathtt{I})\mathtt{e}$**.** Since $\Delta, \bar{\mathtt{x}} : \bar{\eta}' \vdash' (\mathtt{I})\mathtt{e} : \mathtt{I}$, we have that $\Delta, \bar{\mathtt{x}} : \bar{\eta}' \vdash' \mathtt{e} : \eta$ for some η. By induction hypothesis on $\mathtt{e}$ we have that $\Delta \vdash' \mathtt{e}[\bar{\mathtt{v}}/\bar{\mathtt{x}}] : \eta$ for some η. Applying the one of the rules (T-UCAST′), (T-DCAST′) or (T-SCAST′), depending on the subtyping relation between η and $\mathtt{I}$, we get $\Delta \vdash' (\mathtt{I})\mathtt{e} : \mathtt{I}$.

□

Lemma C.6. *If* $\vdash \mathtt{class}\ \mathtt{C}\ \mathtt{implements}\ \bar{\mathtt{I}}\ \mathtt{by}\ \{\bar{\mathtt{G}};\}\ \mathtt{and}\ \{\bar{\mathtt{F}};\ \bullet;\ \bar{\mathtt{M}}\}$ OK*, and* $\mathtt{this} : \langle\, \bar{\mathtt{F}} \ \triangleright\ \mathit{mSig}(\bar{\mathtt{M}}) \,\rangle, \bar{\mathtt{x}} : \bar{\mathtt{J}} \vdash \mathtt{e} : \theta \ \triangleright\ \gamma$*, then* $\mathtt{this} : \mathtt{C}, \bar{\mathtt{x}} : \bar{\mathtt{J}} \vdash' \mathtt{e} : \eta$ *for some* η *such that: if* $\theta = \langle\, \bar{\mathtt{F}} \ \triangleright\ \mathit{mSig}(\bar{\mathtt{M}}) \,\rangle$ *then* $\eta = \mathtt{C}$*, else* $\eta <: \theta$*.*

PROOF. By straightforward induction on the derivation of $\mathtt{this} : \langle\, \bar{\mathtt{F}} \ \triangleright\ \mathit{mSig}(\bar{\mathtt{M}}) \,\rangle, \bar{\mathtt{x}} : \bar{\mathtt{J}} \vdash \mathtt{e} : \theta \ \triangleright\ \gamma$.

□

Lemma C.7. *If* $\vdash \mathtt{class}\ \mathtt{C}\ \mathtt{implements}\ \bar{\mathtt{I}}\ \mathtt{by}\ \{\bar{\mathtt{G}};\}\ \mathtt{and}\ \{\bar{\mathtt{F}};\ \bullet;\ \bar{\mathtt{M}}\}$ OK*, and* $\mathtt{J}\,\mathtt{m}\,(\bar{\mathtt{J}}\,\bar{\mathtt{x}})\{\mathtt{return}\ \mathtt{e};\} \in \bar{\mathtt{M}}$*, then* $\mathtt{this} : \mathtt{C}, \bar{\mathtt{x}} : \bar{\mathtt{J}} \vdash' \mathtt{e} : \eta$ *for some* η *such that* $\eta <: \mathtt{J}$*.*

PROOF. From hypothesis and rule (C-OK) we have

$$\vdash \{\bar{\mathtt{F}};\ \bullet;\ \bar{\mathtt{M}}\} : \mu_1 ... \mu_p \qquad p \geq 0 \qquad \forall i \in 1..p, \quad \mu_i = \zeta_i \ \triangleright\ \langle\, \bar{\mathtt{F}}^{(i)} \ \triangleright\ \overline{\sigma}^{(i)} \ \triangleright\ \bar{\mathtt{I}}^{(i)} \,\rangle$$
$$\cup_{i \in 1..p} \bar{\mathtt{F}}^{(i)} = \bar{\mathtt{F}} \qquad ((\cup_{i \in 1..p} \overline{\sigma}^{(i)}) \cup \mathit{mSig}(\bar{\mathtt{I}})) \subseteq \zeta_1 ... \zeta_p \qquad \forall \mathtt{I}' \in \cup_{i \in 1..p} \bar{\mathtt{I}}^{(i)},\ \exists \mathtt{I} \in \bar{\mathtt{I}}, \quad \mathtt{I} <: \mathtt{I}'.$$

From rule (T-TEBAS) we have

$$\mathtt{this} : \langle\, \bar{\mathtt{F}} \ \triangleright\ \zeta_1 ... \zeta_p \,\rangle \vdash \mathtt{J}\,\mathtt{m}\,(\bar{\mathtt{J}}\,\bar{\mathtt{x}})\{\mathtt{return}\ \mathtt{e};\} : \mathtt{J}\,\mathtt{m}\,(\bar{\mathtt{J}}) \ \triangleright\ \langle\, \bar{\mathtt{F}}^{(j)} \ \triangleright\ \overline{\sigma}^{(j)} \ \triangleright\ \bar{\mathtt{I}}^{(j)} \,\rangle,$$

for some j. From rule (M-OK) we have

$$\mathtt{this} : \langle\, \bar{\mathtt{F}} \ \triangleright\ \zeta_1 ... \zeta_p \,\rangle, \bar{\mathtt{x}} : \bar{\mathtt{J}} \vdash \mathtt{e} : \theta \ \triangleright\ \langle\, \bar{\mathtt{F}}^{(j)} \ \triangleright\ \overline{\sigma}^{(j)} \ \triangleright\ \bar{\mathtt{J}}' \,\rangle,$$

where if $\theta = \langle\, \bar{\mathtt{F}} \ \triangleright\ \zeta_1 ... \zeta_p \,\rangle$ then $(\zeta_1 ... \zeta_p) \cup \mathit{mSig}(\mathtt{J})$ **wf** and $\bar{\mathtt{I}}^{(j)} = \bar{\mathtt{J}}' \cup \mathtt{J}$, else $\theta <: \mathtt{J}$ and $\bar{\mathtt{I}}^{(j)} = \bar{\mathtt{J}}'$.

If $\theta = \langle\, \bar{\mathtt{F}} \ \triangleright\ \zeta_1 ... \zeta_p \,\rangle$ then, by Lemma C.6, we have $\mathtt{this} : \mathtt{C}, \bar{\mathtt{x}} : \bar{\mathtt{J}} \vdash' \mathtt{e} : \mathtt{C}$. Since, from rule (C-OK) we have that $\exists \mathtt{I} \in \bar{\mathtt{I}}$ such that $\mathtt{I} <: \mathtt{J}$, and since $\mathtt{C} <: \mathtt{I}$ by definition of subtyping for runtime expressions, then $\mathtt{C} <: \mathtt{J}$, again by subtyping rules for runtime expressions.

If $\theta \neq \langle\, \bar{\mathtt{F}} \ \triangleright\ \zeta_1 ... \zeta_p \,\rangle$ then, by Lemma C.6, we have $\mathtt{this} : \mathtt{C}, \bar{\mathtt{x}} : \bar{\mathtt{J}} \vdash' \mathtt{e} : \eta$, where $\eta <: \theta <: \mathtt{J}$.

□

PROOF OF THEOREM B.2 (Subject Reduction). By Lemma C.1 we get that $\mathtt{e} = E[r]$ for some r. The proof is by case analysis on the operational semantics rule used.

Case (R-FIELD)**.** Then

$$E[(\texttt{new}\,\texttt{C}(\bar{\texttt{v}})).\texttt{f}_i] \to E[\texttt{v}_i]$$

where $\mathit{fields}(\texttt{C}) = \bar{\texttt{I}}\,\bar{\texttt{f}}$, for some $\bar{\texttt{I}}$ and $\bar{\texttt{f}}$. Since $\Delta \vdash' E[(\texttt{new}\,\texttt{C}(\bar{\texttt{v}})).\texttt{f}_i] : \eta$, from Lemma C.2 we get $\Delta \vdash' (\texttt{new}\,\texttt{C}(\bar{\texttt{v}})).\texttt{f}_i : \eta'$ for some η'. From rule (T-FIELD′) we have that

$$\Delta \vdash' \texttt{new}\,\texttt{C}(\bar{\texttt{v}}) : \texttt{C} \qquad \mathit{choose}(\bar{\texttt{I}}\,\bar{\texttt{f}}, \texttt{f}_i) = \texttt{I}_i\,\texttt{f}_i \qquad \eta' = \texttt{I}_i.$$

From rule (T-NEW′) we have that

$$\forall i \in 1..n, \qquad \Delta \vdash' \texttt{v}_i : \eta_i \qquad \eta_i <: \texttt{I}_i$$

We can then apply Lemma C.3 and conclude that $\Delta \vdash' E[\texttt{v}_i] : \eta''$ for some $\eta'' <: \eta$.

Case (R-INVK)**.** Then

$$E[(\texttt{new}\,\texttt{C}(\bar{\texttt{v}})).\texttt{m}(\bar{\texttt{u}})] \to E[\texttt{e}[\bar{\texttt{u}}/\bar{\texttt{x}}, \texttt{new}\,\texttt{C}(\bar{\texttt{v}})/\texttt{this}]]$$

where $\texttt{class}\,\texttt{C} \cdots \{ \cdots \texttt{m}(\bar{\texttt{I}}\,\bar{\texttt{x}})\,\{\,\texttt{return}\,\texttt{e};\,\} \cdots \}$, for some $\texttt{e}$ and $\bar{\texttt{I}}$. Since $\Delta \vdash' E[(\texttt{new}\,\texttt{C}(\bar{\texttt{v}})).\texttt{m}(\bar{\texttt{u}})] : \eta$, from Lemma C.2 we get $\Delta \vdash' (\texttt{new}\,\texttt{C}(\bar{\texttt{v}})).\texttt{m}(\bar{\texttt{u}}) : \eta'$ for some η'. From rule (T-INVK′) we have that

$$\Delta \vdash' \texttt{new}\,\texttt{C}(\bar{\texttt{v}}) : \texttt{C} \qquad \mathit{choose}(\mathit{mSig}'(\texttt{C}), \texttt{m}) = \texttt{I}\,\texttt{m}(\texttt{J}_1, ..., \texttt{J}_n) \qquad \forall i \in 1..n, \quad \Delta \vdash' \texttt{u}_i : \eta_i \qquad \eta_i <: \texttt{J}_i \quad \eta' = \texttt{I}.$$

From rule (T-NEW′) we have that

$$\forall i \in 1..n, \quad \Delta \vdash' \texttt{v}_i : \eta_i' \quad \eta_i' <: \texttt{I}_i$$

Since class $\texttt{C}$ is well-formed, then method $\texttt{m}$ is well-typed and then, from Lemma C.7, we have that $\texttt{this} : \texttt{C}, \bar{\texttt{x}} : \bar{\texttt{J}} \vdash' \texttt{e} : \eta_0$, where $\eta_0 <: \eta' = \texttt{I}$. Therefore, by Lemma C.5, we have that $\Delta \vdash' \texttt{e}[\bar{\texttt{u}}/\bar{\texttt{x}}, \texttt{new}\,\texttt{C}(\bar{\texttt{v}})/\texttt{this}] : \eta_0'$, where $\eta_0' <: \eta_0$. We can then apply Lemma C.3 and conclude that $\Delta \vdash' E[\texttt{e}[\bar{\texttt{u}}/\bar{\texttt{x}}, \texttt{new}\,\texttt{C}(\bar{\texttt{v}})/\texttt{this}]] : \eta''$ for some $\eta'' <: \eta$.

Case (R-CAST)**.** Then

$$E[(\texttt{I})(\texttt{new}\,\texttt{C}(\bar{\texttt{e}}))] \to E[\texttt{new}\,\texttt{C}(\bar{\texttt{e}})]$$

where $\texttt{class}\,\texttt{C}\,\texttt{implements}\,\bar{\texttt{J}}\,\texttt{by}\,\{ \cdots \}$ and $\exists \texttt{J} \in \bar{\texttt{J}}$ such that $\texttt{J} <: \texttt{I}$. Since $\Delta \vdash' E[(\texttt{I})(\texttt{new}\,\texttt{C}(\bar{\texttt{e}}))] : \eta$, from Lemma C.2 we get $\Delta \vdash' (\texttt{I})(\texttt{new}\,\texttt{C}(\bar{\texttt{e}})) : \eta'$ for some η'. The proof of $\Delta \vdash' (\texttt{I})(\texttt{new}\,\texttt{C}(\bar{\texttt{e}})) : \eta'$ must end with rule (T-UCAST′), since the derivation ending with (T-SCAST′) or (T-DCAST′) contradicts the assumption $\exists \texttt{J} \in \bar{\texttt{J}}$ such that $\texttt{J} <: \texttt{I}$. By the rules (T-NEW′), and (T-UCAST′), we have that $\eta' = \texttt{I}$ and $\Delta \vdash' \texttt{new}\,\texttt{C}(\bar{\texttt{e}}) : \texttt{C}$, where $\texttt{C} <: \texttt{J} <: \texttt{I}$. We can then apply Lemma C.3 and conclude that $\Delta \vdash' E[\texttt{new}\,\texttt{C}(\bar{\texttt{e}})] : \eta''$ for some $\eta'' <: \eta$.

□

PROOF OF THEOREM B.3 (Progress). If $\texttt{e} = E[(\texttt{new}\,\texttt{C}(\bar{\texttt{v}})).\texttt{f}]$, then, by Lemma C.2, $(\texttt{new}\,\texttt{C}(\bar{\texttt{v}})).\texttt{f}$ is well-typed, and then it is easy to check that $\mathit{fields}(\texttt{C})$ is well defined and $\texttt{f}$ appears in it. Similarly, if $\texttt{e} = E[(\texttt{new}\,\texttt{C}(\bar{\texttt{v}})).\texttt{m}(\bar{\texttt{u}})]$, then, by Lemma C.2, $(\texttt{new}\,\texttt{C}(\bar{\texttt{v}})).\texttt{m}(\bar{\texttt{u}})$ is well-typed, and then it is also easy to show that $\texttt{I}\,\texttt{m}(\texttt{I}_1\,\texttt{x}_1, ..., \texttt{I}_n\,\texttt{x}_n)\{\texttt{return}\,\texttt{e}_0\}$ is defined in $\texttt{C}$, for some $\texttt{e}_0$ and $\texttt{I}$, and the property $\#\bar{\texttt{x}} = \#\bar{\texttt{u}}$ derives directly from the typing rule (T-INVK′), since $\#\bar{\texttt{I}} = \#\bar{\texttt{u}}$.
□