Implementing Software Product Lines using Traits*

Lorenzo Bettini Ferruccio Damiani
Dipartimento di Informatica, Università di Torino,
C.so Svizzera, 185 - 10149 Torino, Italy
{bettini,damiani}@di.unito.it

Ina Schaefer†
Chalmers University of Technology,
421 96 Gothenburg, Sweden
schaefer@chalmers.se

ABSTRACT

A *software product line (SPL)* is a set of software systems with well-defined commonalities and variabilities that are developed by managed reuse of common artifacts. In this paper, we present a novel approach to implement SPL by fine-grained reuse mechanisms which are orthogonal to class-based inheritance. We introduce the FEATHERWEIGHT RECORD-TRAIT JAVA (FRTJ) calculus where units of product functionality are modeled by *traits*, a construct that was already shown useful with respect to code reuse, and by *records*, a construct that complements traits to model the variability of the state part of products explicitly. Records and traits are assembled in classes that are used to build products. This composition of product functionalities 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 only once the records, traits and classes shared by different products. Moreover, type-safety of an extension of a (type-safe) SPL can be guaranteed by checking only the newly added parts.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.3 [**Programming Languages**]: Language Constructs and Features; F.3.3 [**Studies of Program Constructs**]: Type Structure

General Terms

Design, Languages, Theory

Keywords

Featherweight Java, Feature Model, Software Product Line, Trait, Type System

*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").

†This author has been partially supported by the Deutsche Forschungsgemeinschaft (DFG).

1. INTRODUCTION

A *software product line (SPL)* is a set of software systems with well-defined commonalities and variabilities [12]. SPL engineering aims at developing these systems by managed reuse. Products are implemented by artifacts from a common product line artifact base. The reuse mechanisms for the implementation have to be flexible enough to statically express SPL variability appropriately. 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 without re-checking already existing products.

Products of a SPL are commonly described in terms of *features*, where a feature is a unit of product functionality. In *feature-oriented programming (FOP)* [5], the artifacts of a SPL are organized in *feature modules*. A product is the result from a composition of features represented as feature modules. FOP approaches are typically based on the OO paradigm (see, e.g., [3, 13]). 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) [23, 14]. FOP approaches overcome the limitations of class-based inheritance by representing a feature module by a collection of 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 its parent. The composition of a feature module with an existing product introduces new classes and/or alters existing ones. Therefore, in different products the same class name can refer to different class definitions.

In this paper, we explore a novel approach to the development of SPL that provides flexible code reuse with static guarantees. The main idea is to overcome the limitations of class-based inheritance with respect to code reuse by replacing it with *trait composition*. A *trait* is a set of methods, completely independent from any class hierarchy. In the original proposals of traits [27, 14] (and in most of the subsequent formulation of traits within a JAVA-like nominal type system [29, 24, 26, 21]) trait composition and class-based inheritance live together as the main reuse construct and as a conceptual design tool, respectively. However, a taxonomy represented by a class hierarchy is an obstacle in implementing SPL. It limits the possibilities for composing products from building blocks in an arbitrary way. Therefore, in our approach, class-based inheritance is ruled out and classes are built only by composition of traits, interfaces and *records* (a construct, introduced in [9], representing the counterpart of traits with respect to the state). The presented approach separates the concepts of types, state, behavior into different and orthogonal linguistic concepts (interfaces, records and traits, respectively). These are the reusable building blocks that can be assembled into classes that are re-used in several products.

We formalize our approach in FEATHERWEIGHT RECORD-TRAIT JAVA (FRTJ), a minimal core calculus (in the spirit of FJ [15]) for interfaces, records, traits and classes. SPL 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 provides static guarantees on safe and consistent class assembly from records, traits and interfaces. Furthermore, it ensures that a SPL is type-safe (i.e, that all the products in the product line are type-safe). The approach easily allows for the safe extension of product lines by products using new classes, interfaces, traits, records and by providing means to efficiently type-check only the new parts.

The main differences of the trait-oriented approach presented in this paper with respect to the approaches based on FOP and class-based inheritance, such as [17, 28, 5, 3, 13], are as follows:

- The efficient 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, e.g., feature modules and class refinements.
- The classes, interfaces, records and traits of all the products co-exist 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.
- 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 considering only the newly added parts.

FRTJ programs may look more verbose than standard class-based programs; however, the degree of re-use provided by records and traits is higher than the re-use 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 SPL using fine-grained reuse mechanisms.

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.

2. INTRODUCING RECORDS AND TRAITS

Traits have been designed 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 be factored into a trait. Traits were introduced and implemented in the dynamically-typed class-based language SQUEAK/SMALLTALK [27, 14], to counter the problems of class-based inheritance with respect to code reuse.[1] Various formulations of traits in a JAVA-like setting can be found in the literature (see, e.g., [29, 24, 9, 26, 10, 21]). Also two recent programming languages, SCALA [25] and FORTRESS [1], incorporate forms of the trait construct.

In this paper, we use the concept of trait as described in [10]. 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 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. 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 [9] 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 [9] 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. [13]). 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 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.

[1] A related reuse construct called "trait" was introduced in [31], for the dynamically-typed prototype-based language SELF, to describe a parent object to which an object may delegate some of its behavior.

```
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

```
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

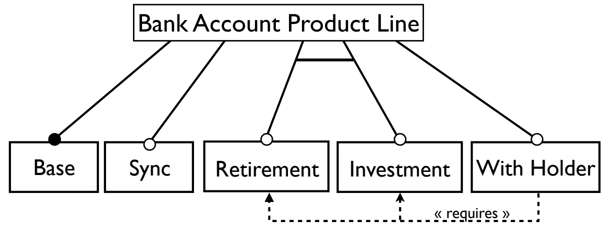

Figure 1: Feature Model for Bank Account Product Line

```
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

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 re-used 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 re-usability of traits to state re-usability of records, and fosters a programming style relying on small components that are easy to re-use.

Traits/records satisfy the so called *flattening principle* [24] (see also [20, 19]), 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 SPL

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 [13]. 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 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

```
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 the 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

```
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

```
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 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. This 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` 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.

```
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

```
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

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.

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

Figure 2: FRTJ: Syntax

4. THE FRTJ CALCULUS

In this section, we outline the FRTJ calculus, a minimal core calculus (in the spirit of FJ [15]) for interfaces, records, traits and classes.

As pointed out in [10], 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. Thus, in the FRTJ calculus, trait, record and class names are not types. The only user defined types are interface names. In this way, the reuse potential of traits and records is increased to appropriately capture product line variability.

Syntax. The syntax of FRTJ is given in Figure 2. We also consider a calculus, FFRTJ (FLAT FRTJ), obtained by removing the portions of the syntax highlighted in gray. We use the overbar sequence notation according to [15]. For instance, the pair "$\bar{\mathtt{I}}\,\bar{\mathtt{x}}$" stands for "$\mathtt{I}_1\,\mathtt{x}_1,...,\mathtt{I}_n\,\mathtt{x}_n$", and "$\bar{\mathtt{I}}\,\bar{\mathtt{f}};$" stands for "$\mathtt{I}_1\,\mathtt{f}_1;...;\mathtt{I}_n\,\mathtt{f}_n;$". The empty sequence is denoted by "$\bullet$".

In FRTJ, there are no constructor declarations. Like in FJ, the syntax of the constructor of a class is fixed with respect the field order, types and names: in every class C, we assume the implicit constructor $\mathtt{C(\bar{I}\,\bar{f})\{this.\bar{f} = \bar{f};\}}$, where $\bar{\mathtt{I}}\,\bar{\mathtt{f}}$ are the fields of C. Note that FFRTJ is indeed a subset of JAVA. The FFRTJ class $\mathtt{class\ C\ implements\ \bar{I}\ by\ \{\ \bar{I}\,\bar{f};\ \}\ and\ \{\ \bar{J}\,\bar{g};\ \bullet;\ \bar{M}\}}$ (where the fields $\bar{\mathtt{J}}\,\bar{\mathtt{g}}$ are a subset of the fields $\bar{\mathtt{I}}\,\bar{\mathtt{f}}$) can be understood as the JAVA class $\mathtt{class\ C\ implements\ \bar{I}\ \{\ \bar{I}\,\bar{f};\ C(\bar{I}\,\bar{f})\{this.\bar{f} = \bar{f};\}\ \bar{M}\ \}}$.

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, records and trait names to interface, records and trait declarations, respectively. A FRTJ program is a 5-tuple $(\mathtt{IT},\mathtt{RT},\mathtt{TT},\mathtt{CT},\mathtt{e})$, where the 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.

Types, Subinterfacing and Subtyping. Nominal types, ranged over by η, are either class names or interface names. The *subinterfacing relation* is the reflexive and transitive closure of the immediate subinterfacing relation declared by the extends clauses in the interface table IT. It is formalized by the judgment $\boxed{\mathtt{I}_1 \trianglelefteq \mathtt{I}_2}$ to be read: "$\mathtt{I}_1$ is a subinterface of $\mathtt{I}_2$". The *subtyping relation* for nominal types is the reflexive and transitive closure of the relation obtained by extending subinterfacing with the interface implementation relation declared by the implements clauses in the class table CT. It is formalized by the judgment $\boxed{\eta_1 <: \eta_2}$ to be read: "η_1 is a subtype of η_2".

Well-Typed* FRTJ *programs. 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.

Reduction. Following FJ [15], the semantics of FRTJ is given by means of a reduction relation of the form $\mathtt{e} \rightarrow \mathtt{e}'$, to be read "expression e reduces to expression $\mathtt{e}'$ in one step". We write $\rightarrow^\star$ to denote the reflexive and transitive of $\rightarrow$. Values are defined by the following syntax: $\boxed{\mathtt{v}\ ::=\ \mathtt{new\,C(\bar{v})}}$.

Properties. Type soundness can be proved by using the standard technique of subject reduction and progress theorems.

THEOREM 4.1 (FRTJ TYPE SOUNDNESS). *If* $\vdash (\mathtt{IT},\mathtt{RT},\mathtt{TT},\mathtt{CT},\mathtt{e}) : \eta$ *and* $\mathtt{e} \rightarrow^\star \mathtt{e}'$ *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* $(\mathtt{I})\mathtt{new\ C}(\bar{\mathtt{e}})$ *where* $\mathtt{C} \not<: \mathtt{I}$.

A formulation of traits in a JAVA-like setting has to support the type-checking of traits in isolation from the classes or traits that use them, so that it is possible to type-check a method defined in a trait only once (instead of having to type-check it in every class or trait using that trait). The FRTJ type system supports the above property through a suitable combination of nominal and structural typing. Within a basic trait expression, the uses of method parameters are type-checked according to the nominal notion of typing defined by the interface hierarchy, while the uses of the this pseudovariable are type-checked according to a structural notion of typing that takes into account the fields and methods *required* by the trait and the methods *provided* by the trait. The following theorem can be established by inspecting the FRTJ typing rules.

THEOREM 4.2 (FRTJ TYPE-CHECKING). *A program can be type-checked by type-checking only once its interfaces, record, traits, and classes.*

5. IMPLEMENTING SPL IN FRTJ

In this section, the methodology to implement SPL in FRTJ is presented. 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* PS is a declaration

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

where 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* PT is a map from product names P to product specifications PS. A SPL L is a 5-tuple $(\mathtt{IT},\mathtt{RT},\mathtt{TT},\mathtt{CT},\mathtt{PT})$ where 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 IT, RT, TT and CT are *well-formed*, i.e., the tables IT/RT/TT/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 the product 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 IT, RT, TT and CT containing exactly the entries for the interfaces, records, traits

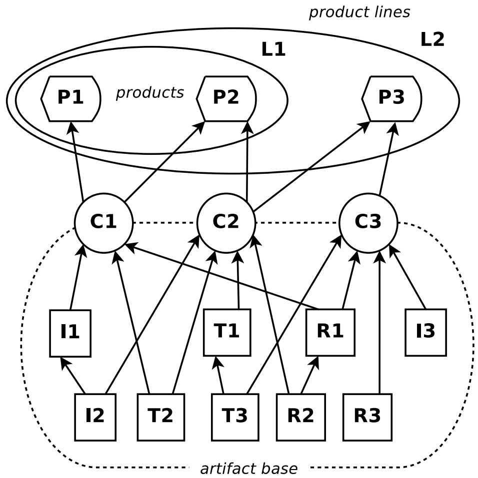

Figure 3: Relation between Artifacts, Products and SPL

and classes reachable from the classes contained the specification of P. We assume that each product specification `product P uses` $\bar{C}$ of PT is *well-formed*, that is: $\bar{C}$ contains exactly classes in CT_P.

Figure 3 depicts the relations between a SPL artifact base, the products and the SPL. On the lowest level, the traits T_1, T_2, T_3, the records R_1, R_2, R_3 and interfaces I_1, I_2, I_3 are used to built classes C_1, C_2, C_3 constituting the SPL artifact base. The classes are then used to built the products P_1, P_2 and P_3. The products are contained in two different SPL L_1 and L_2.

EXAMPLE 5.1. *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* (IT,RT,TT,CT,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*

Using the type system of FRTJ introduced in Section 4, we define type safety of a product line. We write $\vdash$ L OK, to be read: "the SPL L is well-typed", i.e., the code of every product P in L is well-typed. In most approaches (with the exception of, e.g., [30, 13]), the only way to verify that all the products of a SPL are type-safe is to generate and type-check all products individually. As a consequence of Theorem 4.2, in FRTJ, the type safety of a SPL can be verified without type-checking all its products individually, since it is enough to type-check only once each artifact in the artifact base.

THEOREM 5.2 (FRTJ SPL TYPE-CHECKING). *A SPL* L *can be type-checked by type-checking only once its interfaces, records, traits and classes.*

The formalization of SPL and the FRTJ calculus easily allow extending 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 $L' = (IT', RT', TT', CT', PT')$ is an *extension* of the SPL $L = (IT, RT, TT, CT, PT)$ if L has been obtained from L by adding interfaces, records, traits, classes and products (that is if $IT \subseteq IT'$, $RT \subseteq RT'$, $TT \subseteq TT'$, $CT \subseteq CT'$ and $PT \subseteq PT'$ hold). In Figure 3, the SPL L_2 is an extension of SPL L_1.

EXAMPLE 5.3. *The SPL* `BankLine'`*, described by the feature model in Figure 1* ***with*** *the* Sync *feature, extends the SPL* `BankLine` *of Example 5.1. 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 4.2 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 5.4 (FRTJ SPL EXTENSION TYPE-CHECKING). *Let the SPL* $L' = (IT', RT', TT', CT', PT')$ *be an extension of the SPL* $L = (IT, RT, TT, CT, PT)$. *If* L *has been already type-checked (so that the typings of all its artifacts are available), then the products in* $PT' - PT$ *can be type-checked without type-checking the artifacts of* L *and by type-checking only once the interfaces, records, traits, classes in* $IT' - IT$, $RT' - RT$, $TT' - TT$, $CT' - CT$, *respectively.*

6. RELATED WORK

Traits are well suited for designing libraries and enable clean design and reuse which has been shown using SMALLTALK/SQUEAK (see, e.g., [8, 11]). Recently, [6] 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.

The approaches to implementing the variability of SPL in the object-oriented paradigm can be classified into two main directions [18]. First, *annotative approaches*, such as conditional compilation, frames [4] and COLORED FEATHERWEIGHT JAVA (CFJ) [16], 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 SPL in the object-oriented paradigm use a variety of mechanisms, such as aspects [17], mixins [28], or features modules in the AHEAD framework [5]. In [22], product line variability is implemented in SCALA [25] using traits that are realized by mixin-based inheritance. The compositional approaches closest to FRTJ are FEATHERWEIGHT FEATURE JAVA (FFJ) [3] and LIGHTWEIGHT FEATURE JAVA (LFJ) [13]. Both calculi aim at a formalization of feature-based product composition with static guarantees.

FFJ and LFJ use specific linguistic constructs to implement features according to the feature-oriented paradigm [5]. A feature can introduce new classes and refine existing ones. The ordering in

which features are composed is restricted. A feature that refines a class can be added only after the class to be refined has been introduced. Refinement of classes is unavoidable, since the class hierarchy may have to be changed radically for implementing product variability. While refining a class does not change its name, its definition may change completely (even its superclass can be altered). Therefore, class refinement can break code in client classes built before the refinement. The case of a class that is refined by adding new fields is particularly interesting. Both FFJ and LFJ propose a way to initialize fields that are added to a class by refinement: (1) FFJ requires that all fields are initialized by a single constructor call with the values to be assigned to the fields as arguments (as in FJ). Newly added fields are initialized by ensuring that, whenever a superseded constructor in a client class built before the refinement is invoked, the additional fields are initialized to default values. (2) LFJ initializes newly added fields by a default constructor without arguments associated to each class that assigns default values (as in JAVA), and relies on assignment operations to set the fields properly. Both ways have the subtle drawback that, when a class refinement adds new fields, the code in client classes built before the refinement still type-checks, even if (due to a faulty SPL implementation) no code for the proper initialization of the new fields is inserted. In the presented approach, a class name refers to the same definition entity in all the products. If (due to faults in the SPL implementation) the code of a product invokes the constructor of a class not listed in the product specification, the error is automatically detected during type-checking assuming the well-formedness of the product table.

FFJ has a type system to check single product specifications, while LFJ supports the type-checking of a complete SPL. LFJ introduces a *constraint-based type system* (similar to the one in [2]) that supports the type-checking of feature modules in isolation. The type safety of a SPL can be verified by checking the validity of a generated propositional formula expressing its type safety. The FRTJ type system ensures that a SPL is type-safe by type-checking the artifacts in the artifact base only once. Furthermore, it allows type-checking of an extension of a (type-safe) SPL just by considering only the newly added parts.

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 re-use provided by records and traits is higher than the re-use 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. An extended version of this paper is available as [7]. A prototypical implementation of a language based on the FRTJ calculus is currently under development.

Our linguistic constructs which are lower-level than standard OO mechanisms can be used to introduce derived linguistic concepts in order to reduce the amount of code to write. For future work, we plan to investigate the possibility of adding a feature module construct (like the one of LFJ [13]) to FRTJ in order to lift the reuse potential beyond class level. 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.

Acknowledgements. We are grateful to Viviana Bono for initial collaboration on the subject of this paper and thank the OOPS 2010 referees and the referees of an earlier version of this paper for insightful comments and pointers to related work.

8. 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 *Proc. of POPL*, pages 26–37. ACM, 2005.

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

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

[5] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *In Proc. of ICSE '03*, pages 187–197, 2003.

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

[7] L. Bettini, F. Damiani, and I. Schaefer. Implementing SPL using Traits. Technical report, Dipartimento di Informatica, Università di Torino, 2009. Available at *http://www.di.unito.it/~damiani/papers/isplurat.pdf*.

[8] A. P. Black, N. Schärli, and S. Ducasse. Applying traits to the smalltalk collection classes. In *Proc. of OOPSLA '03*, pages 47–64. ACM, 2003.

[9] V. Bono, F. Damiani, and E. Giachino. Separating Type, Behavior, and State to Achieve Very Fine-grained Reuse. In *Electronic proceedings of FTfJP*, 2007.

[10] 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.

[11] D. Cassou, S. Ducasse, and R. Wuyts. Redesigning with traits: the nile stream trait-based library. In *Proc. of ICDL '07*, pages 50–75. ACM, 2007.

[12] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley Longman, 2001.

[13] B. Delaware, W. Cook, and D. Batory. A Machine-Checked Model of Safe Composition. In *Proc. of FOAL*, pages 31–35. ACM, 2009.

[14] 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.

[15] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.

[16] C. Kästner and S. Apel. Type-Checking Software Product Lines - A Formal Approach. In *Automated Softwware Engineering (ASE)*, pages 258–267, 2008.

[17] C. Kästner, S. Apel, and D. S. Batory. A Case Study Implementing Features Using AspectJ. In *SPLC*, pages 223–232, 2007.

[18] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *In Proc. of ICSE '08*, pages 311–320, 2008.

[19] G. Lagorio, M. Servetto, and E. Zucca. Featherweight Jigsaw - A minimal core calculus for modular composition of classes. In *ECOOP 2009*, LNCS 5653, pages 244–268. Springer, 2009.

[20] G. Lagorio, M. Servetto, and E. Zucca. Flattening versus direct semantics for Featherweight Jigsaw. In *Proc. of FOOL*, 2009.

[21] L. Liquori and A. Spiwack. FeatherTrait: A Modest Extension of Featherweight Java. *ACM TOPLAS*, 30(2), 2008.

[22] R. E. Lopez-Herrejon, D. S. Batory, and W. R. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *Proc. of ECOOP*, pages 169–194, 2005.

[23] L. Mikhajlov and E. Sekerinski. A Study of the Fragile Base Class Problem. In *Proc. ECOOP*, number 1445 in LNCS, pages 355–383. Springer, 1998.

[24] O. Nierstrasz, S. Ducasse, and N. Schärli. Flattening traits. *JOT (www.jot.fm)*, 5(4):129–148, 2006.

[25] M. Odersky. The Scala Language Specification, version 2.4. Technical report, Programming Methods Laboratory, EPFL, 2007.

[26] J. Reppy and A. Turon. Metaprogramming with traits. In *ECOOP 2007*, volume 4609 of *LNCS*, pages 373–398. Springer, 2007.

[27] 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.

[28] 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.

[29] C. Smith and S. Drossopoulou. *Chai*: Traits for Java-like languages. In *ECOOP'05*, LNCS 3586, pages 453–478. Springer, 2005.

[30] S. Thaker, D. S. Batory, D. Kitchin, and W. R. Cook. Safe Composition of Product Lines. In *Proc. of GPCE*, pages 95–104. ACM, 2007.

[31] D. Ungar, C. Chambers, B.-W. Chang, and U. Hölzle. Organizing Programs Without Classes. *Lisp and Symbolic Computation*, 4(3):223–242, July 1991.