Verifying traits: an incremental proof system for fine-grained reuse

Verifying Traits: An Incremental Proof System for Fine-Grained Reuse †

Ferruccio Damiani[a] and Johan Dovland[b] and Einar Broch Johnsen[b] and Ina Schaefer[c]

[a] Università di Torino, Dipartimento di Informatica, Italy
[b] Department of Informatics, University of Oslo, Norway
[c] Institute for Software Engineering, Technische Universität Braunschweig, Germany

Abstract. Traits have been proposed as a more flexible mechanism than class inheritance for structuring code in object-oriented programming, to achieve fine-grained code reuse. A trait originally developed for one purpose can be adapted and reused in a completely different context. Formalizations of traits have been extensively studied, and implementations of traits have started to appear in programming languages. So far, work on formally establishing properties of trait-based programs has mostly concentrated on type systems. This paper presents the first deductive proof system for a trait-based object-oriented language. If a specification of a trait can be given a priori, covering all actual usage of that trait, our proof system is modular as each trait is analyzed only once. However, imposing such a restriction may in many cases unnecessarily limit traits as a mechanism for flexible code reuse. In order to reflect the flexible reuse potential of traits, our proof system additionally allows new specifications to be added to a trait in an *incremental* way which does not violate established proofs. We formalize and show the soundness of the proof system.

Keywords: Traits, Object Orientation, Program Verification, Proof Systems, Incremental Reasoning

1. Introduction

In object-oriented languages with class inheritance, classes traditionally have three competing roles as generators of objects, as types for objects, and as units of reuse. In contrast, *traits* are pure units of behavior, designed for flexible, fine-grained reuse [SDNB03, DNS+06]. A trait contains a set of methods which is completely independent from any class hierarchy. Thus, the common methods of a set of classes can be factored into a trait. Traits can be composed in an arbitrary order. The resulting composite unit (which can be a class or another trait) has complete control over the conflicts which may arise in the composition, and must solve these conflicts explicitly. A trait which was developed for a particular purpose may later be adapted and reused in a completely different context. This way, traits achieve a very attractive level of code reuse, but this flexibility can lead to potentially undesired or conflicting program behavior.

† The authors of this paper are listed in alphabetical order. This work has been partially supported by the Deutsche Forschungsgemeinschaft (SCHA 1635/2-1), the Italian MIUR (PRIN 2008 DISCO), the German-Italian University Centre (Vigoni program), and the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (`http://www.hats-project.eu`).
Correspondence and offprint requests to: Einar Broch Johnsen, Department of Informatics, University of Oslo, PO Box 1080 Blindern, N-0316 Oslo, Norway. *Email:* `einarj@ifi.uio.no`.

Since their formulation in SMALLTALK-like languages [SDNB03, DNS+06, BDNW08, BDN+09], various formulations of traits have been studied for inheritance-based JAVA-like languages (e.g., [SD05, NDS06, RT07, BDG07, BDG08, LS08b, LS08a, BDSS10, BDSS13]). The recent programming language FORTRESS [SAC+11] (which has no class-based inheritance) has a trait construct, while the 'trait' construct of SCALA [OSV10] is a form of mixin, i.e., a subclass parametrized over its superclass (e.g., [BC90, LM96, FKF98, ALZ03]). Research on ensuring properties of trait-based programs has so far mostly considered type systems (e.g., [RT06, SD05, RT07, BDG08, LS08a, BCD12, BDD+10, BDGS13]). These approaches establish that the composed program is type correct; i.e., all required fields and methods are present with the appropriate types.

This paper presents a deductive proof system for verifying behavioral properties of trait-based programs. Flexible, fine-grained code reuse motivates the development of traits as a mechanism for code structuring, but this flexibility poses challenges when developing a corresponding proof system: a high degree of code reuse in a code structuring mechanism typically limits the degree of reasoning reuse which can be supported by the proof system. Ideally, when traits are composed into a class, the trait specifications should already provide enough information to ensure that the interface contracts of that class hold. This scenario would result in a purely compositional proof system in which the actual usage of a trait always corresponds to its originally intended usage, as captured by its original specification. These specifications can be established by modular reasoning; i.e., it is sufficient to analyze each trait once during the verification process. In practice, a priori imposing a fixed specification on a trait may overly restrict the flexibility of code reuse in that trait, which goes against the original motivation for traits.

The challenge in developing a deductive proof system for trait-based programs is to support the *flexibility* offered by traits as a code reuse mechanism, while providing an *incremental* and *compositional* reasoning system. Without restricting the flexibility of traits as a code reuse mechanism, a trait cannot be fully verified independently of its context of composition. In order to align our proof system with this flexibility, traits will be associated with *sets of possible specifications*, and the applicable specifications of a trait depend on its context of composition. By incremental reasoning, new specifications may be added *incrementally* to a trait without violating previous specifications and proofs. When traits are composed, the specifications of the composed traits are selected from the compatible specifications of the constituent traits. Hence, our proof system subsumes modularity and supports modular reasoning for traits when applicable, but extends this modularity to incremental reasoning when required for flexible trait reuse. In particular, the incremental approach supports a gradual extension of existing trait libraries with specifications, driven by the verification of programs using the library.

In order to focus the paper on the particular challenges proposed by trait-based programs, the proof system is presented for a kernel of TRAITRECORDJ [BDSS13, Tra11], a trait-based JAVA-like language with a prototype implementation. The proof system presented in this paper can be used to guarantee that programs obtained through the flexible adaptation and composition of traits satisfy critical requirements, by reasoning modularly and incrementally about traits, trait adaptation, and composition. The analysis of trait-based programs is formalized as an inference system which tracks specification sets for traits, when traits are modified and composed. This inference system adapts previous work on lazy behavioral subtyping [DJOS10], which developed an incremental inference system for late bound method calls by separating the required and provided behavior of methods, to trait modification and composition. The inference system presented in this paper does not depend on a particular program logic. For simplicity, we use a Hoare-style notation to specify the pre- and postconditions of method definitions in terms of proof outlines and do not consider, e.g., class or trait invariants. The main approach of our proof system has previously been presented in a short paper at FTfJP 2011 [DDJS11]. This paper extends [DDJS11] with more examples and explanations, details of the formalization, and the proofs of the main results. As far as we know, no other deductive proof system for trait-based languages has been proposed so far.

The paper is structured as follows. Section 2 gives a high-level introduction to our approach to verifying trait-based programs as an incremental reasoning process. Section 3 introduces the kernel trait language used in this paper and Section 4 presents a specification notation for traits. Sections 5 and 6 introduce the two aspects of the proposed proof system for the considered kernel trait language. Section 5 discusses how the proof system is used to verify basic and composed trait expressions and Section 6 explains how to verify interface contracts for classes. The formal inference system for program analysis is defined in Section 7, which also shows soundness for the proof system. Section 8 illustrates by example how the proof system may be used. Related work is discussed in Section 9. Section 10 concludes the paper and discusses future work. The paper includes three appendixes. Appendix A contains technical definitions omitted in the main paper, Appendix B contains the proofs of the soundness theorem and Appendix C details the verification of the example from Section 8.

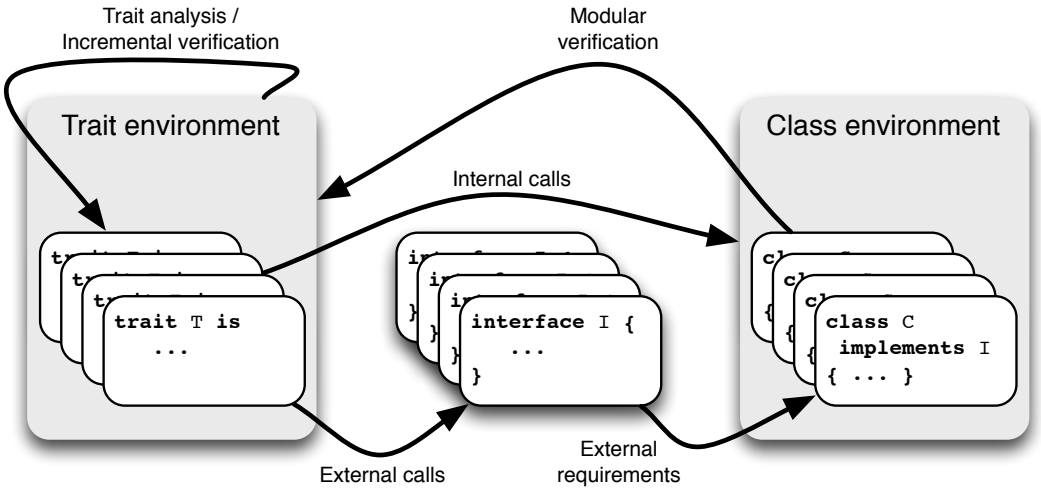

Fig. 1. The workflow for verifying trait-based programs, starting by the external requirements imposed on classes by their behavioral interfaces.

2. Verifying Trait-based Programs as an Incremental Reasoning Process

The main goal of the verification process for trait-based programs is to show that the classes that are built from the traits implement the contracts of their declared interfaces. This section gives an informal introduction to this verification process, which consists of two parts: trait analysis and class analysis. Section 7 shows that the overall verification process meets its goal through the soundness of the formalized proof system (Theorem 7.4).

Figure 1 illustrates the envisaged workflow for verifying trait-based programs using our verification process. We assume given a library of traits, annotated with specifications (possibly without any non-trivial specifications). These trait specifications are stored in the *trait environment*. The specifications of a trait contain a pre-/post-condition pair for each method, denoting the method's guarantee. This guarantee holds if the other methods called by this method satisfy certain properties which are contained as requirements in the method specification. While requirements to *external calls* can be checked via the behavioral interfaces, the guarantees of internal calls depend on how the traits are assembled into classes; *internal calls* are checked with respect to the class environment. During the trait analysis, the specifications of methods in composed trait expressions are derived from the specifications of methods in the basic traits, depending on trait modifiers and composition. The specifications of the methods in the basic traits have to be verified by a suitable proof system before they are stored in the trait environment during the trait analysis. In this setting, the trait environment plays the role of a cache of already established specifications that can be reused if applicable during the analysis of classes. For a trait library, the trait environment can be modularly reused and incrementally extended during the analysis of different programs using the library.

During the class analysis, the verification of a program comprising a set of classes built from traits is driven by the *external requirements* to the classes of the program. These external requirements are specified in the behavioral interfaces which are implemented by the classes of the program. The external requirements to the methods of a class are decomposed into requirements on methods in trait expressions used to build the class. The *class environment* stores the information about the method specifications used to verify the trait expressions in order to retrieve those when verifying the requirements of other methods. When an external requirement is imposed on a method in a trait expression during the class analysis, there are three cases:

1. *Modular verification:* The specifications already contained in the trait environment suffice to prove the external requirements imposed on the method. We obtain *specification reuse* by simply showing that the specifications stored in the trait environment imply the external requirement.
2. *Incremental verification:* The specifications already contained in the trait environment do not suffice to prove the external requirements. The user can add new specifications to the traits, which need to be verified. This verification triggers new trait analysis operations in which external calls are verified based on behavioral interfaces and internal calls based on methods on other traits. During this verification process, we can return to the modular case in which

specifications follow from previous specifications or remain in the incremental case which can repeatedly trigger new trait analysis operations until the analysis is complete.

3. *Non-verifiable case:* The verification process for the program fails if we cannot find provable specifications to be added to the trait environment which resolve the external requirements on the classes.

The approach allows program verification to interact with a trait library without assuming that the library already contains specifications: When verifying the first program built from a set of traits, we can start with an empty trait environment. The verification process will then enter the incremental case and derive appropriate annotations for the methods as required by the program and store them in the trait environment. When another program is built by reusing already verified traits, we can have the modular case such that we are able to reuse the previously established specifications or again enter the incremental case and derive new specifications. In this way, while verifying a set of programs, we stepwise obtain all necessary specifications for the traits that are contained. In practice, we expect that programs will often use a trait in the same way so the potential for specification reuse is good. However, the proposed approach also caters for programs that use some traits in innovative ways, supporting fine-grained specification reuse.

3. A Kernel Language for Trait-Based Programs

For the purposes of this paper we consider a kernel of the trait-based programming language TRAITRECORDJ [BDSS13, Tra11] which highlights the specific features of trait-based programming from a reasoning perspective. In particular, TRAITRECORDJ completely separates the three traditional roles of classes as types, units of code reuse, and object generators. Interfaces and primitive types are the only source language types; fields and methods are typed by source language types, and subtyping reflects the extends-relation in interface declarations. Traits only play the role of units of code reuse and are not types. Class-based inheritance is not present, so classes only play the role of object generators.

A trait in TRAITRECORDJ consists of *provided methods*, which are the methods defined in the trait; *required methods*, which parametrize the behavior of auxiliary methods assumed to be available in a class using the trait; and *required fields*, which are similarly assumed to be available in a class using the trait. The required fields of a trait can be directly accessed in the body of the provided methods of the trait, and the required methods can be called internally in the trait's provided methods. Traits are building blocks which can be used to compose classes and other traits by means of a trait summation operation and a suite of trait alteration operations. Since traits do not introduce any state, a class assembled from traits has to provide the required fields of its constituent traits. The kernel language represents a "normal form" syntax which simplifies the analysis proposed in this paper, since it ensures that trait summation happens as late as possible in a trait composition.

3.1. Kernel Language Syntax

The syntax of the kernel language is given in Figure 2. A program `P` consists of (lists of) interface declarations `ID`, trait declarations `TD`, and class declarations `CD`. Following [IPW01] we use the overline notation for (possibly empty) sequences. In TRAITRECORDJ, interface declarations `ID`, method headers `H`, field declarations `F`, and method declarations `M` have a similar syntax as in JAVA (but ignoring, e.g., visibility modifiers and exceptions). For simplicity, in the kernel language the syntax of methods is restricted. For instance, a method may only contain a single **`return`** statement which must be the last statement in the body of the method, and method parameters are not assignable. These language features are well-understood from the literature (e.g., [Hoa69, Apt81, AdBO09]) and do not impose any new difficulties for our proof system. In the sequel, we will use the notation *body*(`M`) to denote the body of a method declaration `M`. We syntactically distinguish *internal* method calls $\texttt{m}(\overline{\texttt{e}})$ from *external* method calls $\texttt{v.m}(\overline{\texttt{e}})$ (where the variable `v` is typed by an interface).

A trait declaration `TD` binds a trait name `T` to a trait expression. Traits are either basic or composed. A *basic trait* binds a basic trait name `Tb` to a basic trait expression `BTE`, whereas a *composed trait* binds a composed trait name `Tc` to a composed trait expression `CTE`. Only basic traits can be adapted by means of alteration operations, which explains the normal form restriction of the kernel language, and assembled into composed traits.

A *basic trait expression* `BTE`, written $\{\overline{\texttt{F}}; \overline{\texttt{H}}; \overline{\texttt{M}}\}$, *provides* the methods $\overline{\texttt{M}}$ and declares the types of the *required* fields $\overline{\texttt{F}}$ and methods $\overline{\texttt{H}}$ (which can be directly accessed in the bodies of the methods $\overline{\texttt{M}}$). A *trait alteration expression* `TAE` applies trait alteration operations `ao` to a basic trait `Tb`. 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; in particular, when a recursive method is aliased, its recursive invocation refers to the original method. The operation

```
P    ::= ID* TD* CD*                                          program

ID   ::= interface I extends I* { H*; }                       interface declaration
H    ::= [S | void] m (S* x*)                                 method header
S    ::= I | boolean | int                                    source language type
F    ::= S f                                                  field
M    ::= H { S* x*; s*; [return e;] }                         method
s    ::= v = rhs | m(e*) | v.m(e*) | if (e) {s*;}             statement
v    ::= f | x                                                variable
rhs  ::= m(e*) | e.m(e*) | new C(e*) | e                      expressions with side effects
e    ::= v | this | null | true | false | ...                 expressions without side effects

TD   ::= trait Tb is BTE                                      basic trait declaration
       | trait Tc is CTE                                      composed trait declaration
T    ::= Tb | Tc                                              trait names

BTE  ::= {F*; H*; M*}                                         basic trait expression
CTE  ::= TAE | CTE + CTE                                      composed trait expression

TAE  ::= Tb ao*                                               trait alteration expression
ao   ::= [exclude m] | [m aliasAs m]                          trait alteration operation
       | [f renameTo f] | [m renameTo m]

CD   ::= class C implements I* by { F*; } and CTE             class declaration
```

Fig. 2. Kernel language syntax. The following naming convention is used in the syntax: I ∈ interface names; Tb ∈ basic trait names; Tc ∈ composed trait names; C ∈ *Cid* (the class names); f ∈ field names; m ∈ *Mid* (the method names); x ∈ method parameter or local variable names. We further denote by *Mtd* the set of method declarations and by *Label* the set of field and method names.

renameTo creates a new trait by renaming all occurrences of the name of a required field name or of a required or provided method name from an existing trait. A *composed trait expression* CTE is either a trait alteration expression TAE or the sum of two composed trait expressions. The *symmetric sum* + merges two traits to form a new trait and requires that the summed traits are disjoint; i.e., the summed traits cannot provide identically named methods but they may require identically named (and typed) fields or methods.

Classes in TRAITRECORDJ are assembled from a trait expression by providing the fields required by that trait expression. Thus, a class declaration CD binds a class name C to a set of interfaces $\overline{\texttt{I}}$, which specify the possible types for an instance of the class, fields $\overline{\texttt{F}}$, and a trait expression CTE. All fields are private. For simplicity, class constructors are omitted in the paper; each class C is assumed to have an implicit constructor that behaves like the JAVA constructor $\texttt{C}(\overline{\texttt{S}}\ \overline{\texttt{x}})\ \{\ \overline{\texttt{f}} = \overline{\texttt{x}}\ \}$, where $\overline{\texttt{S}}\ \overline{\texttt{f}}$ are all the fields of the class.

Example 3.1. As a running example in this paper, we consider the implementation of a bank account. The following trait TAcc provides the basic operations for inserting and withdrawing money:

```
trait TAcc is {              // no required fields
  boolean validate(int a);   // required method
  void update(int y);        // required method
  void deposit(int x) {update(x);}
  void withdraw(int id, int x){
    boolean v = validate(id);
    if (v) {update(-x);}
  }
}
```

A basic account CAcc may then be defined as follows, where the additional trait TAux defines the auxiliary methods required by TAcc:

```
interface IAcc {
  void deposit(int x);
```

```
  void withdraw(int id, int x);
}

trait TAux is {
  int  bal; int owner;        // required fields
  void update(int y) {bal = bal + y;}
  boolean validate(int id) {return (id == owner);}
}

class CAcc implements IAcc by {int bal; int owner;} and TAcc + TAux
```

Since traits define flexible units for code reuse, `TAcc` may be combined with other traits to define different account behavior. For example, the class `CFeeAcc` charges an additional fee whenever the balance is reduced. We define another interface `IFeeAccount` in order to be able to associate different behavioral specifications later. The class `CFeeAcc` is composed from the traits `TAcc`, `TFee`, and `TAux` where the method `update` is renamed to `basicUpd`.

```
interface IFeeAcc {
  void deposit(int x);
  void withdraw(int id, int x);
}

trait TFee is {
  int  fee; int bal;          // required fields
  void basicUpd(int y);       // required method
  void update(int y) {
    basicUpd(y); if (y<0) {bal = bal - fee;}
  }
}

class CFeeAcc implements IFeeAcc
  by  {int bal; int owner; int fee;}
  and TAcc + TFee + (TAux[update renameTo basicUpd])
```

The public methods of an object are those listed in the interfaces implemented by its class; the other methods and fields are private to the object. For instance, the only public members of classes `CAcc` and `CFeeAcc` are the methods `deposit` and `withdraw`.

3.2. Kernel Language Semantics

The semantics of a class composed from traits is defined through the *flattening principle* [DNS+06] (see also [NDS06, LSZ09]), which states that the semantics of a method introduced in a class through a trait should be identical to the semantics of the same method defined directly within a class. A flattening function defines the semantics of TRAITRECORDJ by translating TRAITRECORDJ class declarations to JAVA class declarations, and a trait expression to a sequence of method declarations. TRAITRECORDJ interfaces are literally JAVA interfaces and need no translation.

Let a TRAITRECORDJ program be represented by an interface table `IT`, a trait table `TT`, and a class table `CT`, which map interface, trait, and class names to interface, trait, and class declarations, respectively. For simplicity, we assume fixed, global tables `IT`, `TT`, and `CT`. The flattening function $[\![\cdot]\!]$, defined in Figure 3, maps a TRAITRECORDJ class declaration to a JAVA class declaration, and a trait expression to a sequence of method declarations. (The implicit class constructor of TRAITRECORDJ is unaltered by the flattening function, and omitted in Figure 3.) We denote by $[\![\texttt{CD}]\!]$ the flattened version of a class declaration `CD`, and by $[\![\texttt{TE}]\!]$ the method declarations which result from flattening the (basic or composed) trait expression `TE`.

The flattening function uses some auxiliary functions, which are defined in Figure 4. Given a set $\overline{\texttt{M}}$ of method definitions, let $\texttt{m} \in \overline{\texttt{M}}$ denote that `m` is defined in $\overline{\texttt{M}}$, and let $\overline{\texttt{M}}(\texttt{m})$ return the definition of `m` in $\overline{\texttt{M}}$. Function *rem* : $Set[Mtd] \times Mid \to Set[Mtd]$ takes a set of method definitions and a method name `m`, and returns the set without the definition of `m`. Function *ren* : $Mtd \times Mid \times Mid \to Set[Mtd]$ performs method renaming. The function *rep* : $Set[Mtd] \times Label \times Label \to Set[Mtd]$ is defined such that:

$$[\![\textbf{class}\ \mathtt{C}\ \textbf{implements}\ \overline{\mathtt{I}}\ \textbf{by}\ \{\ \overline{\mathtt{F}};\ \}\ \textbf{and}\ \mathtt{CTE}]\!] \triangleq \textbf{class}\ \mathtt{C}\ \textbf{implements}\ \overline{\mathtt{I}}\ \{\ \overline{\mathtt{F}};\ [\![\mathtt{CTE}]\!]\ \}$$

$$\begin{array}{rcl}
[\![\{\ \overline{\mathtt{F}};\ \overline{\mathtt{H}};\ \overline{\mathtt{M}}\ \}]\!] & \triangleq & \overline{\mathtt{M}} \\
[\![\mathtt{T}]\!] & \triangleq & [\![\mathtt{TE}]\!] \quad \textbf{if}\ \mathtt{TT(T)} = \textbf{trait}\ \mathtt{T}\ \textbf{is}\ \mathtt{TE} \\
[\![\mathtt{CTE}_1 + \mathtt{CTE}_2]\!] & \triangleq & [\![\mathtt{CTE}_1]\!] \cdot [\![\mathtt{CTE}_2]\!] \\
[\![\mathtt{TAE}[\textbf{exclude}\ \mathtt{m}]]\!] & \triangleq & \mathit{rem}([\![\mathtt{TAE}]\!], \mathtt{m}) \\
[\![\mathtt{TAE}[\mathtt{m}\ \textbf{aliasAs}\ \mathtt{m}']]\!] & \triangleq & \begin{cases} [\![\mathtt{TAE}]\!] \cdot \mathit{ren}([\![\mathtt{TAE}]\!](\mathtt{m}), \mathtt{m}, \mathtt{m}') & \textbf{if}\ \mathtt{m} \in [\![\mathtt{TAE}]\!] \\ [\![\mathtt{TAE}]\!] & \textbf{otherwise} \end{cases} \\
[\![\mathtt{TAE}[\mathtt{f}\ \textbf{renameTo}\ \mathtt{f}']]\!] & \triangleq & \mathit{rep}([\![\mathtt{TAE}]\!], \mathtt{f}, \mathtt{f}') \\
[\![\mathtt{TAE}[\mathtt{m}\ \textbf{renameTo}\ \mathtt{m}']]\!] & \triangleq & \mathit{rep}([\![\mathtt{TAE}]\!], \mathtt{m}, \mathtt{m}')
\end{array}$$

Fig. 3. Flattening TRAITRECORDJ to JAVA (where TE denotes either a basic trait expression BTE or a composed trait expression CTE).

$$\begin{array}{l}
\mathit{rem}(\emptyset, \mathtt{m}) \triangleq \emptyset \\
\mathit{rem}(\mathtt{I}\ \mathtt{n}(\overline{\mathtt{I}}\ \overline{\mathtt{x}})\{t\}\ \overline{\mathtt{M}}, \mathtt{m}) \triangleq \begin{cases} \overline{\mathtt{M}} & \textbf{if}\ \mathtt{m} = \mathtt{n} \\ \mathtt{I}\ \mathtt{n}(\overline{\mathtt{I}}\ \overline{\mathtt{x}})\{t\}\ \mathit{rem}(\overline{\mathtt{M}}, \mathtt{m}) & \textbf{otherwise} \end{cases} \\
\\
\mathit{ren}(\mathtt{I}\ \mathtt{n}(\overline{\mathtt{I}}\ \overline{\mathtt{x}})\{t\}, \mathtt{m}, \mathtt{m}') \triangleq \mathtt{I}\ \mathtt{n}[\mathtt{m}'/\mathtt{m}](\overline{\mathtt{I}}\ \overline{\mathtt{x}})\{t\} \\
\\
\mathit{rep}(\emptyset, \mathtt{l}, \mathtt{l}') \triangleq \emptyset \\
\mathit{rep}(\mathtt{I}\ \mathtt{n}(\overline{\mathtt{I}}\ \overline{\mathtt{x}})\{t\}\ \overline{\mathtt{M}}, \mathtt{l}, \mathtt{l}') \triangleq \mathtt{I}\ \mathtt{n}(\overline{\mathtt{I}}\ \overline{\mathtt{x}})\{t\}[\mathtt{l}'/\mathtt{l}](\overline{\mathtt{I}}\ \overline{\mathtt{x}})\{t[\mathtt{l}'/\mathtt{l}]\}\ \mathit{rep}(\overline{\mathtt{M}}, \mathtt{l}, \mathtt{l}')
\end{array}$$

Fig. 4. Definitions of auxiliary flattening functions, where: l and $\mathtt{l}'$ range over field and method names; $\mathtt{n}[\mathtt{l}'/\mathtt{l}]$ denotes $\mathtt{l}'$ if $\mathtt{n} = \mathtt{l}'$ and l otherwise; and $t[\mathtt{l}'/\mathtt{l}]$ denotes the substitution of all occurrences of field l in t by $\mathtt{l}'$ and the substitution of all internal calls of method l in t by internal calls of method $\mathtt{l}'$.

- if l is a field name, $\mathit{rep}(\overline{\mathtt{M}}, \mathtt{l}, \mathtt{l}')$ replaces all occurrences of l in the bodies of the methods $\overline{\mathtt{M}}$ by $\mathtt{l}'$; and
- if l is a method name, $\mathit{rep}(\overline{\mathtt{M}}, \mathtt{l}, \mathtt{l}')$ replaces all occurrences of l in the headers of the methods $\overline{\mathtt{M}}$ by $\mathtt{l}'$ and replaces all the internal calls of l in the bodies of the methods $\overline{\mathtt{M}}$ by internal calls of $\mathtt{l}'$.

Example 3.2. The flattening of the class CFeeAcc, which is introduced in Example 3.1 above, is as follows (the implicit TRAITRECORDJ constructor is omitted).

```
class CFeeAcc implements IFeeAcc {
  int bal; int owner; int fee;
  void deposit(int x) {update(x);}
  void withdraw(int id, int x){
    boolean v = validate(id); if (v) {update(-x);}
  }
  void basicUpd(int y) {bal = bal + y;}
  boolean validate(int id) {return (id == owner);}
  void update(int y) {
    basicUpd(y); if (y<0) {bal = bal - fee;}
  }
}
```

Example 3.3. This example illustrates the difference between the **aliasAs** and **rename** operations on traits. Consider the following trait TFactorial defining a method factorial that computes the factorial of a natural number by means of an auxiliary method fAux:

```
trait TFactorial =
{ int factorial(int n) { int r = -1; if (n >= 0) r = fAux(n); return r; }
  int fAux(int n) { int r = 1; if (n >= 2) r = n * fAux(n-1); return r; } }
```

The flattening of $[\![\mathtt{TFactorial}[\mathtt{fAux}\ \textbf{aliasAs}\ \mathtt{fact}]]\!]$ is the following sequence of three methods (the first two methods are exactly those provided by the trait TFactorial):

```
int factorial(int n) { int r = -1; if (n >= 0) r = fAux(n); return r; }
int fAux(int n) { int r = 1; if (n >= 2) r = n * fAux(n-1); return r; }
int fact(int n) { int r = 1; if (n >= 2) r = n * fAux(n-1); return r; }
```

while the flattening of $[\![$`TFactorial[fAux renameTo fact]`$]\!]$ is the following sequence of two methods:

```
int factorial(int n) { int r = -1; if (n >= 0) r = fact(n); return r; }
int fact(int n) { int r = 1; if (n >= 2) r = n * fact(n-1); return r; }
```

The TRAITRECORDJ type system ensures that the flattening of a well-typed TRAITRECORDJ program is a well-typed JAVA program. It has been formalized for FEATHERWEIGHT RECORD-TRAIT JAVA (FRTJ) [BDS10, BDS09], a minimal core calculus for TRAITRECORDJ in the spirit of FEATHERWEIGHT JAVA [IPW01]. A type system for the kernel language considered in this paper would be a straightforward adaptation of the type system for FRTJ; in the sequel, it is assumed that programs are well-typed according to such a type system.

4. Specifying Trait-Based Programs

This section considers the specification of trait-based programs. The proof system developed in this paper does not depend on a particular program logic. Let *PL* be a (sound) program logic and let a, p, q range over assertions in the assertion language of *PL*. For simplicity in the presentation we use Hoare triples $\{p\}s\{q\}$ [Hoa69], with a standard partial correctness semantics [Apt81, AdBO09], adapted to the object-oriented setting; in particular, de Boer's technique using sequences in assertions addresses the issue of object creation [dB99]. Thus, the triple $\{p\}s\{q\}$ expresses that if s is executed in a state where the precondition p holds and the execution terminates, then the postcondition q holds after s has terminated. Let $A \vdash_{PL} \{p\}s\{q\}$ denote that $\{p\}s\{q\}$ is derivable from a (possibly empty) set of axioms A using the inference rules of *PL*. An *assertion pair* (p,q) is a pair of assertions such that p is a precondition and q a postcondition (for some sequence of program statements).

Entailment. The standard rule of consequence in Hoare Logic (e.g., [AdBO09]) is insufficient for dealing with sets of assertion pairs, which we will need to flexibly combine information from assertion pairs. We take a relational approach to entailment, and let q^o denote the assertion q in which all occurrences of fields f and method parameters x have been substituted by the corresponding fields f^o and local variables x^o, respectively, avoiding name capture. The assertion pair (p,q) is understood as an input/output relation $\forall \bar{z} \,.\, p \Rightarrow q^o$, where the formula q^o ensures that the postcondition applies to the values of fields and local variables in the post state, $\bar{z}$ are the logical variables in p and q, and the universal quantifier defines the scope of the logical variables (for further details on relational assertions, see e.g. [BvW98, HLL$^+$12]). The standard entailment relation is lifted to assertion pairs and to sets of assertion pairs, as follows [DJOS10]:

Definition 4.1 (Entailment). Let (p,q) and (r,t) be assertion pairs and let $\mathscr{U}$ and $\mathscr{V}$ denote the sets $\{(p_i,q_i) \mid 1 \le i \le n\}$ and $\{(r_i,t_i) \mid 1 \le i \le m\}$. *Entailment* is defined over assertion pairs and sets of assertion pairs by

1. $(p,q) \twoheadrightarrow (r,t) \triangleq (\forall \bar{z}_1 \,.\, p \Rightarrow q^o) \Rightarrow (\forall \bar{z}_2 \,.\, r \Rightarrow t^o)$,
   where $\bar{z}_1$ and $\bar{z}_2$ are the logical variables in (p,q) and (r,t), respectively.
2. $\mathscr{U} \twoheadrightarrow (r,t) \triangleq (\bigwedge_{1 \le i \le n} (\forall \bar{z}_i \,.\, p_i \Rightarrow q_i^o)) \Rightarrow (\forall \bar{z} \,.\, r \Rightarrow t^o)$.
3. $\mathscr{U} \twoheadrightarrow \mathscr{V} \triangleq \bigwedge_{1 \le i \le m} \mathscr{U} \twoheadrightarrow (r_i,s_i)$.

The relation $\mathscr{U} \twoheadrightarrow (r,t)$ corresponds to classic Hoare style reasoning, proving $\{r\}s\{t\}$ from $\{p_i\}s\{q_i\}$ for all $1 \le i \le n$, by means of the consequence and conjunction rules [AdBO09]. When proving entailment, program variables (input and output) are implicitly universally quantified. Furthermore, entailment is reflexive and transitive, and $\mathscr{V} \subseteq \mathscr{U}$ implies $\mathscr{U} \twoheadrightarrow \mathscr{V}$.

Behavioral Interfaces and Method Contracts. The (external) behavior of a program can be described in terms of behavioral interfaces which extend the interfaces of a program with method contracts. A *method contract* guarantees that a method has a certain behavior, captured by an assertion pair relating the prestate and the poststate of the method execution. In addition to describing the relation between the values of the method's formal parameters and its returned output value, a method's effect on the state can be described by means of so-called *model variables* (e.g., [HLL$^+$12]) in the interface. To avoid representation exposure, a *representation function* is used to relate the actual fields of a class

implementing the interface to the model variables of the interface. Logical variables, which are not model variables, are scoped within an assertion pair so they have the same value in the pre- and postcondition (this scope is formally captured in Definition 4.1 by quantification). In the syntax, an *annotated method header* extends a method header H as defined in Figure 2 with an assertion pair, and an *annotated interface declaration* extends an interface declaration ID with contracts for its exported methods. Formally, annotated method headers AH and annotated interface declarations AID have the syntax

$$\begin{array}{rcl} \texttt{AH} & ::= & \texttt{H} : (a,a) \\ \texttt{AID} & ::= & \textbf{interface}\ \texttt{I}\ \textbf{extends}\ \overline{\texttt{I}}\ \{\ \textbf{\textit{model:}}\ \overline{F}\ \overline{\texttt{AH}};\ \} \end{array}$$

where $\overline{F}$ are the model variables of the interface. To easily distinguish annotations used for behavioral reasoning from the code of the program, italics are used for annotations in the examples, for both model variables and assertions.

Example 4.2. To be concrete in the examples, we consider Boolean assertions a in an assertion language defined by

$$a ::= \textbf{\textit{this}} \mid \textbf{\textit{result}} \mid \textbf{\textit{null}} \mid v \mid z \mid op(\overline{a}).$$

In this assertion language, ***this*** denotes the current object, ***result*** the current method's return value, v a program variable, z a logical variable, and op an operation on data types. In particular, the equality of two assertions a_1 and a_2 is denoted $a_1 == a_2$. We now extend the interfaces IAcc and IFeeAcc of Example 3.1 with contracts as follows:

```
interface IAcc {
  model: int bal, int owner
  void deposit(int x) : (x > 0 ∧ bal == b0, bal = b0 + x);
  void withdraw(int id, int x) :
    (x > 0 ∧ bal == b0, (id == owner ⇒ bal == b0 − x) ∧ (id ≠ owner ⇒ bal == b0));
}

interface IFeeAcc {
  model: int bal, int owner, int fee
  void deposit(int x) : (x > 0 ∧ bal == b0, bal = b0 + x);
  void withdraw(int id, int x) :
    (x > 0 ∧ bal == b0, (id == owner ⇒ bal == b0 − x − fee) ∧ (id ≠ owner ⇒ bal == b0));
}
```

Logical variables are conventionally denoted by subscripts and all formal parameters are read-only. For instance, the initial value of *bal* is captured by the logical variable b_0 in $(bal == b_0, bal == b_0 + x)$, and x is the value of the formal parameter.

Specifying Basic Traits. For a basic trait, a *guarantee* for the behavior of each provided method is given by an assertion pair. Since the guarantee of a provided method m may crucially depend on the behavior of the methods *called* by m, the guarantee of m has an associated set of *requirements* on other methods; e.g., the requirement expressed by $\texttt{n} : (r,t)$ is that the method n must guarantee (r,t). The guarantee (p,q) of a method together with an associated set $\overline{\texttt{R}}$ of such requirements constitute a *method specification*, denoted $\langle (p,q), \overline{\texttt{R}} \rangle$.

Definition 4.3 (Validity of method specifications). A method specification $\langle (p,q), \overline{\texttt{R}} \rangle$ for a method m with body s is *valid* if the guarantee for s can be derived from the requirements $\overline{\texttt{R}}$ in the program logic *PL*; i.e., $\overline{\texttt{R}} \vdash_{PL} \{p\} s \{q\}$.

The validity of a specification $\langle (p,q), \overline{\texttt{R}} \rangle$ for a method m can be mechanically checked by providing a *proof outline* [OG76] for the method body of m. A proof outline typically annotates the method body with information which is difficult to infer but needed for a proof, such as loop invariants and method call annotations. We focus on the latter here, and assume that method calls in a proof outline for the body of m are annotated with the behavioral requirements on auxiliary methods which are needed by m to fulfill the guarantee (p,q). If external calls are checked with respect to their method contracts as given in behavioral interfaces, it is sufficient for the developer to annotate internal calls. Let $reqs(O)$ denote the set of behavioral requirements to internal method calls in a proof outline O, i.e., each annotated internal call $\{r\}$ n $\{t\}$ in O leads to a requirement $\texttt{n} : (r,t)$ in $reqs(O)$, and let $O \vdash_{PL} s : (p,q)$ denote that O is *a valid proof outline* for the method body s with the guarantee (p,q); i.e., $O \vdash_{PL} s : (p,q) \Rightarrow reqs(O) \vdash_{PL} \{p\}\ s\ \{q\}$. Generally, it is easier to (mechanically) check $O \vdash_{PL} s : (p,q)$ than to derive a proof of $reqs(O) \vdash_{PL} \{p\}\ s\ \{q\}$.

A trait is designed for flexible reuse. It can be difficult to specify the methods in the trait in a way which covers all possible future usages of the trait. In practice, there may be many possible guarantees for the provided methods

of a trait, depending on the context of use. These guarantees can have different associated proof outlines, which give rise to different requirements on the called methods. Thus, a provided method in a trait can have several specifications reflecting different usage contexts. The initial specifications reflect the originally intended usage of the method; further specifications may be added later if new ways of using the trait are discovered. If the initial specification happens to be sufficient for the later actual usage, this is the special case which *coincides with modular specification*. In general, an *annotated method* associates a list of method specifications to a method declaration and an *annotated basic trait* has annotated method declarations for its provided methods. Similarly, an *annotated class* provides a representation function rp_I for each interface I that it implements, which maps model variables f to assertions over the fields of the class. Formally, requirements R, specifications sp, annotated methods AM, annotated basic traits ABT, and annotated classes AC have the syntax

$$\begin{array}{rcl} \texttt{R} & ::= & \texttt{n} : (a,a) \\ \texttt{sp} & ::= & \langle (a,a), \overline{\texttt{R}} \rangle \\ \texttt{AM} & ::= & \texttt{M}\ \overline{\texttt{sp}} \\ \texttt{ABTE} & ::= & \{\overline{\texttt{F}}; \overline{\texttt{H}}; \overline{\texttt{AM}}\} \\ \texttt{ABT} & ::= & \textbf{trait}\ \texttt{Tb}\ \textbf{is}\ \texttt{ABTE} \\ \texttt{AC} & ::= & \textbf{class}\ \texttt{C}\ \textbf{implements}\ \overline{\texttt{I}}\ \textbf{by}\ \{\ \overline{rp_I}\ \overline{\texttt{F}};\ \}\ \textbf{and}\ \texttt{CTE} \end{array}$$

For ABTE we assume a basic trait declaration $\{\overline{\texttt{F}}; \overline{\texttt{H}}; \overline{\texttt{M}}\}$ such that each AM in $\overline{\texttt{AM}}$ extends a method declaration M in $\overline{\texttt{M}}$ with specifications. The notation introduced for methods can be extended to annotated methods as follows. For a set $\overline{\texttt{AM}}$ of annotated methods, let the function $mtds(\overline{\texttt{AM}})$ return the methods defined in $\overline{\texttt{AM}}$; i.e., $mtds(\overline{\texttt{AM}})$ can be defined by $mtds(\emptyset) \triangleq \emptyset$ and $mtds(\texttt{M}\ \overline{\texttt{sp}}\ \overline{\texttt{AM}}) \triangleq \texttt{M}\ mtds(\overline{\texttt{AM}})$. Then, $\texttt{m} \in \overline{\texttt{AM}}$ is defined by $\texttt{m} \in mtds(\overline{\texttt{AM}})$. Let $\overline{\texttt{AM}}(\texttt{m})$ return the annotated definition of m in $\overline{\texttt{AM}}$, so $\overline{\texttt{AM}}(\texttt{m}) = \texttt{AM}$ if $\texttt{m} \in \texttt{AM}$. The body of an annotated method can be accessed as an unannotated method, $body(\texttt{M}\ \overline{\texttt{sp}}) = body(\texttt{M})$. Furthermore, $specs(\texttt{M}\ \overline{\texttt{sp}}) \triangleq \overline{\texttt{sp}}$. The specification of methods is illustrated by the following example, where specifications are given in a style similar to JML [BCC+05] and Fresco [Wil91].

Example 4.4. Consider the withdraw method of trait TAcc in Example 3.1. This method may be given the two specifications below. For clarity, a specification $\langle (p,q), \overline{\texttt{R}} \rangle$ is here written ***guar:*** (p,q) ***req:*** $\overline{\texttt{R}}$, where the clause ***req:*** $\overline{\texttt{R}}$ is omitted if there are no requirements. The specifications are labelled w1 and w2.

```
trait TAcc is {              // no required fields
  boolean validate(int a); // required method
  void update(int y);       // required method
  void deposit(int x) {update(x);}
  void withdraw(int id, int x){
    boolean v = validate(id);
    if (v) {update(-x);}
  }
  // w1:
  guar: (bal == b0 ∧ id == owner, bal == b0 − x)
  req:  update : (bal == b0, bal == b0 + y),
        validate : (id == owner, result == true)
  // w2:
  guar: (bal == b0 ∧ id ≠ owner, bal == b0)
  req:  validate : (id ≠ owner, result == false)
}
```

Trivial specifications are here omitted for brevity; e.g., that bal is not modified by validate. The specifications w1 and w2 are provided under the assumption that the auxiliary calls to update and validate manipulate the fields bal and owner, which are otherwise not required by the trait. However, they are required by the trait TAux. TAux may be given the following specifications, which do not lead to any requirements:

```
trait TAux is {
  int bal; int owner;      // required fields
  void update(int y) {bal = bal + y;}
  guar: (bal == b0, bal == b0 + y)
```

```
  boolean validate(int id) {return (id == owner);}
  guar: (true,result == (id == owner))
}
```

Specification Entailment. The rule of consequence in Hoare logic [AdBO09] allows us to infer that if $\{p\}s\{q\}$ holds for some statement s and $(p,q) \twoheadrightarrow (p',q')$ then $\{p'\}s\{q'\}$. However, the order of entailment is reversed for *substitutability* of assumptions in proof outlines: Given that $\{p\}s_1;s_2;s_3\{q\}$ holds for statements s_1, s_2, and s_3 by assuming $\{p_1\}s_2\{q_1\}$, let (p_2,q_2) be such that $(p_2,q_2) \twoheadrightarrow (p_1,q_1)$. Then $\{p\}s_1;s_2;s_3\{q\}$ must hold by assuming $\{p_2\}s_2\{q_2\}$. This leads to the following observation about substitutability of decorated method calls in proof outlines:

Observation 4.5. Let s be a statement, and p,q,p_1,p_2,q_1,q_2 assertions such that $(p_2,q_2) \twoheadrightarrow (p_1,q_1)$. If $A,n() : (p_1,q_1) \vdash \{p\}s\{q\}$ then $A,n() : (p_2,q_2) \vdash \{p\}s\{q\}$.

This observation motivates how the entailment relation is lifted to method specifications to express that a specification $\langle (p_2,q_2), \overline{\mathrm{R}}_2 \rangle$ can be *derived* from a proof of a specification $\langle (p_1,q_1), \overline{\mathrm{R}}_1 \rangle$.

Definition 4.6 (Specification Entailment). Assume given specifications $\langle (p_1,q_1), \overline{\mathrm{R}}_1 \rangle$ and $\langle (p_2,q_2), \overline{\mathrm{R}}_2 \rangle$. Specification entailment, denoted $\langle (p_1,q_1), \overline{\mathrm{R}}_1 \rangle \twoheadrightarrow \langle (p_2,q_2), \overline{\mathrm{R}}_2 \rangle$, is defined by

$$\langle (p_1,q_1), \overline{\mathrm{R}}_1 \rangle \twoheadrightarrow \langle (p_2,q_2), \overline{\mathrm{R}}_2 \rangle \triangleq (p_1,q_1) \twoheadrightarrow (p_2,q_2) \wedge \overline{\mathrm{R}}_2 \twoheadrightarrow \overline{\mathrm{R}}_1.$$

The following lemma shows that specification entailment preserves the validity of specifications.

Lemma 4.7. Let the specifications $\langle (p_1,q_1), \overline{\mathrm{R}}_1 \rangle$ and $\langle (p_2,q_2), \overline{\mathrm{R}}_2 \rangle$ be such that $\langle (p_1,q_1), \overline{\mathrm{R}}_1 \rangle \twoheadrightarrow \langle (p_2,q_2), \overline{\mathrm{R}}_2 \rangle$. If $\langle (p_1,q_1), \overline{\mathrm{R}}_1 \rangle$ is valid for a method $\mathtt{m}$, then $\langle (p_2,q_2), \overline{\mathrm{R}}_2 \rangle$ is also valid for $\mathtt{m}$.

Proof. Follows from the transitivity of specification entailment. □

Assume that the user has supplied $\langle (p_2,q_2), \overline{\mathrm{R}}_2 \rangle$ as a specification for some statement s. It follows from Lemma 4.7 that to ensure this specification, we may prove a different specification $\langle (p_1,q_1), \overline{\mathrm{R}}_1 \rangle$ with a (possibly) stronger guarantee. Note that the requirement set $\overline{\mathrm{R}}_2$ may contain requirements that are superfluous in order to derive the proof outline for (p_1,q_1).

5. Compositional Verification of Traits

The goal of our verification technique is to reason incrementally about trait expressions while verifying trait-based programs. By *incremental*, we mean that new specifications may be introduced but that old specifications, and consequently old proofs, are never violated. Due to the flexible reuse potential of traits, we do not assume that a fixed specification of a trait, given a priori, covers all potential usages of that trait. This would be a special modular case of our more general incremental approach. Thus, our incremental approach subsumes a modular approach where this is applicable, but it also supports a gradual introduction of specifications in an existing trait library. Instead, traits have a set of possible method specifications for each provided method. This set can be incrementally extended. We devise compositional proof rules that apply to sets of method specifications when traits are composed.

During the verification of trait expressions, a *trait environment* is constructed to keep track of the specifications for the provided methods of trait expressions. The trait environment formalizes a "specification cache" for a trait library, and stores the currently known specifications for methods in a set of traits expressions. The trait environment supports reasoning reuse for the considered trait library in a modular way, but it may also be extended to support incremental reasoning when the knows specifications are insufficient. The analysis of a program using the traits of the library interacts with the trait environment to verify the program. For the analysis, we assume that the interfaces of a program have been annotated, so every method in an interface has a method contract as defined in Section 4. This allows us to reason compositionally about calls to methods on interface types. In this section, we examine the construction of the trait environment during the analysis of basic traits and composite traits assembled using the trait composition and alteration operations.

Basic Traits. Consider a basic trait $\mathtt{T}$ which provides a method $\mathtt{m}$ with the method body t and assume an annotated method declaration $\mathtt{M}\ \overline{\mathtt{sp}}\langle (p,q), \overline{\mathrm{R}} \rangle$ for $\mathtt{m}$. The correctness of the guarantee (p,q) can be established from the requirement set $\overline{\mathrm{R}}$ either by means of automated techniques or by asking the developer for a proof outline O and letting a verification system verify $O \vdash_{PL} t : (p,q)$. For simplicity we assume in the sequel that it is the developer who provides

a proof outline O when necessary. In the proof outline, external calls may be analyzed directly, since they are based on the method contracts in the behavioral interface of the callee. This leaves the requirements to internal calls, which are provided as annotations in the proof outline. However, the behavior of these internal calls depends on how a class is finally assembled from trait expressions. Even if the proof outline is valid, it remains to check that the requirements $\overline{\mathrm{R}}$ are correct when the class is assembled. Therefore, the method specification $\langle (p,q), \overline{\mathrm{R}} \rangle$ is stored in the trait environment for `T`. A provided method `m` may have different guarantees depending on different requirements on its internally called methods. Each of these guarantees is proven using a different proof outline, leading to different specifications for `m`. Hence, this analysis is repeated for an annotated method `AM` until the validity of all specifications has been checked.

Trait Modifiers. When a trait alteration expression modifies another trait expression `TAE`, it gets its own specification stored in the trait environment. This specification is obtained from the specification of `TAE` and may lead to new proof obligations. *Excluding* a method from a trait does not generate any proof obligations. The trait environment of the resulting trait expression is obtained from the trait environment of the previous trait expression by removing the method specifications of the removed method. *Aliasing* does not generate proof obligations. The trait environment for the resulting trait expression is obtained by copying the method specifications of the aliased method. *Renaming of methods* does not generate proof obligations, but proof obligations for distinct methods may now apply to the same method. The trait environment for the resulting trait expression is obtained by consistently renaming the respective method. *Field renaming* does not generate proof obligations, but some specifications may have to be discarded in order to maintain soundness. Therefore, the trait environment for the trait resulting from the trait alteration $[\mathtt{f}\ \mathbf{renameTo}\ \mathtt{f}']$ is obtained by distinguishing different cases. If the old field `f` does not occur in the previous trait expression, the trait environment for the new trait expression is obtained directly from the trait environment of the old trait expression. Otherwise, for each method `m`, we consider whether the new field $\mathtt{f}'$ occurs in the original body of `m` or not. If $\mathtt{f}'$ occurs in `m`, the specifications of `m` are dropped, which can be illustrated by the triple $\{b == 5\}\ a = 2 * b\ \{a == 10 \wedge b == 5\}$. After $[a\ \mathbf{renameTo}\ b]$ a direct substitution would yield the invalid triple $\{b == 5\}\ b = 2 * b\ \{b == 10 \wedge b == 5\}$. If the new variable $\mathtt{f}'$ does not occur in the original body of `m`, specifications containing occurrences of $\mathtt{f}'$ are dropped, which can be illustrated by the triple $\{b == 5\}\ a = 0\ \{b == 5\}$. This triple is dropped after the renaming $[a\ \mathbf{renameTo}\ b]$ since the triple $\{b == 5\}\ b = 0\ \{b == 5\}$ is not valid. For the remaining specifications, where $\mathtt{f}'$ does not occur in the method body or the specification, new specifications are formed directly by renaming `f` to $\mathtt{f}'$. This discussion illustrates that field renaming should be used with care in order to avoid unintended name clashes in the program (a more fine-grained analysis of field renaming may be obtained depending on the expressivity of the program logic *PL*; e.g., [PBC06]).

Example 5.1. Let the trait `TAux` be as specified in Example 4.4, and consider the rename operation `TAux[update renameTo basicUpd]`. The implementation of `validate` is unaltered by the operation, but the `update` method is renamed, and the specification given in `TAux` applies to the new method:

```
void basicUpd(int y) {bal = bal + y;}
```
guar: $(bal == b_0, bal == b_0 + y)$

Symmetric Sum of Traits. For a composed trait expression `CTE` obtained from two trait expressions by symmetric sum, we maintain the distinction that each method specification has particular assumptions on the required methods. Thus, the trait environment of the composed trait expression is the union of the trait environments of the summand trait expressions. In particular, method specifications are kept in the trait environment even if their requirements cannot become satisfied by the implementations found in other trait expressions of the composition. The reason is that the composed trait expression may be the subject to later trait composition operations. Thus, method specifications that were unsatisfiable in the original composition may again become satisfiable. However, if the composed trait expression is used in a class declaration, the analysis of the class ignores method specifications that are unnecessary in order to verify the interface contract of the class, thereby selecting a set of consistent method specifications from the set of all provided method specifications from the constituents of the composed trait expression.

6. Verifying Trait-Based Programs

For each class, it is necessary to show that every public method exposed through an interface guarantees the contract in that interface. This result should eventually follow from the specification of the methods, as provided by the trait

expression. In the case where the trait specifications contain sufficient guarantees for the public methods, this follows directly in a modular way. Otherwise, a new method specification with the additional guarantees may be added to the trait specifications, and method specifications are collected when a class is assembled from the traits by trait composition. For each added method specification, the respective method must be reinspected to verify the new proof outline associated with the new specification. Such proof outlines may lead to new requirements on internally called methods, which again make it necessary to supply new proof outlines for these methods. This procedure repeats for internal calls until the analysis is complete. Note that all proofs which rely on previously established guarantees of the provided method remain valid. Thus, the presented approach is *incremental*.

Example 6.1. Consider the analysis of the class `CAcc` from Example 3.1, which was implemented by

```
class CAcc implements IAcc
  by {int bal; int owner;} and TAcc + TAux
```

The class must implement the model variables of the interface by providing a representation function (in the terminology of JML [BCC+05]); for simplicity in the examples of this paper, the representation function will always just be the identity function and we omit its explicit representation. Thus, in order to implement the interface `IAcc`, the method `withdraw` in `TAcc` must satisfy the following contract given in Example 4.2: $(x > 0 \wedge bal == b_0, (id == owner \Rightarrow bal == b_0 - x) \wedge (id \neq owner \Rightarrow bal == b_0))$. Since this contract follows by entailment from the guarantees of `w1` and `w2` in Example 4.4, it suffices to ensure that the requirements of `w1` and `w2` are satisfied for the implementations found in `TAux`, which is straightforward.

Example 6.2. Consider the analysis of the class `CFeeAcc` of Example 3.1, which was implemented by

```
class CFeeAcc implements IFeeAcc
  by  {int bal; int owner; int fee;}
  and TAcc + TFee + TAux[update renameTo basicUpd]
```

As for `CAcc`, the representation function here is the identity function, and the contract $(x > 0 \wedge bal == b_0, (id == owner \Rightarrow bal == b_0 - x - fee) \wedge (id \neq owner \Rightarrow bal == b_0))$ is imposed by `IFeeAcc` on the method `withdraw`. This contract does not follow from the guarantees of `w1` and `w2`, so a new proof outline is needed. It suffices to extend the specifications of `withdraw` with the following specification, labelled `w3`:

```
void withdraw(int id, int x) {...}
// w3:
```
guar: $(x > 0 \wedge bal == b_0 \wedge id == owner, bal == b_0 - x - fee)$
req: `update` $: (bal == b_0 \wedge y < 0, bal == b_0 + y - fee)$,
`validate` $: (id == owner, \textbf{\textit{result}} == true)$

The interface contract follows by entailment from the guarantees `w2` and `w3`. Now, the requirements of these specifications need to be verified. The only non-trivial requirement is the one to `update`, which can be verified by the following specification in `TFee`:

```
trait TFee is {...
  void update(int y) {...}
  guar: (y < 0 ∧ bal = b0, bal = b0 + y − fee)
  req:  basicUpd : (bal = b0, bal = b0 + y)
}
```

The requirement of this specification follows from the guarantee of `basicUpd` as given in Example 5.1.

Example 6.3. Assume that the trait `TAcc` is specified as in Example 4.4. We define trait `TMini` by

```
trait TMini is {
  int bal; // required field
  void update(int y) {bal=bal+y;}
  void validate(int id) {return true;}
}
```

Assume that class `CMini` is defined by `TAcc+TMini` where `bal` is the only declared field. The original specifications `w1` and `w2` contain the field `owner`, reflecting that `TAcc` was originally developed together with `TAux`, but this field is not present in the current composition. To analyze method `withdraw`, we may provide the following specification:

```
void withdraw(int id, int x) {...}
// w4:
```
guar: $(bal == b_0, bal == b_0 - x)$
req: `update` : $(bal == b_0, bal == b_0 - x)$
`validate` : $(true, \textbf{\textit{result}} == true)$

This specification expresses that any client can make a withdrawal from a `CMiniAcc`.

7. PST(PL): A Proof System for Verifying Trait-Based Programs

The verification process for trait-based programs outlined in the previous sections is now formalized as a calculus PST(PL) which is parametric in the underlying program logic *PL*. The calculus PST(PL) relies on a sound program logic *PL* and defines inference rules for analyzing trait expressions and classes, given in Sections 7.2 and 7.3 below. The proof system considers annotated basic traits `ABT`. For simplicity, we assume that unannotated method declarations in `AM` have a default guarantee $(true, true)$, with corresponding requirements $(true, true)$ for internal calls.

A program is analyzed in PST(PL) as a sequence of analysis operations concerning the trait and class definitions of the program. These operations manipulate a *proof environment* which consists of a trait environment and a class environment. For each analyzed trait alteration expression, the trait environment is extended with the trait definition and with the specifications for the methods provided by that trait expression. Thus, if the analysis of a trait expression `TAE` is initiated in some trait environment $\mathscr{T}$, the successful analysis of `TAE` will lead to a trait environment $\mathscr{T}'$, such that $\mathscr{T}'$ extends $\mathscr{T}$. In this case we say that $\mathscr{T}'$ is the trait environment *resulting* from the analysis. When analyzing classes, the class environment is extended similarly. Traits and classes are analyzed in the context of the trait and class environments which have been obtained from the analysis of previous traits and classes. In this way, the rules of PST(PL) explain how the analysis of language artefacts constructs a sequence of proof environments, starting with empty trait and class environments. This process formalizes the accumulation of specifications for a collection of traits during the analysis of programs using those traits.

7.1. Judgments in PST(PL)

Judgments in PST(PL) are of the form $\mathscr{C}, \mathscr{T} \vdash \mathscr{P}$, where $\mathscr{C}$ is a *class environment* for class analysis, $\mathscr{T}$ is a *trait environment* for trait analysis, and $\mathscr{P}$ is a sequence of *analysis operations*. We define the syntax for judgments in PST(PL), including the analysis operations and the proof environment which is manipulated by these operations.

7.1.1. Proof Environments

Class environments are used to accumulate knowledge about classes during the analysis. Class environments, which represent classes by a unique name and a tuple $\langle \overline{\mathtt{I}}, \mathtt{CTE}, \overline{\mathtt{F}} \rangle$ of type *Class*, are defined as follows:

Definition 7.1 (Class environments). A class environment $\mathscr{C}$ consists of two mappings $D_{\mathscr{C}}$ and $S_{\mathscr{C}}$, where $D_{\mathscr{C}} : Cid \mapsto Class$, and $S_{\mathscr{C}} : Cid \times Mid \mapsto Set[Spec]$.

Here, $D_{\mathscr{C}}$ contains the definitions of verified classes and $S_{\mathscr{C}}$ contains their verified specifications. The main purpose of the class environment is to record the method specifications used to establish the contracts of the implemented interfaces. The function $impl : Class \to \mathtt{CTE}$ returns the trait expression of a class. Update functions for the class environment $\mathscr{C}$ are defined as follows, where $\overline{\mathtt{sp}}$ is a set of method specifications:

$$\mathscr{C} \oplus (\mathtt{C}, \overline{\mathtt{I}}, \mathtt{CTE}, \overline{rp_I}, \overline{\mathtt{F}}) \triangleq \langle D_{\mathscr{C}}[\mathtt{C} \mapsto \langle \overline{\mathtt{I}}, \mathtt{CTE}, \overline{rp_I}, \overline{\mathtt{F}} \rangle], S_{\mathscr{C}} \rangle$$
$$\mathscr{C} \oplus (\mathtt{C}, \mathtt{m}, \overline{\mathtt{sp}}) \triangleq \langle D_{\mathscr{C}}, S_{\mathscr{C}}[(\mathtt{C}, \mathtt{m}) \mapsto S_{\mathscr{C}}(\mathtt{C}, \mathtt{m}) \cup \overline{\mathtt{sp}}] \rangle$$

Trait environments are used to accumulate knowledge about trait expressions during analysis, and are defined as follows:

Definition 7.2 (Trait environments). A trait environment $\mathscr{T}$ is a mapping from trait alteration expressions to annotated basic traits: $\mathscr{T} : \mathtt{TAE} \to \mathtt{ABTE}$.

Recall that trait alteration expressions `TAE` are defined by `Tb` $\overline{\mathtt{ao}}$; i.e., they consist of a basic trait name followed by a possibly empty sequence of alteration operations. The mapping $\mathscr{T}$ takes such a trait alteration expression and returns

$$\begin{array}{lll} \mathscr{P} & ::= & \texttt{trait}\ \mathtt{T}\ \texttt{is}\ \{\overline{\mathtt{F}};\overline{\mathtt{H}};\overline{\mathtt{AM}}\} \mid \texttt{trait}\ \mathtt{T}\ \texttt{is}\ \mathtt{CTE} \mid \mathit{verify}(\mathtt{T},\overline{\mathtt{AM}}) \mid \mathit{extend}(\mathtt{CTE}) \\ & & \mid \texttt{class}\ \mathtt{C}\ \texttt{implements}\ \overline{\mathtt{I}}\ \texttt{by}\ \{\ \overline{rp_I}\ \overline{\mathtt{F}};\ \}\ \texttt{and}\ \mathtt{CTE} \mid \langle \mathtt{C} : \mathscr{O} \rangle \mid \mathscr{P} \cdot \mathscr{P} \\ \mathscr{O} & ::= & \varepsilon \mid \mathit{discharge}(\overline{\mathtt{R}}) \mid \mathit{analyze}(\mathtt{TAE}, \mathtt{m} : (a,a)) \mid \mathscr{O} \cdot \mathscr{O} \end{array}$$

Fig. 5. Syntax of the analysis operations.

the annotated implementation of the trait as a basic trait expression. In this manner, the trait environment is used to capture the provided methods of the different traits and to associate a number of specifications with each method.

Auxiliary functions on trait environments. For a trait alteration expression TAE, we let $\mathtt{TAE} \in \mathscr{T}$ denote that TAE is in the domain of $\mathscr{T}$, so $\mathscr{T}(\mathtt{TAE})$ is defined. For $\mathscr{T}(\mathtt{TAE}) = \{\overline{\mathtt{F}};\overline{\mathtt{H}};\overline{\mathtt{AM}}\}$ we lift functions over annotated method sets such that $\mathtt{m} \in \mathscr{T}(\mathtt{TAE}) \triangleq \mathtt{m} \in \overline{\mathtt{AM}}$, $\mathscr{T}(\mathtt{TAE})(\mathtt{m}) \triangleq \overline{\mathtt{AM}}(\mathtt{m})$, and $\mathit{mtds}(\mathscr{T}(\mathtt{TAE})) \triangleq \mathit{mtds}(\overline{\mathtt{AM}})$. For a composed trait expression CTE defined by $\mathtt{TAE}_1 + \ldots + \mathtt{TAE}_n$ where each $\mathtt{TAE}_i \in \mathscr{T}$, we define $\mathit{mtds}(\mathtt{CTE})$ straightforwardly by the union $\cup_{1 \le i \le n} \mathit{mtds}(\mathscr{T}(\mathtt{TAE}_i))$. By type safety, we may assume that name conflicts do not occur in the symmetric sum; i.e., each method in a sum of trait expressions is defined in exactly one summand $\mathtt{TAE}_i$. Let the function *addSpec* be defined such that $\mathit{addSpec}(\overline{\mathtt{AM}}, \mathtt{m}, \mathtt{sp})$ returns $\overline{\mathtt{AM}}$ where $\overline{\mathtt{AM}}(\mathtt{m})$ is extended by sp, and let $\mathit{addSpec}(\{\overline{\mathtt{F}};\overline{\mathtt{H}};\overline{\mathtt{AM}}\}, \mathtt{m}, \mathtt{sp}) \triangleq \{\overline{\mathtt{F}};\overline{\mathtt{H}};\mathit{addSpec}(\overline{\mathtt{AM}}, \mathtt{m}, \mathtt{sp})\}$.

The *update function* $\oplus$ is used to extend environments with new definitions and specifications during the analysis. Let T be the name of a basic trait defined by $\{\overline{\mathtt{F}};\overline{\mathtt{H}};\overline{\mathtt{AM}}\}$, m a method name, and $\langle (p,q), \overline{\mathtt{R}} \rangle$ a specification. A trait environment $\mathscr{T}$ is extended with new basic traits and specifications as follows:

$$\begin{array}{l} \mathscr{T} \oplus (\mathtt{T}, \{\overline{\mathtt{F}};\overline{\mathtt{H}};\overline{\mathtt{AM}}\}) \triangleq \mathscr{T}[\mathtt{T} \mapsto \{\overline{\mathtt{F}};\overline{\mathtt{H}};\overline{\mathtt{AM}}\}] \\ \mathscr{T} \oplus (\mathtt{T}, \mathtt{m}, \mathtt{sp}) \triangleq \mathscr{T}[\mathtt{T} \mapsto \mathit{addSpec}(\mathscr{T}(\mathtt{T}), \mathtt{m}, \mathtt{sp})] \end{array}$$

7.1.2. Analysis operations

The analysis assumes that basic traits are annotated as explained in Section 4. The syntax for the analysis operations $\mathscr{P}$ is given in Figure 5. Analysis operations include user-given trait and class definitions, in addition to $\mathit{verify}(\mathtt{T}, \overline{\mathtt{AM}})$, $\mathit{extend}(\mathtt{CTE})$, and $\langle \mathtt{C} : \mathscr{O} \rangle$. The operation $\mathit{verify}(\mathtt{T}, \overline{\mathtt{AM}})$ applies to a trait T, where $\overline{\mathtt{AM}}$ is a set of methods annotated with specifications to be verified for the trait. For a composed trait expression CTE, the operation $\mathit{extend}(\mathtt{CTE})$ is used to construct an entry in the trait environment for each trait alteration expression in CTE. Finally, operation $\langle \mathtt{C} : \mathscr{O} \rangle$ encapsulates the operations $\mathscr{O}$, ensuring that they are analyzed within the context of a specific class C. Each element in $\mathscr{O}$ is either a $\mathit{discharge}(\overline{\mathtt{R}})$ operation, which indicates that the requirement set $\overline{\mathtt{R}}$ must be verified for C, or an $\mathit{analyze}(\mathtt{TAE}, \mathtt{n} : (r,s))$ operation, which indicates that the requirement $\mathtt{n} : (r,s)$ must be further analyzed in the context of class C, where the definition of n stems from the trait expression TAE.

7.2. Trait Analysis in PST(PL)

The inference rules for trait analysis are given in Figure 6. Traits are analyzed compositionally in the context of the trait environment $\mathscr{T}$. A trait is either an annotated basic trait of the form **trait** Tb **is** $\{\overline{\mathtt{F}};\overline{\mathtt{H}};\overline{\mathtt{AM}}\}$ or a composed trait of the form **trait** Tc **is** CTE. For an annotated basic trait $\{\overline{\mathtt{F}};\overline{\mathtt{H}};\overline{\mathtt{AM}}\}$ we assume by type safety that $\overline{\mathtt{F}}$ and $\overline{\mathtt{H}}$ contain all fields and method signatures used in the provided methods. Furthermore, we assume for simplicity that $\overline{\mathtt{F}}$ and $\overline{\mathtt{H}}$ are *minimal* in the sense that they do not contain field and method signatures that are not used in the provided methods.

An annotated basic trait **trait** Tb **is** $\{\overline{\mathtt{F}};\overline{\mathtt{H}};\overline{\mathtt{AM}}\}$ is analyzed by the rule BASICTRAIT, which extends $\mathscr{T}$ with the definition of the trait and generates a *verify* operation for analyzing the user-given specifications for the provided methods of the trait. The *verify* operation on a set $\overline{\mathtt{AM}}$ of annotated methods is decomposed by the rules DECOMP1 and DECOMP2, which leads to a $\mathit{verify}(\mathtt{Tb}, \mathtt{M}\ \mathtt{sp})$ operation for each specification sp of M in $\overline{\mathtt{AM}}$. The predicates $\mathrm{nonempty}(\overline{\mathtt{AM}_i})$ and $\mathrm{nonempty}(\overline{\mathtt{sp}_i})$ (for $i \in \{1,2\}$) express that decomposition rules only apply to nonempty sequences. The rule ADAPTATION builds specification entailment into the proof system. This rule replaces the user-provided specification by a stronger specification. For a method $\mathtt{I}\ \mathtt{m}(\overline{\mathtt{I}}\ \overline{\mathtt{x}})\{t\}$ with the specification $\langle (p,q), \overline{\mathtt{R}} \rangle$, the rule VERIFY requires that the user provides a proof outline O for the method body t such that $O \vdash_{PL} t : (p,q)$ (for example by the user, as discussed in Section 5). In this case $\langle (p,q), \mathit{reqs}(O) \rangle$ is a valid method specification for m, and the annotated method $\mathscr{T}(\mathtt{Tb})(m)$ in the trait environment is extended with this specification.

Upon the successful analysis of Tb, each specification is recorded by the trait environment $\mathscr{T}$. Note that when

$$\text{(BasicTrait)}\quad \frac{\texttt{Tb} \notin \mathscr{T} \qquad \mathscr{C}, \mathscr{T} \oplus (\texttt{Tb}, \{\overline{\texttt{F}}; \overline{\texttt{H}}; \overline{\texttt{AM}}\}) \vdash \mathit{verify}(\texttt{Tb}, \overline{\texttt{AM}}) \cdot \mathscr{P}}{\mathscr{C}, \mathscr{T} \vdash \textbf{trait}\ \texttt{Tb}\ \textbf{is}\ \{\overline{\texttt{F}}; \overline{\texttt{H}}; \overline{\texttt{AM}}\} \cdot \mathscr{P}}$$

$$\text{(Decomp1)}\quad \frac{\text{nonempty}(\overline{\texttt{AM}_1}) \quad \text{nonempty}(\overline{\texttt{AM}_2}) \quad \mathscr{C}, \mathscr{T} \vdash \mathit{verify}(\texttt{T}, \overline{\texttt{AM}_1}) \cdot \mathit{verify}(\texttt{T}, \overline{\texttt{AM}_2}) \cdot \mathscr{P}}{\mathscr{C}, \mathscr{T} \vdash \mathit{verify}(\texttt{T}, \overline{\texttt{AM}_1}\ \overline{\texttt{AM}_2}) \cdot \mathscr{P}}$$

$$\text{(Decomp2)}\quad \frac{\text{nonempty}(\overline{\texttt{sp}_1}) \quad \text{nonempty}(\overline{\texttt{sp}_2}) \quad \mathscr{C}, \mathscr{T} \vdash \mathit{verify}(\texttt{T}, \texttt{M}\ \overline{\texttt{sp}_1}) \cdot \mathit{verify}(\texttt{T}, \texttt{M}\ \overline{\texttt{sp}_2}) \cdot \mathscr{P}}{\mathscr{C}, \mathscr{T} \vdash \mathit{verify}(\texttt{T}, \texttt{M}\ \overline{\texttt{sp}_1}\ \overline{\texttt{sp}_2}) \cdot \mathscr{P}}$$

$$\text{(Adaptation)}\quad \frac{\langle (p', q'), \overline{\texttt{R}}' \rangle \twoheadrightarrow \langle (p, q), \overline{\texttt{R}} \rangle \qquad \mathscr{C}, \mathscr{T} \vdash \mathit{verify}(\texttt{T}, \texttt{M}\langle (p', q'), \overline{\texttt{R}}' \rangle) \cdot \mathscr{P}}{\mathscr{C}, \mathscr{T} \vdash \mathit{verify}(\texttt{T}, \texttt{M}\langle (p, q), \overline{\texttt{R}} \rangle) \cdot \mathscr{P}}$$

$$\text{(Verify)}\quad \frac{O \vdash_{PL} t : (p, q) \quad \mathit{reqs}(O) = \overline{\texttt{R}} \quad \mathscr{C}, \mathscr{T} \oplus (\texttt{T}, \texttt{m}, \langle (p, q), \overline{\texttt{R}} \rangle) \vdash \mathscr{P}}{\mathscr{C}, \mathscr{T} \vdash \mathit{verify}(\texttt{T}, \texttt{I}\ \texttt{m}(\overline{\texttt{I}}\ \overline{\texttt{x}})\{t\}\langle (p, q), \overline{\texttt{R}} \rangle) \cdot \mathscr{P}}$$

$$\text{(CompTrait)}\quad \frac{\mathscr{C}, \mathscr{T} \vdash \mathit{extend}(\texttt{CTE}) \cdot \mathscr{P}}{\mathscr{C}, \mathscr{T} \vdash \textbf{trait}\ \texttt{Tc}\ \textbf{is}\ \texttt{CTE} \cdot \mathscr{P}} \qquad \text{(Decomp3)}\quad \frac{\mathscr{C}, \mathscr{T} \vdash \mathit{extend}(\texttt{TAE}) \cdot \mathit{extend}(\texttt{CTE}) \cdot \mathscr{P}}{\mathscr{C}, \mathscr{T} \vdash \mathit{extend}(\texttt{TAE} + \texttt{CTE}) \cdot \mathscr{P}}$$

$$\text{(Lookup)}\quad \frac{\texttt{TAE} \in \mathscr{T} \qquad \mathscr{C}, \mathscr{T} \vdash \mathscr{P}}{\mathscr{C}, \mathscr{T} \vdash \mathit{extend}(\texttt{TAE}) \cdot \mathscr{P}} \qquad \text{(Extend)}\quad \frac{\texttt{TAE}\ \texttt{ao} \in \mathscr{T} \qquad \texttt{TAE}\ \texttt{ao} \notin \mathscr{T} \qquad \mathscr{C}, \mathscr{T} \oplus (\texttt{TAE}, \texttt{ao}) \vdash \mathscr{P}}{\mathscr{C}, \mathscr{T} \vdash \mathit{extend}(\texttt{TAE}\ \texttt{ao}) \cdot \mathscr{P}}$$

$$\text{(ExtendRec)}\quad \frac{\texttt{TAE} \notin \mathscr{T} \qquad \mathscr{C}, \mathscr{T} \vdash \mathit{extend}(\texttt{TAE}) \cdot \mathit{extend}(\texttt{TAE}\ \texttt{ao}) \cdot \mathscr{P}}{\mathscr{C}, \mathscr{T} \vdash \mathit{extend}(\texttt{TAE}\ \texttt{ao}) \cdot \mathscr{P}}$$

Fig. 6. The inference rules for trait analysis.

the trait is defined it is not known to which actual implementation an internal call will be bound, since the method binding depends on how the traits are used to form classes. Consequently, the imposed requirements are not verified with regard to any implementation during trait analysis. Instead these requirements are verified as needed when a specification is actually used during the analysis of a class.

Composed traits **trait** Tc **is** CTE are analyzed by the rule CompTrait. Here CTE is of the form $\texttt{TAE}_1 + \ldots + \texttt{TAE}_n$ for $n \geq 1$, where each $\texttt{TAE}_i$ is of the form $\texttt{Tb}_i\ \overline{\texttt{ao}}_i$. For simplicity, we here assume that all basic traits $\texttt{Tb}_1, \ldots, \texttt{Tb}_n$ have been analyzed before the analysis of composite trait expressions. By the rule CompTrait, an operation *extend* is generated in order to extend the trait environment for each trait alteration expression $\texttt{TAE}_i$ in the composition. The composed trait expression is decomposed to a sequence of *extend* operations by the rule Decomp3. Rule Lookup applies to $\texttt{TAE}_i$ if $\texttt{TAE}_i$ is already defined in the trait environment. Especially, this rule applies to all basic trait names used in the composition. Otherwise, the analysis depends on the structure of $\texttt{TAE}_i$, and the *extend* operation is analyzed by either Extend or ExtendRec.

Rule Extend is the main rule for extending the trait environment for each $\texttt{TAE}_i$ of the form TAE ao. The premise of the rule ensures that the prefix TAE is defined in the trait environment. The trait environment is extended for TAE ao by the update $\mathscr{T} \oplus (\texttt{TAE}, \texttt{ao})$, which modifies a copy of the annotated basic trait bound to TAE depending on the actual alteration operation ao. The definition of this update function is given in Appendix A. For an expression TAE ao where TAE has not already been analyzed (i.e., TAE is not in $\mathscr{T}$), the rule ExtendRec recursively ensures that the trait environment is extended for TAE before it is extended for TAE ao. After extending the trait environment for TAE, the rule Extend may be applied to *extend*(TAE ao). Since TAE starts with a basic trait name Tb, the recursion over the structure of TAE ao is guaranteed to terminate. In this manner, properties for each $\texttt{TAE}_i$ in the composition are remembered in the trait environment. By the successful analysis of Tc, the mapping $\mathscr{T}$ binds $\texttt{TAE}_i$ to an annotated basic trait definition containing the methods provided by $\texttt{TAE}_i$ with verified specifications. These annotated methods are derived by manipulating those of $\texttt{Tb}_i$ according to the modifiers $\overline{\texttt{ao}}_i$.

A trait alteration expression TAE is introduced into $\mathscr{T}$ by rule BasicTrait or rule Extend. After the initial analysis by one of these rules, the method definitions in TAE are not manipulated, but specifications may be added.

$$\text{(CLASS)}\quad \frac{\mathscr{C} \oplus (\mathtt{C}, \overline{\mathtt{I}}, \mathtt{CTE}, \overline{rp}_I, \overline{\mathtt{F}}), \mathscr{T} \vdash \mathit{extend}(\mathtt{CTE}) \cdot \langle \mathtt{C} : \mathit{discharge}(\mathit{contracts}(\overline{rp}_I, \overline{\mathtt{I}})) \rangle \cdot \mathscr{P}}{\mathscr{C}, \mathscr{T} \vdash \mathbf{class}\ \mathtt{C}\ \mathbf{implements}\ \overline{\mathtt{I}}\ \mathbf{by}\ \{\ \overline{rp}_I\ \overline{\mathtt{F}};\ \}\ \mathbf{and}\ \mathtt{CTE} \cdot \mathscr{P}}$$

$$\text{(DECOMP4)}\quad \frac{\mathscr{C}, \mathscr{T} \vdash \langle \mathtt{C} : \mathit{discharge}(\overline{\mathtt{R}}_1) \cdot \mathit{discharge}(\overline{\mathtt{R}}_2) \cdot \mathscr{O} \rangle \cdot \mathscr{P}}{\mathscr{C}, \mathscr{T} \vdash \langle \mathtt{C} : \mathit{discharge}(\overline{\mathtt{R}}_1 \cup \overline{\mathtt{R}}_2) \cdot \mathscr{O} \rangle \cdot \mathscr{P}}$$

$$\text{(OPENANALYSIS)}\quad \frac{\mathtt{TAE} \in \mathit{impl}(D_{\mathscr{C}}(C)) \qquad \mathtt{m} \in \mathscr{T}(\mathtt{TAE}) \qquad \mathscr{C}, \mathscr{T} \vdash \langle \mathtt{C} : \mathit{analyze}(\mathtt{TAE}, \mathtt{m} : (p,q)) \cdot \mathscr{O} \rangle \cdot \mathscr{P}}{\mathscr{C}, \mathscr{T} \vdash \langle \mathtt{C} : \mathit{discharge}(\mathtt{m} : (p,q)) \cdot \mathscr{O} \rangle \cdot \mathscr{P}}$$

$$\text{(CLOSEANALYSIS)}\quad \frac{\mathit{guar}(S_{\mathscr{C}}(\mathtt{C}, \mathtt{m})) \rightarrowtriangle (p,q) \qquad \mathscr{C}, \mathscr{T} \vdash \langle \mathtt{C} : \mathscr{O} \rangle \cdot \mathscr{P}}{\mathscr{C}, \mathscr{T} \vdash \langle \mathtt{C} : \mathit{discharge}(\mathtt{m} : (p,q)) \cdot \mathscr{O} \rangle \cdot \mathscr{P}}$$

$$\text{(ANALYZE)}\quad \frac{\overline{\mathtt{sp}} \subseteq \mathit{specs}(\mathscr{T}(\mathtt{TAE})(\mathtt{m})) \qquad \mathit{guar}(\overline{\mathtt{sp}}) \rightarrowtriangle (p,q) \qquad \mathscr{C} \oplus (\mathtt{C}, \mathtt{m}, \overline{\mathtt{sp}}), \mathscr{T} \vdash \langle \mathtt{C} : \mathit{discharge}(\mathit{req}(\overline{\mathtt{sp}})) \cdot \mathscr{O} \rangle \cdot \mathscr{P}}{\mathscr{C}, \mathscr{T} \vdash \langle \mathtt{C} : \mathit{analyze}(\mathtt{TAE}, \mathtt{m} : (p,q)) \cdot \mathscr{O} \rangle \cdot \mathscr{P}}$$

$$\text{(INCREMENT)}\quad \frac{\mathscr{C}, \mathscr{T} \vdash \mathit{verify}(\mathtt{TAE}, \mathscr{T}(\mathtt{TAE})(\mathtt{m})\langle (p,q), \overline{\mathtt{R}} \rangle) \cdot \langle \mathtt{C} : \mathit{analyze}(\mathtt{TAE}, \mathtt{m} : (p,q)) \cdot \mathscr{O} \rangle \cdot \mathscr{P}}{\mathscr{C}, \mathscr{T} \vdash \langle \mathtt{C} : \mathit{analyze}(\mathtt{TAE}, \mathtt{m} : (p,q)) \cdot \mathscr{O} \rangle \cdot \mathscr{P}}$$

$$\text{(EMPDISCHARGE)}\quad \frac{\mathscr{C}, \mathscr{T} \vdash \langle \mathtt{C} : \mathscr{O} \rangle \mathscr{P}}{\mathscr{C}, \mathscr{T} \vdash \langle \mathtt{C} : \mathit{discharge}(\emptyset) \cdot \mathscr{O} \rangle \cdot \mathscr{P}} \qquad \text{(EMPCLASS)}\quad \frac{\mathscr{C}, \mathscr{T} \vdash \mathscr{P}}{\mathscr{C}, \mathscr{T} \vdash \langle \mathtt{C} : \varepsilon \rangle \cdot \mathscr{P}}$$

Fig. 7. The inference rules for class analysis. Here, $\mathtt{TAE} \in \mathtt{CTE}$ denotes that TAE is a syntactical part of CTE, i.e., if CTE is $\mathtt{TAE}_1 + \ldots + \mathtt{TAE}_n$, then $\mathtt{TAE} = \mathtt{TAE}_i$ for some i ($1 \leq i \leq n$).

The inference system for trait analysis ensures that there exists a valid proof outline for each specification recorded by the trait environment. (This is formalized by Lemma B.2 in Appendix B.) Especially, the analysis ensures that each user given specification in a basic trait is valid.

7.3. Class Analysis in PST(PL)

We now consider the analysis of a class declaration $\mathbf{class}\ \mathtt{C}\ \mathbf{implements}\ \overline{\mathtt{I}}\ \mathbf{by}\ \{\ \overline{rp}_I\ \overline{\mathtt{F}};\ \}\ \mathbf{and}\ \mathtt{CTE}$. As for composed traits, the expression CTE is of the form $\mathtt{TAE}_1 + \ldots + \mathtt{TAE}_n$ and by type safety we have that each provided method m is defined in exactly one of the summand trait expressions $\mathtt{TAE}_i$. In addition to the trait environment $\mathscr{T}$, the class analysis extends a class environment $\mathscr{C}$ which contains the definitions and specifications of classes. The analysis of class C is driven by the contracts of its interfaces $\overline{\mathtt{I}}$. Upon the successful analysis of C, each contract of a provided method m in the interfaces of the class follows by *entailment* from the guarantees of $S_{\mathscr{C}}(\mathtt{C}, \mathtt{m})$. If m is provided by $\mathtt{TAE}_i$, the interface contracts are ensured by reusing already verified specifications contained in $\mathscr{T}(\mathtt{TAE}_i)(\mathtt{m})$, and by extending this set if needed. Thus, $S_{\mathscr{C}}(\mathtt{C}, \mathtt{m})$ contains the subset of $\mathscr{T}(\mathtt{TAE}_i)(\mathtt{m})$ that is actually used to verify the current class. In addition, the requirements imposed by the used specifications are analyzed with regard to the implementation to which they bind in C. For a method specification $\mathtt{sp} = \langle (p,q), \overline{\mathtt{R}} \rangle$, we define the functions $\mathit{guar} : \mathit{Spec} \to \mathit{Guar}$ and $\mathit{req} : \mathit{Spec} \to \mathit{Set}[\mathit{Req}]$ where $\mathit{guar}(\mathtt{sp}) \triangleq (p,q)$ and $\mathit{req}(\mathtt{sp}) \triangleq \overline{\mathtt{R}}$. These functions are straightforwardly lifted to sets of method specifications, returning sets of guarantees and requirements, respectively. Thus, if $\mathtt{sp} \in S_{\mathscr{C}}(\mathtt{C}, \mathtt{m})$, then each requirement $\mathtt{n} : (r,s) \in \mathit{req}(\mathtt{sp})$ follows by entailment from the guarantees of $S_{\mathscr{C}}(\mathtt{C}, \mathtt{n})$.

Let $\mathit{contracts}(\overline{rp}_I, \overline{\mathtt{I}})$ denote the set of method contracts of $\overline{\mathtt{I}}$ obtained by applying the respective representation functions $\overline{rp}_I$ to the model variables in the assertions of each interface, which are of the form $\mathtt{m} : (p,q)$. The analysis of the class C is initiated by the rule CLASS. By this rule, the class environment $\mathscr{C}$ is extended by the definition of C, and the analysis operation $\langle \mathtt{C} : \mathit{discharge}(\mathit{contracts}(\overline{rp}_I, \overline{\mathtt{I}})) \rangle$ is generated. This operation reflects that the analysis of C

is driven by the contracts of the implemented interfaces, in which the model fields of the contracts have been adapted to the class by application of its representation function for each interface, and that these contracts are analyzed in the context of C. The set of method contracts is decomposed by the rule DECOMP4, and each contract $\mathtt{m}:(p,q)$ is analyzed either by rule OPENANALYSIS or CLOSEANALYSIS. The rule CLOSEANALYSIS applies to a *discharge* operation if the requirement follows from earlier verified specifications for the current class. Otherwise, the requirement is opened for analysis at the trait level by rule OPENANALYSIS. This rule selects the conjunct TAE in CTE where m is defined, leading to an operation $\mathit{analyze}(\mathtt{TAE},\mathtt{m}:(p,q))$. This operation is analyzed by rule ANALYZE.

Rule ANALYZE captures *modular reasoning* in the inference system, reusing previous analysis from the trait environment. This rule applies when the contract can be derived from previously verified method specifications for m in TAE. In that case, the analysis continues with an *discharge* operation for the requirements of these specifications, which are analyzed in the same manner as the original interface contract. Consequently, each requirement is either discharged by CLOSEANALYSIS, or analyzed using OPENANALYSIS with respect to the actual implementation of the called method in the current trait composition CTE.

The rule INCREMENT captures *incremental reasoning* in the inference system and allows the user to extend the verified specifications of a trait in the trait environment, incrementally building knowledge about the traits. It needs to be applied if the requirement of an *analyze* operation does not follow from previously verified specifications of the current trait. The user must suggest the additional specification $\langle(p,q),\overline{\mathtt{R}}\rangle$ for the trait, which is analyzed by a *verify* operation for the current trait. This *verify* operation is resolved by the inference rules for trait analysis and the trait environment is extended before the class analysis proceeds. To cover the case of self-recursion, the method specification itself is assumed when analyzing the requirements, similar to rule BASICTRAIT above. This captures the standard approach to reasoning about recursive method calls [Hoa71]. An empty set of requirements or contracts is discarded by the rule EMPDISCHARGE, and the successful analysis of class C is completed by the rule EMPCLASS.

7.4. Soundness of PST(PL)

When reasoning about a set of mutually recursive methods, the guarantees in the specifications of all methods are assumed to hold in order to verify the body of each method (e.g., [AdBO09]). We now extend this approach to define the *consistency* of a set of proof outlines for methods in a flattened class with given interfaces. The flattened version of a class that is defined by **class** C **implements** $\overline{\mathtt{I}}$ **by** $\{\ \overline{rp}_I,\overline{\mathtt{F}};\ \}$ **and** CTE is given by **class** C **implements** $\overline{\mathtt{I}}\ \{\ \overline{rp}_I,\overline{\mathtt{F}};\ \overline{\mathtt{M}}\ \}$ as defined in Section 3.1 (where the annotation with representation functions is unchanged by the flattening).

Definition 7.3 (Consistency). Consider a flattened class **class** C **implements** $\overline{\mathtt{I}}\ \{\ \overline{rp}_I,\overline{\mathtt{F}};\ \overline{\mathtt{M}}\ \}$. For each method $\mathtt{m}\in\overline{\mathtt{M}}$ with method body t, let $S_{\mathtt{m}}$ be a set of method specifications such that for each $\langle(p,q),\overline{\mathtt{R}}\rangle\in S_{\mathtt{m}}$, there exists a proof outline O where $O\vdash_{PL} t:(p,q)$ and $\overline{\mathtt{R}}=\mathit{reqs}(O)$. The specifications $S_{\overline{\mathtt{M}}}$ are *consistent* iff, for all $\mathtt{m}\in\overline{\mathtt{M}}$:

1. $\forall(\mathtt{m}:(r,s))\in \mathit{contracts}(\overline{rp}_I,\overline{\mathtt{I}})\ .\ \mathit{guar}(S_{\mathtt{m}})\twoheadrightarrow(r,s)$
2. $\forall\langle(p,q),\overline{\mathtt{R}}\rangle\in S_{\mathtt{m}}\ .\ \forall(\mathtt{n}:(r,s))\in\overline{\mathtt{R}}\ .\ \mathit{guar}(S_{\mathtt{n}})\twoheadrightarrow(r,s)$

The first condition expresses that the interface contracts are satisfied, whereas the second condition expresses that the requirements of all internal calls are satisfied. Previous work [DJOS10] defines a sound calculus for analyzing single inheritance class hierarchies. Given a consistent set of specifications, the analysis of a flattened class succeeds in this calculus. In order to ensure soundness of PST(PL), it thereby suffices to prove that the successful analysis of some class C leads to a consistent set of specifications for the flattened version of C.

Theorem 7.4 (Soundness of Trait Verification). For a given class **class** C **implements** $\overline{\mathtt{I}}$ **by** $\{\ \overline{\mathtt{F}};\ \}$ **and** CTE, if the successful analysis of C in PST(PL) results in a class environment $\mathscr{C}$, then the set of method specifications for C in $\mathscr{C}$ are consistent with the flattened version of C.

The proof of this theorem is given in Appendix B.

8. Example: A Trait-Based Implementation of Bank Accounts

This section illustrates how traits can be used to construct a range of classes implementing behavioral interfaces which specify different bank accounts, and how these classes can be verified by means of the proposed incremental proof system. We start with the trait definitions given in Figure 8, where each trait contains a set of annotated methods. To

```
trait TBasicAccount is {
  int bal;              // required field
  void update(nat x);   // required method
  void deposit(nat x) {bal=bal+x;} ⟨(bal == b0, bal == b0 + x), ∅⟩
  void withdraw(nat x) {update(x);}
}

trait TBasicUpd is {
  int bal;              // required field, no required methods
  void update(nat x) {bal=bal-x;} ⟨(bal == b0, bal == b0 − x), ∅⟩
}

trait TFeeUpd is {
  nat fee;              // required field
  void bUpdate(nat x); // required method
  void update(nat x) {bUpdate(x+fee)}
}

trait TPosUpd is {
  int bal;              // required field
  void bUpdate(nat x); // required method
  void update(nat x) {if (bal >= x) {bUpdate(x);}}
}
```

Fig. 8. Basic trait definitions for implementing bank accounts.

simplify the presentation, we assume that the language provides the type **nat** of positive integers. Default specifications, with guarantees $(true, true)$, are omitted. Let us assume that only a few specifications are initially provided in the traits. The incremental aspect of PST(PL) is used to provide additional specifications to the traits when these are required by the class analysis.

The trait `TBasicAccount` implements basic functionality for bank accounts by the two methods `deposit` and `withdraw`; the method `deposit` increases the balance `bal` of the account by the parameter value and `withdraw` decreases the balance of the account by calling an auxiliary method `update`. The exact definition of `update` depends on the specific account, so `update` is a required method in this trait. Additionally, `deposit` assumes access to a field `bal`, so `bal` is also required by the trait. To implement an account, the trait `TBasicAccount` must be composed with another trait where `update` is defined, in a class where `bal` is defined.

The trait `TBasicUpd` provides a basic implementation of the `update` method which simply reduces the balance by the argument value. In contrast, the trait `TFeeUpd` charges a fee for each update, and `update` is defined by calling an auxiliary, required method `bUpdate` where the argument is increased by `fee`. In a similar way, the trait `TPosUpd` calls an auxiliary, required method `bUpdate`. However, in this case the required method in `TPosUpd` is only called if the current balance is greater or equal to the parameter value.

Bank accounts with different behavioral properties may be assembled, depending on how the traits of Figure 8 are combined. Different trait combinations lead to different behaviors for the `withdraw` method. These bank accounts are first specified in terms of behavioral interfaces and then implemented in classes using different trait combinations. We define an interface hierarchy such that each type of account extends a superinterface `IAccount` which declares the methods `deposit` and `withdraw`, but where `withdraw` has no behavioral requirement. This interface further specifies the effect of making a deposit as increasing the balance of the bank account by the deposited amount. The subinterfaces vary in the properties they specify for `withdraw`. In the interface `IBasicAccount`, `withdraw` decreases the balance of the bank account by *exactly* the withdrawn amount; in `IFeeAccount` the balance is decreased by a *fee* on withdrawals; in `IPosAccount` the account cannot be overdrawn; in `IPosFeeAccount` a fee is charged on withdrawal and the withdrawn amount cannot be larger than the balance on the account; and in `IFeePosAccount`, a fee is charged on withdrawals and the balance is guaranteed not to be overdrawn. The definitions of these interfaces are given in Figure 9; observe that the method `deposit` is only specified in `IAccount` whereas `withdraw` is given different specifications in the different subinterfaces of `IAccount`.

The interfaces are implemented by classes, shown in Figure 10, which combine `TBasicAccount` with other

```
interface IAccount {
  model: int bal
  void deposit(nat x) : (bal == b0, bal == b0 + x)
  void withdraw(nat x) // no behavioral requirement
}

interface IBasicAccount extends IAccount {
  model: int bal
  void withdraw(nat x) : (bal == b0, bal == b0 - x)
}

interface IFeeAccount extends IAccount {
  model: int bal, int fee
  void withdraw(nat x) : (bal == b0, bal == b0 - x - fee)
}

interface IPosAccount extends IAccount {
  model: int bal
  void withdraw(nat x) :
    (bal == b0, (b0 ≥ x ⇒ bal == b0 - x) ∧ (b0 < x ⇒ bal == b0)),
    (bal ≥ 0, bal ≥ 0)
}

interface IPosFeeAccount extends IAccount {
  model: int bal, int fee
  void withdraw(nat x) :
     (bal == b0, (b0 ≥ x ⇒ bal == b0 - x - fee) ∧ (b0 < x ⇒ bal == b0))
     (bal ≥ 0, bal ≥ -fee)
}

interface IFeePosAccount extends IAccount {
  model: int bal, int fee
  void withdraw(nat x) :
     (bal == b0, (b0 ≥ (x + fee) ⇒ bal == b0 - x - fee) ∧ (b0 < (x + fee) ⇒ bal == b0))
     (bal ≥ 0, bal ≥ 0)
}
```

Fig. 9. Interface declarations for different bank accounts.

traits in different ways. The `update` method is renamed in some of the traits such that the call to `update` in `withdraw` may lead to a chain of calls to renamed `update` methods. The class `CBasicAccount` is defined in terms of the basic `update` method. The class `CFeeAccount` combines the basic `update` method with the one from `TFeeUpd`, so the balance is reduced by an additional fee for each withdrawal. For comparison, the flattened version of this class is shown in Figure 11. The class `CPosAccount` is defined in a similar way, but the update is not performed if it would lead to a negative balance. The classes `CPosFeeAccount` and `CFeePosAccount`, combine all the traits `TFeeUpd`, `TPosUpd`, and `TBasicUpd`. Here, the names reflect the binding order of a call to `update`. In `CPosFeeAccount`, a call to `update` will bind to the implementation found in `TPosUpd` and a recursive call to the implementation in `TFeeUpd`. Thus, in `CPosFeeAccount` the balance must be positive before the fee is added to the argument. This means that $-fee$ is the lower limit of the balance, as specified by the interface `IPosFeeAccount`. In contrast, the traits of `CFeePosAccount` are ordered such that the fee is added to the argument before the positive balance test. In this way, `CFeePosAccount` ensures that the balance is non-negative after a call to `update`. The verification of these classes with respect to their interfaces is discussed in Appendix C.

Figure 12 summarizes the environments $\mathscr{C}, \mathscr{T}$ resulting from the analysis of the different classes, focussing on the method `withdraw`. We assume that classes are analyzed in the top-down order in which they appear in the table, after the analysis of the basic trait definitions in Figure 8. Thus, the analysis is assumed to start in some initial environments

```
class CBasicAccount implements IBasicAccount
  by {int bal;} and TBasicAccount + TBasicUpd

class CFeeAccount implements IFeeAccount
  by {int bal, nat fee;}
  and TBasicAccount + TFeeUpd + TBasicUpd[update renameTo bUpdate]

class CPosAccount implements IPosAccount
  by {int bal}
  and TBasicAccount + TPosUpd + TBasicUpd[update renameTo bUpdate]

class CPosFeeAccount implements IPosFeeAccount
  by {int bal, nat fee;}
  and TBasicAccount + TPosUpd + TBasicUpd[update renameTo bbUpdate]
      + TFeeUpd[bUpdate renameTo bbUpdate][update renameTo bUpdate]

class CFeePosAccount implements IFeePosAccount
  by {int bal, nat fee;}
  and TBasicAccount + TFeeUpd + TBasicUpd[update renameTo bbUpdate]
      + TPosUpd[bUpdate renameTo bbUpdate][update renameTo bUpdate]
```

Fig. 10. Class definitions for the different bank accounts.

```
class CFeeAccount implements IFeeAccount {
  int bal, nat fee;
  void deposit (nat x) {bal=bal+x;}
  void withdraw(nat x) {update(x);}
  void update  (nat x) {bUpdate(x+fee);}
  void bUpdate (nat x) {bal=bal-x;}
}
```

Fig. 11. The flattened version of class `CFeeAccount`.

$\mathscr{C}_0$ and $\mathscr{T}_0$ where $\mathscr{C}_0$ is empty and where each basic trait is bound to an annotated basic trait expression in $\mathscr{T}_0$, corresponding to the definition found in Figure 8. The environment $\mathscr{T}_0$ is further detailed in Appendix C. Figure 12 shows the valid specifications for the methods in the classes `CFeeAccount`, `CBasicAccount`, `CPosAccount`, `CPosFeeAccount`, and `CFeePosAccount`. Furthermore, the trait in which the method is defined is indicated in the column *Trait*, and the column labelled *PO* indicates whether a new specification for the trait was needed to make the proof go through. Thus, if $\mathscr{C}$ and $\mathscr{T}$ are the class and trait environments resulting from the class analysis, a row `C m sp TAE` in the table means that $\texttt{sp} \in S_{\mathscr{C}}(\texttt{C},\texttt{m})$ and $\texttt{sp} \in \mathit{specs}(\mathscr{T}(\texttt{TAE})(\texttt{m}))$, and an $*$ indicates that a new specification for the method was provided during the analysis of `C`.

The specifications for each class in Figure 12 are consistent: Each internal call requirement follows from the guarantee of the called method, and each method contract in the interface follows from the corresponding guarantee in the class. If an interface has two contracts for `withdraw`, both are entailed by the verified specification. Note that the number of user-supplied specifications may be reduced by introducing uninterpreted boolean assertions in specifications (see [DDJ+12]). For instance, `withdraw` in `TBasicAccount` could then be specified by $\langle (p,q), \texttt{update} : (p,q) \rangle$ for some p and q, which means that it would suffice to provide only one proof outline for this method.

9. Related Work

This section relates the presented work on a proof system for a trait-based object-oriented language to related work on proof systems for object-oriented programming languages. We refer to [BDSS13] for an in-depth discussion of traits

Class	Method	Valid specifications	Trait	PO
CFeeAccount	withdraw	$\langle(bal == b_0, bal == b_0 - x - fee),$ $\texttt{update} : (bal == b_0, bal == b_0 - x - fee)\rangle$	TBasicAccount	$*$
	update	$\langle(bal == b_0, bal == b_0 - x - fee)$ $\texttt{bUpdate} : (bal == b_0, bal == b_0 - x)\rangle$	TFeeUpd	$*$
	bUpdate	$\langle(bal == b_0, bal == b_0 - x), \emptyset\rangle$	TBasicUpd[bU]	
CBasicAccount	withdraw	$\langle(bal == b_0, bal == b_0 - x), \texttt{update} : (bal == b_0, bal == b_0 - x)\rangle$	TBasicAccount	$*$
	update	$\langle(bal == b_0, bal == b_0 - x), \emptyset\rangle$	TBasicUpd	
CPosAccount	withdraw	$\langle(bal == b_0,\ (b_0 \geq x \Rightarrow bal == b_0 - x) \wedge (b_0 < x \Rightarrow bal == b_0)),$ $\texttt{update} : (bal == b_0, (b_0 \geq x \Rightarrow bal == b_0 - x) \wedge$ $(b_0 < x \Rightarrow bal == b_0))\rangle$	TBasicAccount	$*$
	update	$\langle(bal == b_0,\ b_0 \geq x \Rightarrow bal == b_0 - x) \wedge (b_0 < x \Rightarrow bal == b_0)),$ $\texttt{bUpdate} : (bal == b_0, bal == b_0 - x)\rangle$	TPosAccount	$*$
	bUpdate	$\langle(bal == b_0, bal == b_0 - x), \emptyset\rangle$	TBasicUpd[bU]	
CPosFeeAccount	withdraw	$\langle(bal == b_0,\ (b_0 \geq x \Rightarrow bal == b_0 - x - fee) \wedge$ $(b_0 < x \Rightarrow bal == b_0)),$ $\texttt{update} : (bal == b_0, (b_0 \geq x \Rightarrow bal == b_0 - x - fee) \wedge$ $(b_0 < x \Rightarrow bal == b_0))\rangle$	TBasicAccount	$*$
	update	$\langle(bal == b_0,\ (b_0 \geq x \Rightarrow bal == b_0 - x - fee) \wedge$ $(b_0 < x \Rightarrow bal == b_0)),$ $\texttt{bUpdate} : (bal == b_0, bal == b_0 - x - fee)\rangle$	TPosAccount	$*$
	bUpdate	$\langle(bal == b_0, bal == b_0 - x - fee),$ $\texttt{bbUpdate} : (bal == b_0, bal == b_0 - x)\rangle$	TFeeUpd[bbU]	
	bbUpdate	$\langle(bal == b_0, bal == b_0 - x), \emptyset\rangle$	TBasicUpd[bbU]	
CFeePosAccount	withdraw	$\langle(bal == b_0,\ (b_0 \geq (x + fee) \Rightarrow bal == b_0 - x - fee) \wedge$ $(b_0 < (x + fee) \Rightarrow bal == b_0)),$ $\texttt{update} : (bal == b_0, (b_0 \geq (x + fee) \Rightarrow bal == b_0 - x - fee) \wedge$ $(b_0 < (x + fee) \Rightarrow bal == b_0))\rangle$	TBasicAccount	$*$
	update	$\langle(bal == b_0,\ (b_0 \geq (x + fee) \Rightarrow bal == b_0 - x - fee) \wedge$ $(b_0 < (x + fee) \Rightarrow bal == b_0)),$ $\texttt{bUpdate} : (bal == b_0, (b_0 \geq x \Rightarrow bal == b_0 - x) \wedge$ $(b_0 < x \Rightarrow bal == b_0))\rangle$	TFeeAccount	$*$
	bUpdate	$\langle(bal == b_0, (b_0 \geq x \Rightarrow bal == b_0 - x) \wedge (b_0 < x \Rightarrow bal == b_0)),$ $\texttt{bbUpdate} : (bal == b_0, bal == b_0 - x)\rangle$	TPosUpd[bbU]	
	bbUpdate	$\langle(bal == b_0, bal == b_0 - x), \emptyset\rangle$	TBasicUpd[bbU]	

Abbreviations for trait expressions:

```
TBasicUpd[bU]  = TBasicUpd[update renameTo bUpdate]
TBasicUpd[bbU] = TBasicUpd[update renameTo bbUpdate]
TFeeUpd[bbU]   = TFeeUpd[bUpdate renameTo bbUpdate][update renameTo bUpdate]
TPosUpd[bbU]   = TPosUpd[bUpdate renameTo bbUpdate][update renameTo bUpdate]
```

Fig. 12. Summary of the analysis of the `withdraw` method in each class.

and of how the trait language considered in this paper relates to other trait languages. We do not attempt to give a general discusion of object-oriented program verification here, but focus on how different approaches address code reuse mechanisms.

Multiple specifications of methods have been recognized as a convenient way of specifying behavior, for example in Fresco capsules [Wil91], with the also-constructs of JML [BCC+05], and in Parkinson and Bierman's work [PB08]. In these approaches, two specifications are flattened (or "desugared") into a single specification by a composition rule; e.g., in Fresco two specifications (p_1, q_1) and (p_2, q_2) of the same method have the interpretation $(p_1 \vee p_2, (p_1 \rightarrow q_1^o) \wedge (p_2 \rightarrow q_2^o))$ according to our notation. Note that this example demonstrates how it is never necessary to allow multiple specifications, but rather it is a convenience which is particularly clear with incremental approaches to deductive reasoning. In the context of the flexible code reuse offered by traits, there can be several specifications and we have therefore opted for an approach based on reasoning over sets of specifications instead of flattening these sets.

Single inheritance is the object-oriented code reuse mechanism which is by far the most studied and best supported in formal systems for program analysis. In the context of single inheritance, behavioral reasoning about extensible class hierarchies with late bound method calls is often performed in the context of *behavioral subtyping* (see, e.g., [LW94, Ame91, LN06, PHM99]). Behavioral subtyping is an incremental reasoning strategy in the sense that a subclass may be analyzed in the context of previously defined classes. In order to avoid reverification of superclasses, method overriding must preserve the specifications of overridden methods. This approach has also been

used for SCALA's 'trait' construct, but *"significantly reduced the applicability and thereby benefits of traits"* [Sch10]. Combining separation logic with object-oriented structuring mechanisms, Parkinson and Bierman propose abstract predicate families which relate a set of implementations to the late binding mechanism [PB08]. The approach separates behavioral subtyping requirements on dynamic specifications from code reuse flexibility on static specifications and goes beyond behavioral subtyping by allowing new specifications of inherited code in a subclass, which need not respect the dynamic specifications of the superclass. *Lazy behavioral subtyping* [DJOS10] is an incremental reasoning strategy which supports more flexible code reuse than behavioral subtyping. With lazy behavioral subtyping, the *requirements* that a method guarantee imposes on late bound method calls are identified, and the main idea is that there is no need to preserve the full specifications of overridden methods. In order to avoid reverification of superclass methods, only the weaker *requirements* imposed on late bound method calls need to be preserved by method redefinitions in subclasses. Lazy behavioral subtyping is more flexible than the approach of [PB08] as it completely separates interface inheritance (for late bound external calls) and code reuse, and only requires compliance for requirements on internal calls instead of their specifications. Although traits do not support late bound internal calls, the flexible reuse of traits motivates us to follow the lazy behavioral subtyping approach in maintaining a separation of concerns between required and guaranteed assertions for method calls and definitions, respectively, and in allowing sets of specifications for each method definition.

Multiple inheritance is widely used in modeling notations such as UML [BRJ99], to capture that a concept naturally inherits from several other concepts. Versions of multiple inheritance are found in C++, CLOS, Eiffel, and OCaml. Creol [JOY06] has proposed a so-called healthy binding strategy which resolves horizontal name conflicts by avoiding accidental overriding. The proof systems presented in [NCMM09, DJOS11, LQ08, vSC10] are the only proof systems we know for multiple inheritance class hierarchies. The work in [NCMM09] presents a Hoare-style program logic for Eiffel that handles multiple inheritance based on an existing program logic for single inheritance by extending the method lookup definition. In [LQ08], method calls are assumed to be fully qualified in order to avoid ambiguities, and diamond inheritance is not considered. In [vSC10], ambiguities are assumed to be resolved by the programmer, a method can only be inherited if it has the same implementation in all parents. In contrast, [DJOS11] applies lazy behavioral subtyping to multiple inheritance and shows that healthy method binding is sufficient to allow incremental reasoning about multiple inheritance class hierarchies. The main challenges for reasoning about class hierarchies with multiple inheritance are related to late bound method calls, such as accidental or ambiguous overriding (sometimes called the "diamond problem"). In contrast, traits do not support late binding and require explicit disambiguation from the programmer. However, the flexible composition supported by traits necessitates a *delayed selection* of relevant method specifications, such that the requirements in our proof system are first checked when classes are assembled from traits. Technically, this makes lazy behavioral subtyping fairly different from the proposed proof system for traits. We are not aware of any previous proposal for a deductive proof system for a trait-based language.

Invasive code reuse mechanisms such as aspect-oriented programming (AOP) [KLM+97, FES10], feature-oriented programming (FOP) [BSR04, AKL13] and delta-oriented programming (DOP) [SBB+10] have been proposed to improve code reuse at different levels of the software design by allowing methods defined in one module to be intercepted or changed in another module. Similar to traits, this code reuse flexibility poses a challenge for program verification. In contrast to traits, AOP, FOP and DOP have a notion of original call, which allows the redefined method to be called from inside the redefinition. Most prominently, AOP aims to modularize code that crosscuts the basic program modules [KLM+97]; so-called advice is used to allow a definition in an aspect to affect methods defined elsewhere. The two levels of modularity supported by AOP often improve code reuse in a program, compared to a single level of modularity as in traditional object-oriented programming. However, the highly invasive nature of aspects makes it very challenging to reason formally about program behavior. In contrast to traits, aspects rarely modify the state of a program in practice [KF07]; i.e., the aspects are largely external to the functionality of the core programs. Neither Aldrich [Ald05] nor Krishnamurthi and Fisler [KF07] address aspects which modify the program state in their work. Extending AOP with open modules allow hiding implementation details from the advice mechanism, by using interfaces to export pointcuts. For a functional core calculus of open modules, Aldrich uses weak bisimulation techniques to reason about module equivalence with respect to the application of advice [Ald05]. Krishnamurthi and Fisler use CTL model checking to verify AOP programs expressed as state machines with pointcut interfaces [KF07], to identify when the application of an aspect may violate properties of the original program. Similar to our work their approach is incremental and uses a cache. Extending the idea of pointcut interfaces, translucent contracts have been proposed to behaviorally restrict the applicability of aspects [BRLM11], thereby making AOP amenable to modular deductive verification. These works on AOP are akin to ours in that they address reasoning about mechanisms which aim at better code reuse than class inheritance. However, the proposed use of interfaces between classes and aspects would quickly become unwieldy if adapted to the composition of trait expressions.

Whereas AOP addresses changes to method calls in terms of pointcuts, FOP and DOP address changes to a set

of classes to accomodate feature selection in software product lines, allowing in particular methods to be redefined. Verification of FOP programs based on a meta-representation of all program variants that can be generated from a set of feature modules is considered in [TSAH12]. In [TSKA11], proof scripts for single feature modules are extracted from proofs for complete program variants and later composed for other program variants using these feature modules. Hähnle and Schaefer propose an approach to deductive verification for DOP which relies on behavioral subtyping for DOP [HS12]; i.e., specifications of methods introduced by deltas must be more specific than previous versions of these methods. Thus, a delta can only modify state in a way which refines the program state and each delta can be verified by approximating called methods defined in other deltas by the specification of their first declaration. We have proposed a different approach [DDJ$^+$12]; which considers deltas as transformations of a program, and delta specifications as higher-order specifications which are instantiated at the delta composition time to transform a specific specification. This transformational approach uses symbolic assumptions on called methods and thus separates the specifications of method implementations from the requirements to method calls in a way which is similar to lazy behavioral subtyping [DJOS10]. The transformational approach leads to a two-phase verification process: the verification of deltas and the verification of the actual products based on the specifications already established for the used deltas. We believe that a transformational approach similar to [DDJ$^+$12] could be applied to traits and a comparison between the approach taken in this paper and a transformational proof system would be an interesting extension of the work presented in this paper.

10. Conclusion and Future Work

This paper describes an approach to behavioral reasoning about trait-based object-oriented programs. Traits have been proposed as a particularly flexible way to achieve a high degree of fine-grained code reuse. We develop a deductive proof system, PST(PL), which reflects this fine-grained reuse potential at the level of behavioral reasoning. The approach focusses on verifying interface contracts for classes assembled from traits, based on method specifications in the traits. For flexible reuse, methods in traits have associated sets of possible specifications. These sets can be extended with new specifications as needed for class verification in an incremental way. When verifying a class, the interface contracts of the class drive the selection of possible specifications from the specification sets in the traits, in such a way that the internal consistency among the selected specifications is guaranteed and such that the interface contracts are valid. We show the soundness of PST(PL).

Trait-based code reuse is an interesting challenge for deductive proof systems because a high degree of reuse in a code structuring mechanism typically limits the degree of reasoning reuse which can be supported by the proof system. In particular, a trait cannot be fully verified independently of its context of composition. The presented approach addresses this challenge by proposing an incremental and compositional proof system for traits, in which specifications for methods in trait alteration expressions and in composed trait expressions are compositionally derived from basic trait expressions, and in which the sets of possible specifications can be incremented at need during the verification process. The approach combines bottom-up compositional reasoning about trait expressions with top-down selection of consistent specifications for methods declared in different traits during the verification process for classes.

The complexity of verifying trait-based programs using our proof system can be compared to the naive approach of first flattening the programs and then verifying them. When flattening each program built from a set of traits before verifying it, we need to verify each method in isolation which requires as many proofs are there are methods. In this case, there is no specification reuse. In the completely modular case of our proof system where the specifications in the trait environment suffice to establish the external requirements, we do not need any new code analysis to verify the new program. In the fully incremental case of our proof system where each external requirement needs a new trait specification to be verified, we need one proof for each method which is the same as for the naive approach. In all cases where some external requirements can be shown modularly from existing specifications and some external specifications need incremental verification by adding new specifications, we have some specification reuse and thereby less code analysis than when verifying flattened programs. With our approach, the gain comes from analyzing many programs with the incrementally constructed trait environment, to achieve specification reuse similar to modular reasoning for common ways of using of traits.

In this paper, we have concentrated on verifying method contracts given as pre- and postconditions. As future work, it is interesting to investigate to what extent invariants could be useful for trait-based programs, both at the level of classes and traits. A trait invariant can, for instance, capture relations between the required fields of a trait; this will extend the range of properties that can be incrementally verified for trait-based programs. Further, we plan to extend the KeY system [BHS07] for deductive verification of JAVA programs to a trait-based language such as TRAITRECORDJ [BDSS13] and to implement the proof system proposed in this paper within KeY.

Acknowledgements. The authors are grateful to Wolfgang Ahrendt, Richard Bubel, Olaf Owe, and Volker Stolz for valuable discussions on the subject of this work, and to the FAOC editor and anonymous reviewers for many useful comments and suggestions for improving the presentation and for further pointers to related work.

References

[AdBO09] Krzysztof R. Apt, Frank S. de Boer, and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Systems*. Texts and Monographs in Computer Science. Springer, 3rd edition, 2009.

[AKL13] Sven Apel, Christian Kästner, and Christian Lengauer. Language-independent and automated software composition: The FeatureHouse experience. *IEEE Transactions on Software Engineering*, 39(1):63–79, 2013.

[Ald05] Jonathan Aldrich. Open modules: Modular reasoning about advice. In Andrew P. Black, editor, *Proc. European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *Lecture Notes in Computer Science*, pages 144–168. Springer, 2005.

[ALZ03] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam—designing a Java extension with mixins. *Transactions on Programming Languages and Systems*, 25(5):641–712, September 2003.

[Ame91] Pierre America. Designing an object-oriented programming language with behavioural subtyping. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer, 1991.

[Apt81] Krzysztof R. Apt. Ten years of Hoare's logic: A survey — Part I. *Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / European Conference on Object-Oriented Programming*, pages 303–311. ACM Press, 1990.

[BCC+05] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.

[BCD12] Lorenzo Bettini, Sara Capecchi, and Ferruccio Damiani. On flexible dynamic trait repacement for Java-like languages. *Science of Computer Programming*, 2012. Available online `doi:10.1016/j.scico.2012.11.003`.

[BDD+10] Lorenzo Bettini, Ferruccio Damiani, Marco De Luca, Kathrin Geilmann, and Jan Schäfer. A calculus for boxes and traits in a Java-like setting. In Dave Clarke and Gul Agha, editors, *Coordination Models and Languages*, volume 6116 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2010.

[BDG07] Viviana Bono, Ferruccio Damiani, and Elena Giachino. Separating Type, Behavior, and State to Achieve Very Fine-grained Reuse. In *Proc. Formal Techniques for Java-like Programs (FTfJP)*, 2007.

[BDG08] Viviana Bono, Ferruccio Damiani, and Elena Giachino. On Traits and Types in a Java-like setting. In *Fifth IFIP International Conference On Theoretical Computer Science (TCS'08)*, volume 273 of *International Federation for Information Processing*, pages 367–382. Springer, 2008.

[BDGS13] Lorenzo Bettini, Ferruccio Damiani, Kathrin Geilmann, and Jan Schäfer. Combining traits with boxes and ownership types in a Java-like setting. *Science of Computer Programming*, 78(2):218–247, 2013.

[BDN+09] Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, 2009.

[BDNW08] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Stateful traits and their formalization. *Computer Languages, Systems & Structures*, 34(2-3):83–108, 2008.

[BDS09] Lorenzo Bettini, Ferruccio Damiani, and Ina Schaefer. Implementing SPL using Traits. Technical report, Dipartimento di Informatica, Università di Torino, 2009. Available at `www.di.unito.it/~damiani/papers/isplurat.pdf`.

[BDS10] Lorenzo Bettini, Ferruccio Damiani, and Ina Schaefer. Implementing software product lines using traits. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC'10)*, pages 2096–2102. ACM Press, 2010.

[BDSS10] Lorenzo Bettini, Ferruccio Damiani, Ina Schaefer, and Fabio Strocco. A prototypical Java-like language with records and traits. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java (PPPJ'10)*, pages 129–138. ACM Press, 2010.

[BDSS13] Lorenzo Bettini, Ferruccio Damiani, Ina Schaefer, and Fabio Strocco. TRAITRECORDJ: A programming language with traits and records. *Science of Computer Programming*, 2013. In press, available online `doi:10.1016/j.scico.2011.06.007`.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer, 2007.

[BRJ99] Grady Booch, James E. Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[BRLM11] Mehdi Bagherzadeh, Hridesh Rajan, Gary T. Leavens, and Sean L. Mooney. Translucid contracts: expressive specification and modular verification for aspect-oriented interfaces. In Paulo Borba and Shigeru Chiba, editors, *Proceedings of the 10th International Conference on Aspect-Oriented Software Development (AOSD 2011)*, pages 141–152. ACM Press, 2011.

[BSR04] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30:355–371, 2004.

[BvW98] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer, 1998.

[dB99] Frank S. de Boer. A WP-calculus for OO. In Wolfgang Thomas, editor, *Proceedings of Foundations of Software Science and Computation Structure (FOSSACS'99)*, volume 1578 of *Lecture Notes in Computer Science*, pages 135–149. Springer, 1999.

[DDJ+12] Ferruccio Damiani, Johan Dovland, Einar Broch Johnsen, Olaf Owe, Ina Schaefer, and Ingrid Chieh Yu. A transformational proof system for delta-oriented programming. In Eduardo Santana de Almeida, Christa Schwanninger, and David Benavides, editors, *Proc. 16th International Software Product Line Conference (SPLC'12), Volume 2*, pages 53–60. ACM Press, 2012.

[DDJS11] Ferrucio Damiani, Johan Dovland, Einar Broch Johnsen, and Ina Schaefer. Verifying traits: A proof system for fine-grained reuse. In *Proc. Formal Techniques for Java-like Programs (FTfJP)*, pages 8:1–8:6. ACM Press, 2011.

[DJOS10] Johan Dovland, Einar Broch Johnsen, Olaf Owe, and Martin Steffen. Lazy behavioral subtyping. *Journal of Logic and Algebraic Programming*, 79(7):578–607, 2010.

[DJOS11] Johan Dovland, Einar Broch Johnsen, Olaf Owe, and Martin Steffen. Incremental reasoning with lazy behavioral subtyping for multiple inheritance. *Science of Computer Programming*, 76(10):915–941, 2011.

[DNS+06] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits: A mechanism for fine-grained reuse. *Transactions on Programming Languages and Systems*, 28(2):331–388, 2006.

[FES10] Bruno De Fraine, Erik Ernst, and Mario Südholt. Essential AOP: The A calculus. In Theo D'Hondt, editor, *Proc. European Conference on Object-Oriented Programming (ECOOP)*, volume 6183 of *Lecture Notes in Computer Science*, pages 101–125. Springer, 2010.

[FKF98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proc. Principles of Programming Languages (POPL)*, pages 171–183. ACM Press, 1998.

[HLL+12] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew J. Parkinson. Behavioral interface specification languages. *ACM Computing Surveys*, 44(3):16, 2012.

[Hoa69] Charles Antony Richard Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580, 1969.

[Hoa71] Charles Antony Richard Hoare. Procedures and parameters: An axiomatic approach. In E. Engeler, editor, *Symposium On Semantics of Algorithmic Languages*, volume 188 of *Lecture Notes in Mathematics*, pages 102–116. Springer, 1971.

[HS12] Reiner Hähnle and Ina Schaefer. A Liskov principle for delta-oriented programming. In Tiziana Margaria and Bernhard Steffen, editors, *Proc. 5th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, volume 7609 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2012.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[JOY06] Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2):23–66, November 2006.

[KF07] Shriram Krishnamurthi and Kathi Fisler. Foundations of incremental aspect model-checking. *Transactions on Software Engineering and Methodology*, 16(2), 2007.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proc. European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

[LM96] Marc Van Limberghen and Tom Mens. Encapsulation and composition as orthogonal operators on mixins: A solution to multiple inheritance problems. *Object Oriented Systems*, 3(1):1–30, 1996.

[LN06] Gary T. Leavens and David A. Naumann. Behavioral subtyping, specification inheritance, and modular reasoning. Technical Report 06-20a, Department of Computer Science, Iowa State University, Ames, Iowa, 2006.

[LQ08] Chenguang Luo and Shengchao Qin. Separation logic for multiple inheritance. *Electronic Notes in Theoretical Computer Science*, 212:27–40, April 2008. Proc. First Intl. Conf. on Foundations of Informatics, Computing and Software (FICS).

[LS08a] Luigi Liquori and Arnaud Spiwack. Extending FeatherTrait Java with interfaces. *Theoretical Computer Science*, 398(1-3):243–260, 2008.

[LS08b] Luigi Liquori and Arnaud Spiwack. FeatherTrait: A modest extension of Featherweight Java. *Transactions on Programming Languages and Systems*, 30(2):1–32, 2008.

[LSZ09] Giovanni Lagorio, Marco Servetto, and Elena Zucca. Flattening versus direct semantics for Featherweight Jigsaw. In *Proc. Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2009.

[LW94] Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[NCMM09] Martin Nordio, Cristiano Calcagno, Peter Müller, and Bertrand Meyer. A Sound and Complete Program Logic for Eiffel. In M. Oriol, editor, *TOOLS-EUROPE 2009*, volume 33 of *Lecture Notes in Business and Information Processing*, pages 195–214, 2009.

[NDS06] Oscar Nierstrasz, Stéphane Ducasse, and Nathanael Schärli. Flattening traits. *Journal of Object Technology*, 5(4):129–148, 2006.

[OG76] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, 1976.

[OSV10] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Press, 2 edition, 2010.

[PB08] Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In *Proc. Principles of Programming Languages (POPL)*, pages 75–86. ACM Press, 2008.

[PBC06] Matthew J. Parkinson, Richard Bornat, and Cristiano Calcagno. Variables as resource in Hoare logics. In *Proc. Symposium on Logic in Computer Science (LICS'06)*, pages 137–146. IEEE Computer Society Press, 2006.

[PHM99] Arnd Poetzsch-Heffter and Peter Müller. A programming logic for sequential Java. In S. Doaitse Swierstra, editor, *8th European Symposium on Programming Languages and Systems (ESOP'99)*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer, 1999.

[RT06] John H. Reppy and Aaron Turon. A foundation for trait-based metaprogramming. In *Proceedings of FOOL/WOOD*, 2006.

[RT07] John H. Reppy and Aaron Turon. Metaprogramming with traits. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, volume 4609 of *Lecture Notes in Computer Science*, pages 373–398. Springer, 2007.

[SAC+11] Guy L. Steele Jr., Eric E. Allen, David Chase, Christine H. Flood, Victor Luchangco, Jan-Willem Maessen, and Sukyoung Ryu. Fortress (Sun HPCS language). In David A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 718–735. Springer, 2011.

[SBB+10] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-Oriented Programming of Software Product Lines. In Jan Bosch and Jaejoon Lee, editors, *Software Product Lines: Going Beyond (SPLC'10)*, volume 6287 of *Lecture Notes in Computer Science*, pages 77–91. Springer, 2010.

[Sch10] Malte Schwerhoff. Verifying Scala traits. Semester Report, Swiss Federal Institute of Technology Zurich (ETH), October 2010.

[SD05] Charles Smith and Sophia Drossopoulou. *Chai*: Traits for Java-like languages. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *Lecture Notes in Computer Science*, pages 453–478. Springer, 2005.

$$\mathscr{T} \oplus (\texttt{TAE}, \texttt{ao}) \triangleq \mathscr{T}[\texttt{TAE ao} \mapsto \mathit{mod}(\mathscr{T}(\texttt{TAE}), \texttt{ao})]$$

$$\mathit{mod}(\{\overline{\texttt{F}}; \overline{\texttt{H}}; \overline{\texttt{AM}}\}, [\textbf{exclude}\ \texttt{m}]) \triangleq \begin{cases} \{\overline{\texttt{F}}; \overline{\texttt{H}}; \overline{\texttt{AM}}\} & \textbf{if } \texttt{m} \notin \overline{\texttt{AM}} \\ \{\mathit{fields}(\mathit{remMtd}(\overline{\texttt{AM}}, \texttt{m})); \mathit{calls}(\mathit{remMtd}(\overline{\texttt{AM}}, \texttt{m})); \mathit{remMtd}(\overline{\texttt{AM}}, \texttt{m})\} & \textbf{otherwise} \end{cases}$$

$$\mathit{mod}(\{\overline{\texttt{F}}; \overline{\texttt{H}}; \overline{\texttt{AM}}\}, [\texttt{m}\ \textbf{aliasAs}\ \texttt{m}']) \triangleq \begin{cases} \{\overline{\texttt{F}}; \overline{\texttt{H}}; \overline{\texttt{AM}}\} & \textbf{if } \texttt{m} \notin \overline{\texttt{AM}} \\ \{\overline{\texttt{F}}; \overline{\texttt{H}} \setminus \texttt{m}'; \overline{\texttt{AM}}\ \mathit{renId}(\overline{\texttt{AM}}(\texttt{m}), \texttt{m}, \texttt{m}')\} & \textbf{otherwise} \end{cases}$$

$$\mathit{mod}(\{\overline{\texttt{F}}; \overline{\texttt{H}}; \overline{\texttt{AM}}\}, [\texttt{m}\ \textbf{renameTo}\ \texttt{m}']) \triangleq \{\overline{\texttt{F}}; \mathit{calls}(\mathit{renMtd}(\overline{\texttt{AM}}, \texttt{m}, \texttt{m}')); \mathit{renMtd}(\overline{\texttt{AM}}, \texttt{m}, \texttt{m}')\}$$

$$\mathit{mod}(\{\overline{\texttt{F}}; \overline{\texttt{H}}; \overline{\texttt{AM}}\}, [\texttt{f}\ \textbf{renameTo}\ \texttt{f}']) \triangleq \{\mathit{fields}(\mathit{renFld}(\overline{\texttt{AM}}, \texttt{f}, \texttt{f}')); \overline{\texttt{H}}; \mathit{renFld}(\overline{\texttt{AM}}, \texttt{f}, \texttt{f}')\}$$

$$\mathit{remMtd}(\emptyset, \texttt{m}) \triangleq \emptyset$$

$$\mathit{remMtd}(\texttt{I n}(\overline{\texttt{I}}\ \overline{\texttt{x}})\{t\}\ \overline{\texttt{sp}}\ \overline{\texttt{AM}}, \texttt{m}) \triangleq \begin{cases} \overline{\texttt{AM}} & \textbf{if } \texttt{m} = \texttt{n} \\ \texttt{I n}(\overline{\texttt{I}}\ \overline{\texttt{x}})\{t\}\ \overline{\texttt{sp}}\ \mathit{remMtd}(\overline{\texttt{AM}}, \texttt{m}) & \textbf{otherwise} \end{cases}$$

$$\mathit{renId}(\texttt{I n}(\overline{\texttt{I}}\ \overline{\texttt{x}})\{t\}\ \overline{\texttt{sp}}, \texttt{m}, \texttt{m}') \triangleq \texttt{I n}[\texttt{m}'/\texttt{m}](\overline{\texttt{I}}\ \overline{\texttt{x}})\{t\}\ \overline{\texttt{sp}}$$

$$\mathit{renMtd}(\emptyset, \texttt{m}, \texttt{m}') \triangleq \emptyset$$

$$\mathit{renMtd}(\texttt{AM}\ \overline{\texttt{AM}}, \texttt{m}, \texttt{m}') \triangleq \mathit{renMtd}(\texttt{AM}, \texttt{m}, \texttt{m}')\ \mathit{renMtd}(\overline{\texttt{AM}}, \texttt{m}, \texttt{m}')$$

$$\mathit{renMtd}(\texttt{I n}(\overline{\texttt{I}}\ \overline{\texttt{x}})\{t\}\ \overline{\texttt{sp}}, \texttt{m}, \texttt{m}') \triangleq \texttt{I n}[\texttt{m}'/\texttt{m}](\overline{\texttt{I}}\ \overline{\texttt{x}})\{t[\texttt{m}'/\texttt{m}]\}\ \mathit{renSM}(\overline{\texttt{sp}}, \texttt{m}, \texttt{m}')$$

$$\mathit{renFld}(\emptyset, \texttt{f}, \texttt{f}') \triangleq \emptyset$$

$$\mathit{renFld}(\texttt{AM}\ \overline{\texttt{AM}}, \texttt{f}, \texttt{f}') \triangleq \mathit{renFld}(\texttt{AM}, \texttt{f}, \texttt{f}')\ \mathit{renFld}(\overline{\texttt{AM}}, \texttt{f}, \texttt{f}')$$

$$\mathit{renFld}(\texttt{M}\ \overline{\texttt{sp}}, \texttt{f}, \texttt{f}') \triangleq \begin{cases} \mathit{renMF}(\texttt{M}, \texttt{f}, \texttt{f}')\ \mathit{discard}(\overline{\texttt{sp}}, \{\texttt{f}, \texttt{f}', \mathit{assign}(\texttt{M})\}) & \textbf{if } \texttt{f} \in \mathit{fields}(\texttt{M}) \wedge \texttt{f}' \in \mathit{fields}(\texttt{M}) \\ \mathit{renMF}(\texttt{M}, \texttt{f}, \texttt{f}')\ \mathit{renSF}(\mathit{discard}(\overline{\texttt{sp}}, \{\texttt{f}'\}), \texttt{f}, \texttt{f}') & \textbf{if } \texttt{f}' \notin \mathit{fields}(\texttt{M}) \\ \texttt{M}\ \mathit{discard}(\overline{\texttt{sp}}, \{\texttt{f}\}) & \textbf{otherwise} \end{cases}$$

$$\mathit{renSM}(\varepsilon, \texttt{m}, \texttt{m}') \triangleq \varepsilon$$

$$\mathit{renSM}(\langle (p,q), \overline{\texttt{R}}\rangle\ \overline{\texttt{sp}}, \texttt{m}, \texttt{m}') \triangleq \langle (p,q), \mathit{renRM}(\overline{\texttt{R}}, \texttt{m}, \texttt{m}')\rangle\ \mathit{renSM}(\overline{\texttt{sp}}, \texttt{m}, \texttt{m}')$$

$$\mathit{renRM}(\emptyset, \texttt{m}, \texttt{m}') \triangleq \emptyset$$

$$\mathit{renRM}(\texttt{n} : (r,s)\ \overline{\texttt{R}}, \texttt{m}, \texttt{m}') \triangleq \texttt{n}[\texttt{m}'/\texttt{m}] : (r,s)\ \mathit{renRM}(\overline{\texttt{R}}, \texttt{m}, \texttt{m}')$$

$$\mathit{renSF}(\varepsilon, \texttt{f}, \texttt{f}') \triangleq \varepsilon$$

$$\mathit{renSF}(\langle (p,q), \overline{\texttt{R}}\rangle\ \overline{\texttt{sp}}, \texttt{f}, \texttt{f}') \triangleq \langle (p[\texttt{f}'/\texttt{f}], q[\texttt{f}'/\texttt{f}]), \mathit{renRF}(\overline{\texttt{R}}, \texttt{f}, \texttt{f}')\rangle\ \mathit{renSF}(\overline{\texttt{sp}}, \texttt{f}, \texttt{f}')$$

$$\mathit{renRF}(\emptyset, \texttt{f}, \texttt{f}') \triangleq \emptyset$$

$$\mathit{renRF}(\texttt{n} : (r,s)\ \overline{\texttt{R}}, \texttt{f}, \texttt{f}') \triangleq \texttt{n} : (r[\texttt{f}'/\texttt{f}], s[\texttt{f}'/\texttt{f}])\ \mathit{renRF}(\overline{\texttt{R}}, \texttt{f}, \texttt{f}')$$

$$\mathit{renMF}(\texttt{I n}(\overline{\texttt{I}}\ \overline{\texttt{x}})\{t\}, \texttt{f}, \texttt{f}') \triangleq \texttt{I n}(\overline{\texttt{I}}\ \overline{\texttt{x}})\{t[\texttt{f}'/\texttt{f}]\}$$

$$\mathit{discard}(\varepsilon, \overline{\texttt{f}}) \triangleq \varepsilon$$

$$\mathit{discard}(\texttt{sp}\ \overline{\texttt{sp}}, \overline{\texttt{f}}) \triangleq \begin{cases} \mathit{discard}(\overline{\texttt{sp}}, \overline{\texttt{f}}) & \textbf{if } \overline{\texttt{f}} \cap \mathit{vars}(\texttt{sp}) \neq \emptyset \\ \texttt{sp}\ \mathit{discard}(\overline{\texttt{sp}}, \overline{\texttt{f}}) & \textbf{otherwise} \end{cases}$$

Fig. 13. The definition of $\mathscr{T} \oplus (\texttt{TAE}, \texttt{ao})$ with associated auxiliary functions.

[SDNB03] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, volume 2743 of *Lecture Notes in Computer Science*, pages 248–274. Springer, 2003.

[Tra11] TraitRecordJ website, May 2011. `http://traitrecordj.sourceforge.net/`.

[TSAH12] Thomas Thüm, Ina Schaefer, Sven Apel, and Martin Hentschel. Family-based deductive verification of software product lines. In *International Conference on Generative Programming and Component Engineering (GPCE)*, pages 11–20. ACM Press, 2012.

[TSKA11] Thomas Thüm, Ina Schaefer, Martin Kuhlemann, and Sven Apel. Proof composition for deductive verification of software product lines. In *Proc. Intl. Workshop on Variability-intensive Systems Testing, Validation and Verification*, pages 270–277. IEEE Computer Society Press, 2011.

[vSC10] Stephan van Staden and Cristiano Calcagno. Reasoning about multiple related abstractions with MultiStar. In *Proc. Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 504–519. ACM Press, 2010.

[Wil91] Alan Wills. Capsules and types in Fresco: Program verification in Smalltalk. In Pierre America, editor, *Proc. European Conference on Object-Oriented Programming (ECOOP)*, volume 512 of *Lecture Notes in Computer Science*, pages 59–76. Springer, 1991.

A. Auxiliary Functions for Extending the Trait Environment

This section details the auxiliary functions used to update the trait environment for trait alteration expressions in the rule EXTEND of the proof system for trait analysis given in Section 7.2. The auxiliary functions are defined in Figure 13. The update function $\mathscr{T} \oplus (\texttt{TAE}, \texttt{ao})$ creates an entry for `TAE ao` in $\mathscr{T}$ by *modifying* the existing entry for `TAE`. For a set $\overline{\texttt{M}}$ of method definitions, let the functions *fields*$(\overline{\texttt{M}})$ and *calls*$(\overline{\texttt{M}})$ return the required fields and method signatures of $\overline{\texttt{M}}$, respectively. These functions are straightforwardly lifted to annotated methods by ignoring specifications (e.g., *fields*$(\texttt{M}\ \overline{\texttt{sp}}) \triangleq$ *fields*$(\overline{\texttt{M}})$) and to sets of annotated methods. For a set $\overline{\texttt{H}}$ of method signatures, let $\overline{\texttt{H}} \setminus \texttt{m}$ return the set $\overline{\texttt{H}}$ except the signature for `m`. Without loss of generality, we may assume that there are no name clashes between fields and locally defined variables in methods. Let *assign*$(\texttt{M})$ denote the subset of *fields*$(\texttt{M})$ to which a method `M` assigns values (the write-set of `M`). Given a method specification `sp`, let *vars*$(\texttt{sp})$ denote the set of free variables in `sp`.

Trait alteration operations construct a new trait expression from an old trait expression. The modification function *mod* is defined by cases for the different trait alteration operations. For the operation [**`exclude`** `m`], the new trait expression is obtained by removing the definition and specifications of `m`, if present, from the old trait expression. For the operation [`m` **`aliasAs`** `m'`], the new trait expression is obtained by duplicating the definition of `m`, if defined, and giving the copy the new name `m'`. The new method `m'` has the same specifications as the old method. For the [`m` **`renameTo`** `m'`] operation, the new trait expression is obtained by a consistent renaming in all method definitions and requirements.

As discussed in Section 5, field renaming requires special care since a direct substitution may invalidate specifications. For the operation [`f` **`renameTo`** `f'`], this is reflected in the definition of *renFld*$(\texttt{M}\ \overline{\texttt{sp}}, \texttt{f}, \texttt{f}')$ which applies to each method in the old trait expression. If both `f` and `f'` occur in the method body, the new method is obtained by *discarding* all previous specifications, except those that do not contain variables occurring in `f`, `f'`, and the write-set of `M`. Thus, the remaining specifications are neither affected by the substitution nor by the execution of `M`. This is a strong restriction on the specifications, capturing that the behavior of the method has changed. Note that it is not a sufficient requirement that the specification itself does not contain `f` or `f'` in order to be preserved; for example $\{\mathit{true}\}\ \texttt{f} := 5; \texttt{f}' := 6; \texttt{u} := \texttt{f}\ \{u == 5\}$ becomes invalid if `f` is replaced by `f'`. The second line in the definition of *renFld*$(\texttt{M}\ \overline{\texttt{sp}}, \texttt{f}, \texttt{f}')$ covers the case where the new field `f'` does not occur in the method body. In this case, specifications which contain the new field `f'` are discarded to avoid name clashes. In the remaining specifications, all occurrences of `f` are renamed. The **otherwise** case covers the situation $\texttt{f}' \in \mathit{fields}(\texttt{M}) \wedge \texttt{f} \notin \mathit{fields}(\texttt{M})$. In this case, the old field `f` does not occur in the method, which means that the method definition itself is not changed. To avoid name clashes, we then discard the specifications which contain `f`. The soundness of $\mathscr{T} \oplus (\texttt{TAE}, \texttt{ao})$ follows by Lemma B.2. (A more fine-grained analysis of field renaming may be possible in program logics *PL* which support variables-as-resources [PBC06].)

The auxiliary function *renSM*$(\overline{\texttt{sp}}, \texttt{m}, \texttt{m}')$ renames the method `m` in $\overline{\texttt{sp}}$. Since method names do not occur in assertions, this function is defined in terms of *renRM* which replaces each requirement $\texttt{n} : (r, s)$ by $\texttt{n}[\texttt{m}'/\texttt{m}] : (r, s)$. Similarly, the function *renSF*$(\overline{\texttt{sp}}, \texttt{f}, \texttt{f}')$ renames the field `f` in $\overline{\texttt{sp}}$, and is defined in terms of *renRF* which renames fields in requirements. The function *renMF* renames fields in method definitions, and the function *discard*$(\overline{\texttt{sp}}, \overline{\texttt{f}})$ returns $\overline{\texttt{sp}}$ except specifications that contain variables in $\overline{\texttt{f}}$.

B. Soundness of PST(PL)

Appendix B.1 presents the proof for the soundness of PST(PL). The main soundness result is formalized by Theorem 7.4, which expresses that the verification of a class leads to a consistent set of specifications for that class. Appendix B.2 contains auxiliary lemmas used for the soundness proof. Especially, the proof of Lemma B.2 ensures that the trait environment $\mathscr{T}$ is sound after the successful analysis of the basic trait declarations; i.e., there is a valid proof outline for all specifications recorded in $\mathscr{T}$ during the analysis of traits and classes. We here assume that trait and class environments have been constructed from the empty trait and class environments by application of the rules of PST(PL).

B.1. Main Theorem

Theorem 7.4. *For a given class* **class** C **implements** $\overline{\mathtt{I}}$ **by** $\{\ \overline{rp_I}, \overline{\mathtt{F}};\ \}$ **and** CTE*, if the successful analysis of* C *in* PST(PL) *leads to a class environment* $\mathscr{C}$*, then the set of method specifications for* C *in* $\mathscr{C}$ *are consistent for the flattened version of* C.

Proof. We first check Condition 2 in Definition 7.3. For each $\mathtt{m} \in \mathit{mtds}(\mathtt{CTE})$, we consider the set $S_{\mathscr{C}}(\mathtt{C},\mathtt{m})$. Let $\mathtt{CTE} \triangleq \mathtt{TAE}_1 + \ldots + \mathtt{TAE}_n$. By type safety, there must be exactly one i (for $1 \leq i \leq n$) such that $m \in \mathit{mtds}(\mathscr{T}(\mathtt{TAE}_i))$. By Lemma B.6 we know that $S_{\mathscr{C}}(\mathtt{C},\mathtt{m}) \subseteq \mathit{specs}(\mathscr{T}(\mathtt{TAE}_i)(m))$. Let $\langle (p,q), \overline{\mathtt{R}} \rangle \in S_{\mathscr{C}}(\mathtt{C},\mathtt{m})$. By Lemma B.2, there exists a proof outline O for the method body of m with guarantee (p,q) such that $\overline{\mathtt{R}} \twoheadrightarrow \mathit{reqs}(O)$. For each $\mathtt{n} : (r,s) \in \overline{\mathtt{R}}$, we prove $\mathit{guar}(S_{\mathscr{C}}(\mathtt{C},\mathtt{n})) \twoheadrightarrow (r,s)$ by induction over the inference rules of PST(PL).

To include $\langle (p,q), \overline{\mathtt{R}} \rangle$ in $S_{\mathscr{C}}(\mathtt{C},\mathtt{m})$, rule ANALYZE must be applied, which again generates a *discharge* operation for a set of requirements which includes $\overline{\mathtt{R}}$. Repeatedly applying rule DECOMP4 results in a $\mathit{discharge}(\mathtt{n} : (r,s))$ operation, which is analyzed by either OPENANALYSIS or CLOSEANALYSIS. If OPENANALYSIS is applied, we get an operation $\mathit{analyze}(\mathtt{TAE}_j, \mathtt{n} : (r,s))$ where j is such that $\mathtt{n} \in \mathit{mtds}(\mathscr{T}(\mathtt{TAE}_j))$. This operation must again be handled by the ANALYZE rule, which ensures the entailment $\mathit{guar}(S_{\mathscr{C}}(\mathtt{C},\mathtt{n})) \twoheadrightarrow (r,s)$. Before ANALYZE can be applied in the proof, INCREMENT may be needed to introduce the necessary specifications $\overline{\mathtt{sp}}$ into the trait environment to ensure that $\mathit{guar}(\overline{\mathtt{sp}}) \twoheadrightarrow (r,s)$. The application of ANALYZE again generates a *discharge* operation for a set of requirements which includes $\mathit{req}(\overline{\mathtt{sp}})$. Since the analysis of C succeeds, the repeated application of ANALYZE must eventually terminate. If CLOSEANALYSIS is applied, the conclusion $\mathit{guar}(S_{\mathscr{C}}(\mathtt{C},\mathtt{n})) \twoheadrightarrow (r,s)$ follows directly.

For Condition 1 in Definition 7.3, we have by the rules CLASS and DECOMP4 that a $\mathit{discharge}(\mathtt{m} : (r,s))$ operation is analyzed for each $\mathtt{m} : (r,s) \in \mathit{contracts}(\overline{rp_I}, \overline{\mathtt{I}})$. By an argument corresponding to the one for *discharge* above, we get $S_{\mathscr{C}}(\mathtt{C},\mathtt{m}) \twoheadrightarrow (r,s)$.

It follows from Lemma B.5 that the sets of method specifications $S_{\mathscr{C}}(\mathtt{C},\mathtt{m})$ for all $\mathtt{m} \in \mathit{mtds}(\mathtt{CTE})$ constitute a consistent set of method specifications for the flattened version of C. □

B.2. Auxiliary Lemmas

Lemma B.1. Let the trait alteration expression TAE be defined by $\mathtt{Tb}\ \overline{\mathtt{ao}}$. If $\mathtt{TAE} \in \mathscr{T}$, then for any (possibly empty) prefix $\overline{\mathtt{ao}}'$ of $\overline{\mathtt{ao}}$, we also have $\mathtt{Tb}\ \overline{\mathtt{ao}}' \in \mathscr{T}$.

Proof. The proof is by induction over the length of the list $\overline{\mathtt{ao}}$ of alteration operations. If $\overline{\mathtt{ao}}$ is empty, then TAE is a basic trait Tb which is introduced in $\mathscr{T}$ by rule BASICTRAIT. For the induction step, consider $\mathtt{TAE}\ \mathtt{ao}$, where $\mathtt{TAE} = \mathtt{Tb}\ \overline{\mathtt{ao}}$. Since $\mathtt{TAE}\ \mathtt{ao}$ must have been introduced by rule EXTEND, we have that $\mathtt{TAE} \in \mathscr{T}$, i.e., we have $\mathtt{Tb}\ \overline{\mathtt{ao}} \in \mathscr{T}$. By the induction hypothesis, $\mathtt{Tb}\ \overline{\mathtt{ao}}' \in \mathscr{T}$ for all prefixes $\overline{\mathtt{ao}}'$ of $\overline{\mathtt{ao}}$. □

Lemma B.2 (Soundness of $\mathscr{T}$). Let $\mathscr{T}$ be a trait environment such that all basic traits in $\mathscr{T}$ are successfully analyzed. Then the following holds for all $\mathtt{TAE} \in \mathscr{T}$:

$$\forall (\mathtt{M}\ \overline{\mathtt{sp}}) \in \mathscr{T}(\mathtt{TAE})\ .\ \forall \langle (p,q), \overline{\mathtt{R}} \rangle \in \overline{\mathtt{sp}}\ .\ \exists O\ .\ O \vdash_{PL} \mathit{body}(\mathtt{M}) : (p,q)\ \wedge\ \overline{\mathtt{R}} \twoheadrightarrow \mathit{reqs}(O)$$

Proof. The proof is by induction over the structure of TAE. By Lemma B.1 each prefix of TAE is also in $\mathscr{T}$.

Base case: $\mathtt{TAE} = \mathtt{Tb}$. Then $\langle (p,q), \overline{\mathtt{R}} \rangle$ is included in $\overline{\mathtt{sp}}$ by either VERIFY or BASICTRAIT. If rule VERIFY is applied, the lemma follows immediately from the premise of the rule. The application of rule BASICTRAIT, possibly followed by a number of DECOMP1 and DECOMP2 applications, results in an operation $\mathit{verify}(\mathtt{Tb}, \mathtt{M}\langle (p,q), \overline{\mathtt{R}} \rangle)$ for every specification $\langle (p,q), \overline{\mathtt{R}} \rangle$ of some M in Tb. This operation is analyzed by VERIFY after zero or more applications of ADAPTATION. We close this case of the proof by induction over the number of applications of ADAPTATION. For zero applications, the lemma follows directly from the application of VERIFY. For the induction step, we assume a specification $\langle (p',q'), \overline{\mathtt{R}}' \rangle$ such that $\langle (p',q'), \overline{\mathtt{R}}' \rangle \twoheadrightarrow \langle (p,q), \overline{\mathtt{R}} \rangle$ and $O' \vdash_{PL} \mathit{body}(\mathtt{M}) : (p',q')$ and $\overline{\mathtt{R}}' \twoheadrightarrow \mathit{reqs}(O')$ for some proof outline O'. By Lemma 4.7, there is a proof outline O such that $O \vdash_{PL} \mathit{body}(\mathtt{M}) : (p,q)$ and $\overline{\mathtt{R}} \twoheadrightarrow \mathit{reqs}(O)$.

Induction step: $\mathtt{TAE} = \mathtt{TAE}'\ \mathtt{ao}$. By Lemma B.1, $\mathtt{TAE}' \in \mathscr{T}$. Let $\mathscr{T}(\mathtt{TAE}') = \{\overline{\mathtt{F}}; \overline{\mathtt{H}}; \overline{\mathtt{AM}}\}$. The induction hypothesis *IH* assumes proof outlines for these methods; i.e., $\forall (\mathtt{M}\ \overline{\mathtt{sp}}) \in \overline{\mathtt{AM}}\ .\ \forall \langle (p,q), \overline{\mathtt{R}} \rangle \in \overline{\mathtt{sp}}\ .\ \exists O\ .\ O \vdash_{PL} \mathit{body}(\mathtt{M}) : (p,q) \wedge \overline{\mathtt{R}} \twoheadrightarrow \mathit{reqs}(O)$.

The only rules that extend $\mathscr{T}$ with a new specification for non-basic traits are VERIFY and EXTEND. If $\langle (p,q), \overline{\mathtt{R}} \rangle$ is included in the trait environment by VERIFY, the conclusion follows immediately. If EXTEND is applied, the trait

environment is extended to $\mathscr{T}[\texttt{TAE} \mapsto \mathit{mod}(\mathscr{T}(\texttt{TAE}'), \texttt{ao})]$, where the definition of *mod* can be found in Figure 13. The different cases for ao ensure the conditions of the lemma as follows:

- $\texttt{ao} = [\textbf{exclude}\ \texttt{m}]$. Here $\mathscr{T}(\texttt{TAE})$ contains the set $\overline{\texttt{AM}}' = \overline{\texttt{AM}} \setminus \overline{\texttt{AM}}(\texttt{m})$. Proof outlines for $\overline{\texttt{AM}}'$ follow from *IH*.
- $\texttt{ao} = [\texttt{m}\ \textbf{aliasAs}\ \texttt{m}']$. Then $\mathscr{T}(\texttt{TAE})$ contains the annotated methods $\overline{\texttt{AM}}$, and proof outlines for these specifications follow from *IH*. The trait $\mathscr{T}(\texttt{TAE})$ contains additional specifications if $\texttt{m} \in \overline{\texttt{AM}}$. Let $\overline{\texttt{AM}}(\texttt{m}) = \texttt{I m}(\overline{\texttt{I x}})\{t\}\ \overline{\texttt{sp}}$. Then $\texttt{I m}'(\overline{\texttt{I x}})\{t\}\ \overline{\texttt{sp}}$ is included in $\mathscr{T}(\texttt{TAE})$. Proof outlines for $\overline{\texttt{sp}}$ follow directly from *IH*.
- $\texttt{ao} = [\texttt{m}\ \textbf{renameTo}\ \texttt{m}']$. For each $\texttt{I n}(\overline{\texttt{I x}})\{t\}\ \overline{\texttt{sp}}$ in $\overline{\texttt{AM}}$, we have by Figure 13 that the following method is included in $\mathscr{T}(\texttt{TAE})$: $\texttt{I n}[\texttt{m}'/\texttt{m}](\overline{\texttt{I x}})\{t[\texttt{m}/\texttt{m}']\}$ $\mathit{renSM}(\overline{\texttt{sp}}, \texttt{m}, \texttt{m}')$. For each $\langle (p,q), \overline{\texttt{R}} \rangle \in \overline{\texttt{sp}}$, we have by *IH* that there is a proof outline O for t such that $O \vdash_{PL} t : (p,q)$ where $\overline{\texttt{R}} \twoheadrightarrow \mathit{reqs}(O)$. From the definition of *renSM* in Figure 13, the specification $\langle (p,q), \mathit{renRM}(\overline{\texttt{R}}, \texttt{m}, \texttt{m}') \rangle$ is included in $\mathscr{T}(\texttt{TAE})(\texttt{n}[\texttt{m}'/\texttt{m}])$. From the definition of *renRM* we observe that if $\overline{\texttt{R}} \twoheadrightarrow \mathit{reqs}(O)$, then $\mathit{renRM}(\overline{\texttt{R}}, \texttt{m}, \texttt{m}') \twoheadrightarrow \mathit{renRM}(\mathit{reqs}(O), \texttt{m}, \texttt{m}')$. We construct a proof outline O' for $t[\texttt{m}'/\texttt{m}]$ by replacing each call $\{r\}\ \texttt{m}\ \{s\}$ in O by $\{r\}\ \texttt{m}'\ \{s\}$, i.e, $\mathit{reqs}(O') = \mathit{renRM}(\mathit{reqs}(O), \texttt{m}, \texttt{m}')$. The desired conclusion $\mathit{renRM}(\overline{\texttt{R}}, \texttt{m}, \texttt{m}') \twoheadrightarrow \mathit{reqs}(O')$ then follows since $\overline{\texttt{R}} \twoheadrightarrow \mathit{reqs}(O)$.
- $\texttt{ao} = [\texttt{f}\ \textbf{renameTo}\ \texttt{f}']$. For each method $\texttt{I m}(\overline{\texttt{I x}})\{t\}\ \overline{\texttt{sp}}$ in $\overline{\texttt{AM}}$, the modified method $\mathit{renFld}(\texttt{I m}(\overline{\texttt{I x}})\{t\}\ \overline{\texttt{sp}}, \texttt{f}, \texttt{f}')$ is included in $\mathscr{T}(\texttt{TAE})$. The different cases in the definition of this function are considered separately.
  - *Case* $\texttt{f} \in \mathit{fields}(t) \wedge \texttt{f}' \in \mathit{fields}(t)$. In this case, f is replaced by f′ in the method body t and all specifications which contain f, f′, or variables in $\mathit{assign}(t)$ are discarded. Thus, if a specification $\langle (p,q), \overline{\texttt{R}} \rangle$ is included, we have $\mathit{vars}\langle (p,q), \overline{\texttt{R}} \rangle \cap \{\texttt{f}, \texttt{f}', \mathit{assign}(t)\} = \emptyset$. By *IH*, there is a proof outline O such that $O \vdash_{PL}\ t : (p,q)$ where $\overline{\texttt{R}} \twoheadrightarrow \mathit{reqs}(O)$. It then follows that O is a proof outline for $t[\texttt{f}'/\texttt{f}]$ since O is valid for t. Especially, for some statement t_1 not containing method calls, if $\{s\}\ t_1\ \{r\}$ is valid for some s and r such that $\mathit{vars}(s,r) \cap \{\texttt{f}, \texttt{f}', \mathit{assign}(t_1)\} = \emptyset$ then $\{s\}\ t_1[\texttt{f}'/\texttt{f}]\ \{r\}$ is also valid.
  - *Case* $\texttt{f}' \notin \mathit{fields}(t)$. The new method body is obtained by renaming f to f′. For each specification $\langle (p,q), \overline{\texttt{R}} \rangle \in \overline{\texttt{sp}}$, the modified specification $\mathit{renSF}(\langle (p,q), \overline{\texttt{R}} \rangle, \texttt{f}, \texttt{f}')$ is added to $\mathscr{T}(\texttt{TAE})$ if $\texttt{f}' \notin \mathit{vars}(\{p,q,\overline{\texttt{R}}\})$. By *IH* there is a proof outline O such that $O \vdash_{PL} t : (p,q)$ and $\overline{\texttt{R}} \twoheadrightarrow \mathit{reqs}(O)$. Since f′ does not occur in O, a proof outline for $t[\texttt{f}'/\texttt{f}]$ can be constructed directly by field renaming, resulting in the specification $\mathit{renSF}(\langle (p,q), \overline{\texttt{R}} \rangle, \texttt{f}, \texttt{f}')$.
  - *Case* $\texttt{f}' \in \mathit{fields}(t) \wedge \texttt{f} \notin \mathit{fields}(t)$. In this case, the method body is unaltered by the rename operation. However, we discard specifications containing the renamed field f. Proof outlines for the remaining specifications follow directly by *IH*.

□

Observation B.3. Let $\mathit{mtds}(\texttt{AM}) = \overline{\texttt{M}}$ and $\texttt{M} = \texttt{I n}(\overline{\texttt{I x}})\{t\}$. The following properties follow from the definitions in Figure 13 and Figure 4:

1. $\mathit{mtds}(\mathit{remMtd}(\overline{\texttt{AM}}, \texttt{m})) = \mathit{mtds}(\overline{\texttt{AM}} \setminus \overline{\texttt{AM}}(\texttt{m})) = \overline{\texttt{M}} \setminus \overline{\texttt{M}}(\texttt{m}) = \mathit{rem}(\overline{\texttt{M}}, \texttt{m}) = \mathit{rem}(\mathit{mtds}(\overline{\texttt{AM}}), \texttt{m})$
2. $\mathit{mtds}(\mathit{renId}(\texttt{M}\ \overline{\texttt{sp}}, \texttt{m}, \texttt{m}')) = \mathit{mtds}(\texttt{I n}[\texttt{m}'/\texttt{m}](\overline{\texttt{I x}})\{t\}\ \overline{\texttt{sp}}) = \texttt{I n}[\texttt{m}'/\texttt{m}](\overline{\texttt{I x}})\{t\} = \mathit{ren}(\texttt{M}, \texttt{m}, \texttt{m}')$
3. $\mathit{mtds}(\mathit{renMtd}(\overline{\texttt{AM}}, \texttt{m}, \texttt{m}')) = \mathit{rep}(\mathit{mtds}(\overline{\texttt{AM}}), \texttt{m}, \texttt{m}')$
4. $\mathit{mtds}(\mathit{renFld}(\overline{\texttt{AM}}, \texttt{f}, \texttt{f}')) = \mathit{rep}(\mathit{mtds}(\overline{\texttt{AM}}), \texttt{f}, \texttt{f}')$

Property 3 above follows by induction over the size of $\overline{\texttt{AM}}$ with the base case $\mathit{mtds}(\mathit{FSrenMtd}(\emptyset, \texttt{m}, \texttt{m}')) = \emptyset = \mathit{rep}(\emptyset, \texttt{m}, \texttt{m}')$ and the induction step

$$\begin{array}{l} \mathit{mtds}(\mathit{renMtd}(\overline{\texttt{AM}}\ \texttt{I n}(\overline{\texttt{I x}})\{t\}\ \overline{\texttt{sp}}, \texttt{m}, \texttt{m}')) = \\ \mathit{mtds}(\mathit{renMtd}(\overline{\texttt{AM}}, \texttt{m}, \texttt{m}'))\ \mathit{mtds}(\texttt{I n}[\texttt{m}'/\texttt{m}](\overline{\texttt{I x}})\{t[\texttt{m}'/\texttt{m}]\}\ \overline{\texttt{sp}}) = \\ \mathit{rep}(\mathit{mtds}(\overline{\texttt{AM}}), \texttt{m}, \texttt{m}')\ \texttt{I n}[\texttt{m}'/\texttt{m}](\overline{\texttt{I x}})\{t[\texttt{m}'/\texttt{m}]\} = \mathit{rep}(\mathit{mtds}(\overline{\texttt{AM}}\ \texttt{I n}(\overline{\texttt{I x}})\{t\}\ \overline{\texttt{sp}}), \texttt{m}, \texttt{m}'). \end{array}$$

Property 4 follows by a similar argument.

Lemma B.4. Given a trait expression TAE such that $\texttt{TAE} \in \mathscr{T}$, then $\mathit{mtds}(\mathscr{T}(\texttt{TAE})) = [\![\texttt{TAE}]\!]$.

Proof. The proof is by induction over the structure of TAE.

Base case: TAE is a basic trait $\textbf{trait}\ \texttt{Tb}\ \textbf{is}\ \{\overline{\texttt{F}}; \overline{\texttt{H}}; \overline{\texttt{AM}}\}$ be the definition of Tb. Let $\overline{\texttt{M}} = \mathit{mtds}(\overline{\texttt{AM}})$. This trait is introduced in $\mathscr{T}$ by rule BASICTRAIT. By induction over the inference rules, we then know that $\mathscr{T}(\texttt{Tb}) = \{\overline{\texttt{F}}; \overline{\texttt{H}}; \overline{\texttt{AM}}'\}$ for some $\overline{\texttt{AM}}'$ such that $\mathit{mtds}(\overline{\texttt{AM}}') = \mathit{mtds}(\overline{\texttt{AM}})$, since no rules add new method definitions to a trait in $\mathscr{T}$. (However, $\overline{\texttt{AM}}'$ may contain more specifications than $\overline{\texttt{AM}}$.) From the definition of flattening for a basic trait (Figure 3), we get

$$\mathit{mtds}(\mathscr{T}(\texttt{Tb})) = \mathit{mtds}(\overline{\texttt{AM}}') = \overline{\texttt{M}} \overset{\textit{Figure 3}}{=} [\![\{\overline{\texttt{F}}; \overline{\texttt{H}}; \overline{\texttt{M}}\}]\!] \overset{\textit{Figure 3}}{=} [\![\texttt{Tb}]\!].$$

Induction step: $\texttt{TAE} = \texttt{TAE}'\ \texttt{ao}$. Since $\texttt{TAE} \in \mathscr{T}$, we know from Lemma B.1 that $\texttt{TAE}' \in \mathscr{T}$. Assume that $\mathscr{T}(\texttt{TAE}') = \{\overline{\texttt{F}}; \overline{\texttt{H}}; \overline{\texttt{AM}}\}$, so $\mathit{mtds}(\mathscr{T}(\texttt{TAE}')) = \mathit{mtds}(\overline{\texttt{AM}})$. The induction hypothesis *IH* is that $\mathit{mtds}(\mathscr{T}(\texttt{TAE}')) = [\![\texttt{TAE}']\!]$, so, $[\![\texttt{TAE}']\!] = \mathit{mtds}(\overline{\texttt{AM}})$. The different trait alteration operations are treated as separate cases:

- $\texttt{ao} = [\textbf{exclude}\ \texttt{m}]$:

$$\mathit{mtds}(\mathscr{T}(\texttt{TAE})) \overset{\textit{Figure 13}}{=} \mathit{mtds}(\mathit{remMtd}(\overline{\texttt{AM}}, \texttt{m})) \overset{\textit{Obs. B.3}}{=} \mathit{rem}(\mathit{mtds}(\overline{\texttt{AM}}), \texttt{m}) \overset{\textit{IH}}{=} \mathit{rem}([\![\texttt{TAE}']\!], \texttt{m}) \overset{\textit{Figure 3}}{=} [\![\texttt{TAE}'[\textbf{exclude}\ m]]\!] = [\![\texttt{TAE}]\!]$$

- $\texttt{ao} = [\texttt{m}\ \textbf{aliasAs}\ \texttt{m}']$. Case $\texttt{m} \notin \overline{\texttt{AM}}$:

$$\mathit{mtds}(\mathscr{T}(\texttt{TAE})) \overset{\textit{Figure 13}}{=} \mathit{mtds}(\overline{\texttt{AM}}) \overset{\textit{IH}}{=} [\![\texttt{TAE}']\!] \overset{\textit{Figure 3}}{=} [\![\texttt{TAE}'[\texttt{m}\ \textbf{aliasAs}\ \texttt{m}']]\!] = [\![\texttt{TAE}]\!]$$

  Case $\texttt{m} \in \overline{\texttt{AM}}$:

$$\mathit{mtds}(\mathscr{T}(\texttt{TAE})) \overset{\textit{Figure 13}}{=} \mathit{mtds}(\overline{\texttt{AM}})\ \mathit{mtds}(\mathit{renId}(\overline{\texttt{AM}}(\texttt{m}), \texttt{m}, \texttt{m}')) \overset{\textit{Obs. B.3}}{=} \mathit{mtds}(\overline{\texttt{AM}})\ \mathit{ren}(\overline{\texttt{AM}}(\texttt{m}), \texttt{m}, \texttt{m}') \overset{\textit{IH}}{=} [\![\texttt{TAE}']\!]\ \mathit{ren}([\![\texttt{TAE}']\!](\texttt{m}), \texttt{m}, \texttt{m}') \overset{\textit{Figure 3}}{=} [\![\texttt{TAE}[\texttt{m}\ \textbf{aliasAs}\ \texttt{m}']]\!] = [\![\texttt{TAE}]\!]$$

- $\texttt{ao} = [\texttt{m}\ \textbf{renameTo}\ \texttt{m}']$:

$$\mathit{mtds}(\mathscr{T}(\texttt{TAE})) \overset{\textit{Figure 13}}{=} \mathit{mtds}(\mathit{renMtd}(\overline{\texttt{AM}}, \texttt{m}, \texttt{m}')) \overset{\textit{Obs. B.3}}{=} \mathit{rep}(\mathit{mtds}(\overline{\texttt{AM}}), \texttt{m}, \texttt{m}') \overset{\textit{IH}}{=} \mathit{rep}([\![\texttt{TAE}']\!], \texttt{m}, \texttt{m}') \overset{\textit{Figure 3}}{=} [\![\texttt{TAE}'[\texttt{m}\ \textbf{renameTo}\ \texttt{m}']]\!] = [\![\texttt{TAE}]\!]$$

- $\texttt{ao} = [\texttt{f}\ \textbf{renameTo}\ \texttt{f}']$:

$$\mathit{mtds}(\mathscr{T}(\texttt{TAE})) \overset{\textit{Figure 13}}{=} \mathit{mtds}(\mathit{renFld}(\overline{\texttt{AM}}, \texttt{f}, \texttt{f}')) \overset{\textit{Obs. B.3}}{=} \mathit{rep}(\mathit{mtds}(\overline{\texttt{AM}}), \texttt{f}, \texttt{f}') \overset{\textit{IH}}{=} \mathit{rep}([\![\texttt{TAE}']\!], \texttt{f}, \texttt{f}') \overset{\textit{Figure 3}}{=} [\![\texttt{TAE}'[\texttt{f}\ \textbf{renameTo}\ \texttt{f}']]\!] = [\![\texttt{TAE}]\!]$$

□

Lemma B.5. Let CTE be a composed trait expression defined by $\texttt{TAE}_1 + \ldots + \texttt{TAE}_n$, and assume that each $\texttt{TAE}_i \in \mathscr{T}$. Then $\mathit{mtds}(\texttt{CTE}) = [\![\texttt{CTE}]\!]$.

Proof. This follows directly from the definition of $\mathit{mtds}(\texttt{CTE})$ together with Lemma B.4 and the definition of flattening (Section 3.2).

$$\mathit{mtds}(\texttt{CTE}) \triangleq \bigcup_{1 \le i \le n} \mathit{mtds}(\mathscr{T}(\texttt{TAE}_i)) \overset{\textit{Lemma B.4}}{=} \bigcup_{1 \le i \le n} [\![\texttt{TAE}_i]\!] \overset{\textit{Figure 3}}{=} [\![\texttt{CTE}]\!]$$

□

Lemma B.6. Consider $\textbf{class}\ \texttt{C}\ \textbf{implements}\ \overline{\texttt{I}}\ \textbf{by}\ \{\ \overline{rp_I}\ \overline{\texttt{F}};\ \}\ \textbf{and}\ \texttt{CTE}$ such that $\texttt{CTE} \triangleq \texttt{TAE}_1 + \ldots + \texttt{TAE}_n$. Let the environments $\mathscr{T}$ and $\mathscr{C}$ result from the successful analysis of this class. For $\texttt{m} \in \mathit{mtds}(\texttt{TAE}_i)$ and $1 \le i \le n$, we have

$$S_{\mathscr{C}}(\texttt{C}, \texttt{m}) \subseteq \mathit{specs}(\mathscr{T}(\texttt{TAE}_i)(m)).$$

Proof. By type safety, each method in CTE is defined in exactly one $\texttt{TAE}_i$. The relation $S_{\mathscr{C}}(\texttt{C}, \texttt{m}) \subseteq \mathit{specs}(\mathscr{T}(\texttt{TAE}_i)(m))$ follows by induction over the inference rules of PST(PL). When the analysis of class C starts by rule CLASS, the mapping $S_{\mathscr{C}}(\texttt{C}, \texttt{m})$ is initially empty, so the subset relation holds initially. Since $S_{\mathscr{C}}(\texttt{C}, \texttt{m})$ is only extended by ANALYZE, it suffices to consider this rule. The rule maintains the relation since if $S_{\mathscr{C}}(\texttt{C}, \texttt{m})$ is extended by some set $\overline{\texttt{sp}}$ by the rule, it follows from the premises that $\overline{\texttt{sp}} \subseteq \mathit{specs}(\mathscr{T}(\texttt{TAE}_i)(m))$. □

C. Verification of the Example

This section details the verification of the bank account classes from Section 8. In order to verify proof outlines, we must fix the program logic *PL*. In this example, we use a standard Hoare logic for sequential statements [AdBO09].

Since formal parameters are read-only in the example, we may use the following rule for analyzing method calls:

$$\text{(HOARECALL)}\quad \frac{\{p\}\ \mathtt{m}(\overline{\mathtt{x}})\ \{q\}}{\{p[\overline{e}/\overline{x}]\}\ \mathtt{v{=}m}(\overline{\mathtt{e}})\ \{q[\overline{e}/\overline{x}][v/\textbf{\textit{result}}]\}}$$

where $\overline{\mathtt{x}}$ are the formal parameters of the method m. Note that the annotations of a call v=m($\overline{\mathtt{e}}$) may be strengthened by assertions over logical and method local variables. Let $\mathscr{X}$ and $\mathscr{Z}$ denote the sets of method local variables (including formal parameters) and logical variables, respectively. If $\mathtt{m} : (p,q) \in \mathit{reqs}(O)$ is a requirement in some proof outline O, then O may contain decorated call statements of the form

$$\{p[\overline{e}/\overline{x}] \wedge c\}\ \mathtt{v{=}m}(\overline{\mathtt{e}})\ \{q[\overline{e}/\overline{x}][v/\textbf{\textit{result}}] \wedge c\}$$

where $\mathit{vars}(c) \subseteq \{\mathscr{X} \cup \mathscr{Z}\}$.

Initially in an analysis, the trait and class environments are empty. Let $\mathscr{T}_0$ be the trait environment resulting from the analysis of all the basic traits in Figure 8, so each basic trait is bound to an annotated basic trait expression in $\mathscr{T}_0$. These expressions correspond to the definitions in Figure 8, for example:

$$\mathscr{T}_0(\mathtt{TBasicUpd}) = \\ \{\ \textbf{int}\ \mathtt{bal};\ \emptyset;\ \textbf{void}\ \mathtt{update}(\textbf{nat}\ \mathtt{x})\{\mathtt{bal{=}bal{-}x}\}\ \langle (bal = b_0, bal = b_0 - x), \emptyset \rangle\ \}.$$

The only proof outlines which need to be supplied during the basic trait analysis are for this update method and for deposit in TBasicAccount. It is straightforward to provide these proof outlines given the guarantees.

We now consider the analysis of the class CFeeAccount in detail, assuming that the class is analyzed in the environment $\mathscr{C}_0, \mathscr{T}_0$, where $\mathtt{CFeeAccount} \notin \mathscr{C}_0$. Figure 14 shows the application of the proof system to the analysis of this class, leading to a successful analysis, where $\mathscr{C}_5, \mathscr{T}_3$ are the resulting environments.

In the first step, the class definition leads to the analysis of the trait expression TBasicAccount + CTE followed by discharge operations for the contracts of the interface IFeeAccount. The trait expression decomposes and as the basic traits are already analyzed, we get to the extend operation for the trait alteration expression TBasicUpd[update **rT** bUpdate]. The result is an extended trait environment $\mathscr{T}_1$ where a trait expression for the renaming of TBasicUpd is included in the trait environment. It is not necessary to reanalyze the renamed method, the specification is generated by renaming the existing method specification in TBasicUpd.

At this point, the discharge operation is at the head of the analysis operation sequence, and is decomposed. For the method deposit, ANALYZE can be applied and the specification found in TBasicAccount is included in the class environment, resulting in the class environment $\mathscr{C}_2$. Since neither withdraw nor update is specified in TBasicAccount and TFeeUpd, respectively, the analysis continues with OPENANALYSIS. Proof outlines for these methods must be provided at the level of their trait declarations by applying INCREMENT and checked by applying VERIFY. In these proof outlines, the requirement for the call to bUpdate follows from the generated specification of this method.

$$\mathcal{C}_5, \mathcal{T}_3 \vdash$$

(EmpClass)

$$\mathcal{C}_5, \mathcal{T}_3 \vdash \langle \texttt{CFeeAccount} : \varepsilon \rangle$$

(EmpDischarge)

$$\mathcal{C}_5, \mathcal{T}_3 \vdash \langle \texttt{CFeeAccount} : \textit{discharge}(\emptyset) \rangle$$

(Analyze) (14)

$$\mathcal{C}_4, \mathcal{T}_3 \vdash \langle \texttt{CFeeAccount} : \textit{analyze}(\texttt{TBasicUpd[update } \textbf{rT} \texttt{ bUpdate]}, \texttt{bUpdate} : (bal == b_0, bal == b_0 - x)) \rangle$$

(OpenAnalysis) (13)

$$\mathcal{C}_4, \mathcal{T}_3 \vdash \langle \texttt{CFeeAccount} : \textit{discharge}(\texttt{bUpdate} : (bal == b_0, bal == b_0 - x)) \rangle$$

(Analyze) (12)

$$\mathcal{C}_3, \mathcal{T}_3 \vdash \langle \texttt{CFeeAccount} : \textit{analyze}(\texttt{TFeeUpd}, \texttt{update} : (bal == b_0, bal == b_0 - x - fee)) \rangle$$

(Verify) (11)

$$\mathcal{C}_3, \mathcal{T}_2 \vdash \textit{verify}(\texttt{TFeeUpd}, \texttt{U}\ \langle (bal == b_0, bal == b_0 - x - fee), \texttt{bUpdaate} : (bal == b_0, bal == b_0 - x) \rangle) \cdot \langle \texttt{CFeeAccount} : \ldots \rangle$$

(Increment) (10)

$$\mathcal{C}_3, \mathcal{T}_2 \vdash \langle \texttt{CFeeAccount} : \textit{analyze}(\texttt{TFeeUpd}, \texttt{update} : (bal == b_0, bal == b_0 - x - fee)) \rangle$$

(OpenAnalysis) (9)

$$\mathcal{C}_3, \mathcal{T}_2 \vdash \langle \texttt{CFeeAccount} : \textit{discharge}(\texttt{update} : (bal == b_0, bal == b_0 - x - fee)) \rangle$$

(Analyze) (8)

$$\mathcal{C}_2, \mathcal{T}_2 \vdash \langle \texttt{CFeeAccount} : \textit{analyze}(\texttt{TBasicAccount}, \texttt{withdraw} : (bal == b_0, bal == b_0 - x - fee)) \rangle$$

(Verify) (7)

$$\mathcal{C}_2, \mathcal{T}_1 \vdash \textit{verify}(\texttt{TBasicAccount}, \texttt{W}\ \langle (bal == b_0, bal == b_0 - x - fee), \texttt{update} : (bal == b_0, bal == b_0 - x) - fee) \rangle) \cdot \langle \texttt{CFeeAccount} : \ldots \rangle$$

(Increment) (6)

$$\mathcal{C}_2, \mathcal{T}_1 \vdash \langle \texttt{CFeeAccount} : \textit{analyze}(\texttt{TBasicAccount}, \texttt{withdraw} : (bal == b_0, bal == b_0 - x - fee)) \rangle$$

(OpenAnalysis) (5)

$$\mathcal{C}_2, \mathcal{T}_1 \vdash \langle \texttt{CFeeAccount} : \textit{discharge}(\texttt{withdraw} : (bal == b_0, bal == b_0 - x - fee)) \rangle$$

(EmpDischarge)

$$\mathcal{C}_2, \mathcal{T}_1 \vdash \langle \texttt{CFeeAccount} : \textit{discharge}(\emptyset) \cdot \textit{discharge}(\texttt{withdraw} : (bal == b_0, bal == b_0 - x - fee)) \rangle$$

(Analyze) (4)

$$\mathcal{C}_1, \mathcal{T}_1 \vdash \langle \texttt{CFeeAccount} : \textit{analyze}(\texttt{TBasicAccount}, \texttt{deposit} : (bal == b_0, bal == b_0 + x)) \cdot \textit{discharge}(\texttt{withdraw} : (bal == b_0, bal == b_0 - x - fee)) \rangle$$

(OpenAnalysis) (3)

$$\mathcal{C}_1, \mathcal{T}_1 \vdash \langle \texttt{CFeeAccount} : \textit{discharge}(\texttt{deposit} : (bal == b_0, bal == b_0 + x)) \cdot \textit{discharge}(\texttt{withdraw} : (bal == b_0, bal == b_0 - x - fee)) \rangle$$

(Decomp4)

$$\mathcal{C}_1, \mathcal{T}_1 \vdash \langle \texttt{CFeeAccount} : \textit{discharge}(\texttt{deposit} : (bal == b_0, bal == b_0 + x) \cup \texttt{withdraw} : (bal == b_0, bal == b_0 - x - fee)) \rangle$$

(Extend) (2)

$$\mathcal{C}_1, \mathcal{T}_0 \vdash \textit{extend}(\texttt{TBasicUpd[update } \textbf{rT} \texttt{ bUpdate]}) \cdot \langle \texttt{CFeeAccount} : \ldots \rangle$$

(Lookup)

$$\mathcal{C}_1, \mathcal{T}_0 \vdash \textit{extend}(\texttt{TFeeUpd}) \cdot \textit{extend}(\texttt{TBasicUpd[update } \textbf{rT} \texttt{ bUpdate]}) \cdot \langle \texttt{CFeeAccount} : \ldots \rangle$$

(Decomp3)

$$\mathcal{C}_1, \mathcal{T}_0 \vdash \textit{extend}(\texttt{TFeeUpd + TBasicUpd[update } \textbf{rT} \texttt{ bUpdate]}) \cdot \langle \texttt{CFeeAccount} : \ldots \rangle$$

(Lookup)

$$\mathcal{C}_1, \mathcal{T}_0 \vdash \textit{extend}(\texttt{TBasicAccount}) \cdot \textit{extend}(\texttt{CTE}) \cdot \langle \texttt{CFeeAccount} : \ldots \rangle$$

(Decomp3)

$$\mathcal{C}_1, \mathcal{T}_0 \vdash \textit{extend}(\texttt{TBasicAccount} + \texttt{CTE}) \cdot \langle \texttt{CFeeAccount} : \textit{discharge}(\texttt{deposit} : (bal == b_0, bal == b_0 + x)) \cdot \textit{discharge}(\texttt{withdraw} : (bal == b_0, bal == b_0 - x - fee)) \rangle$$

(Class) (1)

$$\mathcal{C}_0, \mathcal{T}_0 \vdash \textbf{class}\ \texttt{CFeeAccount}\ \textbf{implements}\ \texttt{IFeeAccount}\ \textbf{by}\ \{\textbf{int}\ \texttt{bal},\ \textbf{nat}\ \texttt{fee}\}\ \textbf{and}\ \texttt{TBasicAccount} + \texttt{CTE}$$

Side conditions and auxiliary computations in the proof:

(1) $\mathcal{C}_1 = \langle D_{\mathcal{C}_0}[\texttt{CFeeAccount} \mapsto \langle \texttt{IFeeAccount}, \texttt{TBasicAccount} + \texttt{CTE}, \textbf{int}\ \texttt{bal},\ \textbf{nat}\ \texttt{fee} \rangle], S_{\mathcal{C}_0} \rangle$

(2) $\mathcal{T}_1 = \mathcal{T}_0[\texttt{TBasicUpd[update } \textbf{rT} \texttt{ bUpdate]} \mapsto \textit{mod}(\mathcal{T}_0(\texttt{TBasicUpd}), [\texttt{update } \textbf{rT} \texttt{ bUpdate}])]$
where $\textit{mod}(\mathcal{T}_0(\texttt{TBasicUpd}), [\texttt{update } \textbf{rT} \texttt{ bUpdate}])$ evaluates to:
$\{\textbf{int}\ \texttt{bal};\ \emptyset;\ \textbf{void}\ \texttt{bUpdate}(\textbf{nat}\ \texttt{x})\{\texttt{bal==bal-x}\}\ \langle (bal == b_0, bal == b_0 - x), \emptyset \rangle\ \}$

(3) $\texttt{TBasicAccount} \in \textit{impl}(D_{\mathcal{C}_1}(\texttt{CFeeAccount})) \wedge \texttt{deposit} \in \mathcal{T}_1(\texttt{TBasicAccount})$

(4) $\langle (bal == b_0, bal == b_0 + x), \emptyset \rangle \in \textit{specs}(\mathcal{T}_1(\texttt{TBasicAccount})(\texttt{deposit}))$
$\mathcal{C}_2 = \langle D_{\mathcal{C}_1}, S_{\mathcal{C}_1}[(\texttt{CFeeAccount}, \texttt{deposit}) \mapsto \langle (bal == b_0, bal == b_0 + x), \emptyset \rangle] \rangle$

(5) $\texttt{TBasicAccount} \in \textit{impl}(D_{\mathcal{C}_2}(\texttt{CFeeAccount})) \wedge \texttt{withdraw} \in \mathcal{T}_1(\texttt{TBasicAccount})$

(6) $\texttt{W} =\ \textbf{void}\ \texttt{withdraw}(\textbf{nat}\ \texttt{x})\{\texttt{update(x)}\}$

(7) $\mathcal{T}_2 = \mathcal{T}_1[\texttt{TBasicAccount} \mapsto$
$\textit{addSpec}(\mathcal{T}_1(\texttt{TBasicAccount}), \texttt{withdraw}, \langle (bal == b_0, bal == b_0 - x - fee), \texttt{update} : (bal == b_0, bal == b_0 - x - fee) \rangle)]$

(8) $\langle (bal == b_0, bal == b_0 - x - fee), \texttt{update} : (bal == b_0, bal == b_0 - x - fee) \rangle \in \textit{specs}(\mathcal{T}_2(\texttt{TBasicAccount})(\texttt{withdraw}))$
$\mathcal{C}_3 = \langle D_{\mathcal{C}_2}, S_{\mathcal{C}_2}[(\texttt{CFeeAccount}, \texttt{withdraw}) \mapsto \langle (bal == b_0, bal == b_0 - x - fee), \texttt{update} : (bal == b_0, bal == b_0 - x - fee) \rangle] \rangle$

(9) $\texttt{TFeeUpd} \in \textit{impl}(D_{\mathcal{C}_3}(\texttt{CFeeAccount})) \wedge \texttt{update} \in \mathcal{T}_2(\texttt{TFeeUpd})$

(10) $\texttt{U} =\ \textbf{void}\ \texttt{update}(\textbf{nat}\ \texttt{x})\{\texttt{bUpdate(x+fee)}\}$

(11) $\mathcal{T}_3 = \mathcal{T}_2[\texttt{TFeeUpd} \mapsto \textit{addSpec}(\mathcal{T}_2(\texttt{TFeeUpd}), \texttt{update}, \langle (bal == b_0, bal == b_0 - x - fee), \texttt{bUpdate} : (bal == b_0, bal == b_0 - x) \rangle]$
and the proof outline follows directly by substituting formal parameters in the requirement according to rule HoareCall

(12) $\langle (bal == b_0, bal == b_0 - x - fee), \texttt{bUupdate} : (bal == b_0, bal == b_0 - x) \rangle \in \textit{specs}(\mathcal{T}_3(\texttt{TFeeUpd})(\texttt{update}))$
$\mathcal{C}_4 = \langle D_{\mathcal{C}_3}, S_{\mathcal{C}_3}[(\texttt{CFeeAccount}, \texttt{update}) \mapsto \langle (bal == b_0, bal == b_0 - x - fee), \texttt{bUpdate} : (bal == b_0, bal == b_0 - x) \rangle] \rangle$

(13) $\texttt{TBasicUpd[update } \textbf{rT} \texttt{ bUpdate]} \in \textit{impl}(D_{\mathcal{C}_4}(\texttt{CFeeAccount})) \wedge \texttt{bUpdate} \in \mathcal{T}_3(\texttt{TBasicUpd[update } \textbf{rT} \texttt{ bUpdate]})$

(14) $\langle (bal == b_0, bal == b_0 - x), \emptyset \rangle \in \textit{specs}(\mathcal{T}_3(\texttt{TBasicUpd[update } \textbf{rT} \texttt{ bUpdate]})(\texttt{update}))$
$\mathcal{C}_5 = \langle D_{\mathcal{C}_4}, S_{\mathcal{C}_4}[(\texttt{CFeeAccount}, \texttt{bUpdate}) \mapsto \langle (bal == b_0, bal == b_0 - x), \emptyset \rangle] \rangle$

Fig. 14. Analysis details for the class `CFeeAccount`. To simplify the presentation, side conditions and auxiliary computations for the different rule applications are given as notes, `CTE` abbreviates the expression `TFeeUpd + TBasicUpd[update` **`rT`** `bUpdate]`, **`rT`** abbreviates **`renameTo`**, and $\langle \texttt{CFeeAccount} : \ldots \rangle$ denotes that the class contains the same operations as found on the line below.