User-Changeable Visibility: Resolving Unanticipated Name Clashes in Traits

Stéphane Ducasse
Language and Software Evolution – LISTIC Université de Savoie & INRIA Futurs
stephane.ducasse.free.fr

Roel Wuyts
IMEC (Leuven, Belgium) and Université Libre de Bruxelles
homepages.ulb.ac.be/~rowuyts

Alexandre Bergel
Hasso-Plattner-Institut, Germany & LERO, Trinity College Dublin, Ireland
www.bergel.eu

Oscar Nierstrasz
Software Composition Group, University of Bern
www.iam.unibe.ch/~oscar

Abstract

A trait is a unit of behaviour that can be composed with other traits and used by classes. Traits offer an alternative to multiple inheritance. Conflict resolution of traits, while flexible, does not completely handle accidental method name conflicts: if a trait with method *m* is composed with another trait defining a different method *m* then resolving the conflict may prove delicate or infeasible in cases where both versions of *m* are still needed. In this paper we present *freezeable traits*, which provide an expressive composition mechanism to support unanticipated method composition conflicts. Our solution introduces private trait methods *and* lets the class composer change method visibility at composition time (from public to private and vice versa). Moreover two class composers may use different composition policies for the same trait, something which is not possible in mainstream languages. This approach respects the two main design principles of traits: the class composer is empowered and traits can be flattened away. We present an implementation of freezeable traits in Smalltalk. As a side-effect of this implementation we introduced private (early-bound and invisible) methods to Smalltalk by distinguishing object-sends from self-sends. Our implementation uses compile-time bytecode manipulation and, as such, introduces no run-time penalties.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Languages Constructs and Features – *Classes and objects; Inheritance*

General Terms Languages

Keywords Dynamic typing, traits, encapsulation, information hiding, composition.

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.

1. Introduction

Traits are pure units of reuse consisting only of methods [SDNB03, DNS+06]. Traits can be composed to form either other traits or classes. They are recognised for their potential in supporting better composition and reuse, hence their integration in the latest versions of languages such as Perl 6, Squeak [IKM+97], Scala [sca], Slate [Sla] and Fortress [for]. Although traits were originally designed for dynamically-typed languages, there has also been considerable interest in applying traits to statically-typed languages [FR03, SD05, NDS06].

One of the key design principles behind traits is that they empower the *composer* of traits. The composer has the full control of the composition and the conflict resolution [DNS+06]. The design principle behind traits was to favor simplicity over completeness [Bra92] or expressiveness [Mey97]. In our opinion it is important that language features stay simple to avoid overwhelming existing developers, and to ease their introduction in existing languages when possible. This simplicity-by-design works well in most cases, but has two drawbacks. First of all there is no notion of visibility in traits, which may increase the number of con-

flicting methods that have to be handled at composition time. This does not pose any conceptual problems, but can become tedious in practice. But worse, a number of unanticipated method clash problems are sometimes poorly handled. This is shown in the contrived but compact example of Figure 1.

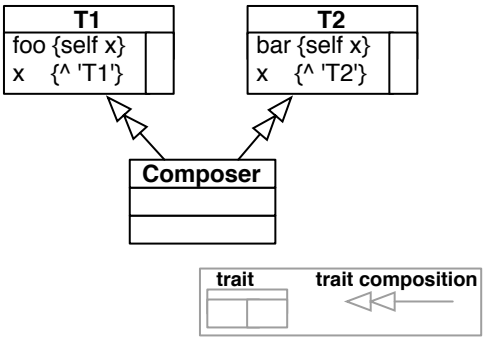

Figure 1. Name clash with traits: in Composer either T1's x, T2's x or a new method x is called by both foo *and* bar.

Figure 1[1] shows a class, Composer, that uses two traits T1 and T2, each of which introduce a method x. T1 and T2 are therefore said to be *in conflict* since both define a method x. One strong requirement is not to modify T1 or T2 as they might be used by other classes or as part of other trait compositions. With the original trait conflict resolution, the conflict may be resolved by either *removing* one method x, from the composition or by defining a method x in the Composer class, essentially overriding both x's. These two simple approaches to handle the conflict turned out to be very practical (as shown by the refactorings of several large hierarchies [BSD03][2]).

However, there are some cases where the conflict resolution of traits is not satisfactory: there is *no way* to let the methods foo *and* bar access the method x contained in their *respective* traits. Either T1's x, T2's x *or* a new method x has to be present in the class Composer. That said, if the x methods are specific helper methods in their respective trait (*i.e.*, the behavior of foo depends on the behavior of T1's x and the behavior of bar depends on the behavior of T2's x), then the composition and conflict resolution will break foo, bar or both. This prevents traits from being used in an unanticipated manner in certain scenarios, and lowers their reusability.

The composition problem we just described with traits is actually more general and exists in other systems as well, together with a number of solutions. Eiffel's key composition mechanism offers the ability to rename methods to resolve method conflicts [Mey97]. Encapsulation policies offer a way to constrain encapsulation interfaces for different users [SBD04]. This approach could be used to solve unanticipated method name clashes in traits (which actually was the original goal, even if not presented as such in the paper). While encapsulation policies can be used to deal with unanticipated method name conflicts they introduce an extra concept in the language: the encapsulation policy. Integrating encapsulation policies in a host language is difficult. This is why we looked for another solution and arrived at *freezable traits*, presented in this paper.

Freezable traits is a composition mechanism that deals with unanticipated method-name conflicts. It is based on the ability to have private methods (early bound and non-visible) within a trait, that can *change* their visibility at *composition time*. Also, two composers of the same trait can have different composition concerns. This supports the key principle of traits: the composer has full control over the composition. The composer may decide to make a method private or public to solve a conflict in a specific composition. Our solution supports the flattening property of traits [DNS+06, NDS06] *i.e.*, that traits can be flattened away and do not change the run-time semantics of method lookup.

A point of vocabulary. A *composer* is a class or a composite trait composed from other traits. A *client* is a class that sends messages to an instance of a class that may have been composed from traits (see Figure 2).

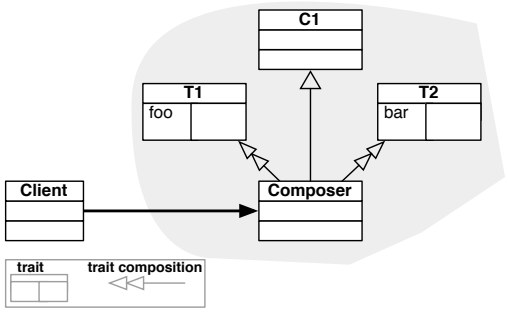

Figure 2. A Composer (a class or a trait) composes behavior from existing traits and superclasses. A Client uses the result of the composition.

In Section 2 we present the original traits design and identify its limitations for resolving method conflicts. In Section 3 we describe freezable traits, a minimal extension of the original traits model in which traits define access rights for methods, and composers may redefine those rights. In Section 4 we describe the operator semantics. In Section 5 we present the implementation of freezable traits in a dynamically-typed language. Our implementation is based on bytecode transformation, is purely static, and does not introduce any run-time costs. In Section 6 we discuss the tradeoffs of the various mechanisms for resolving conflicts. In Section 8 we provide an overview of related work. In Sec-

[1] Double arrowhead represents trait composition. ∧ indicates a return value. (*cf.*, Smalltalk syntax)

[2] Conflicts did not arise for the simple fact that the code being refactored came from a single inheritance hierarchy.

tion 9 we conclude by summarising the presented work and outlining future work.

2. Traits and their limitations

This section presents traits in a nutshell [DNS+06]. A reader already familiar with traits may skip this section and jump directly to Section 2.2, which describes the limitations of conflict resolution. These limitations are resolved through the introduction of freezable traits in Section 3.

2.1 Traits as units of behaviour

Reusable groups of methods. Traits are sets of methods that serve as the behavioural building block of classes and primitive units of code reuse [DNS+06]. In addition to offering behaviour, traits also *require methods*, *i.e.*, methods that are needed so that trait behaviour is fulfilled. Traits do not define state, instead they require accessor methods.

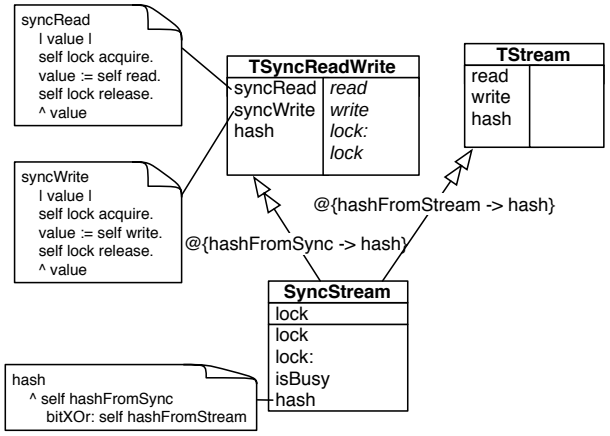

Figure 3. The class SyncStream is composed of the two traits TSyncReadWrite and TStream.

Figure 3 shows a class SyncStream that uses two traits, TSyncReadWrite and TStream. The trait TSyncReadWrite provides the methods syncRead, syncWrite and hash. It requires the methods read and write, and the two accessor methods lock and lock:. We use an extension to UML to represent traits (the right column lists required methods while the left one lists the provided methods).

Explicit composition. A class contains a super-class reference, uses a set of traits, defines state (variables) and behaviour (methods) that glue the traits together; a class implements the required trait methods and resolves any method conflicts.

Trait composition respects the following three rules:

- Methods defined in the composer take precedence over trait methods. This allows the methods defined in a composer to override methods with the same name provided by the used traits; we call these methods *glue methods*.
- Flattening property. In any class composer the traits can be in principle in-lined to give an equivalent class definition that does not use traits.
- Composition order is irrelevant. All the traits have the same precedence, and hence conflicting trait methods must be explicitly disambiguated.

Conflict resolution. While composing traits, method conflicts may arise. A *conflict* arises if we combine two or more traits that provide identically named methods that do not originate from the same trait. There are two strategies to resolve a conflict: by implementing a (glue) method at the level of the class that *overrides* the conflicting methods, or by *excluding* a method from all but one trait. Traits allow method *aliasing*; this makes it possible to introduce an additional name for a method provided by a trait. The new name is used to obtain access to a method that would otherwise be unreachable because it has been overridden [DNS+06].

In Figure 3, the class SyncStream is composed from TSyncReadWrite and TStream. The trait composition associated to SyncStream is:

TSyncReadWrite alias *hashFromSync* → *hash*
+ TStream alias *hashFromStream* → *hash*

The class SyncStream is composed of (i) the trait TSyncReadWrite for which the method hash is aliased to hashFromSync and (ii) the trait TStream for which the method hash is aliased to hashFromStream.

Method composition operators. The semantics of trait composition is based on four operators: sum (+), override (▷), exclusion (−) and aliasing (alias →) [DNS+06].

The *sum* trait TSyncReadWrite + TStream contains all of the non-conflicting methods of TSyncReadWrite and TStream. If there is a method conflict, that is, if TSyncReadWrite and TStream both define a method with the same name, then in TSyncReadWrite + TStream that name is bound to a known method conflict. The + operator is associative and commutative.

The *override* operator (▷) constructs a new composition trait by extending an existing trait composition with some explicit local definitions. For instance, SyncStream overrides the method hash obtained from its trait composition.

A trait can *exclude* methods from an existing trait using the exclusion operator −. Thus, for instance, TStream − {read, write} has a single method hash. Exclusion is used to avoid conflicts, or if one needs to reuse a trait that is "too big" for one's application.

The *method aliasing* operator alias → creates a new trait by providing an additional name for an existing method. For example, if TStream is a trait that defines read, write

and hash, then TStream alias *hashFromStream → hash* is a trait that defines read, write, hash and hashFromStream. The additional method hashFromStream has the same body as the method hash. Aliases are used to make conflicting methods available under another name, perhaps to meet the requirements of some other trait, or to avoid overriding. Note that since the body of the aliased method is not changed in any way, an alias to a recursive method is not recursive.

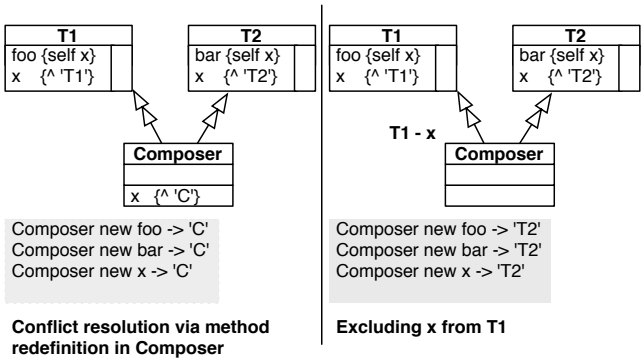

Figure 4. Trait conflict resolution strategies: either via method redefinition or via method exclusion.

In the following section we use the compact example shown in Figure 4.

Trait Design Principles. Three principles guided the design of traits: first, traits should represent a minimal perturbation to a classic class-based language. Second, conflicts are detected automatically and there is no implicit conflict resolution. When conflicts are detected the composer is responsible for resolving them explicitly using the available composition operators. Third, traits can be flattened away (they do not impose a runtime semantics and can therefore be compiled away).

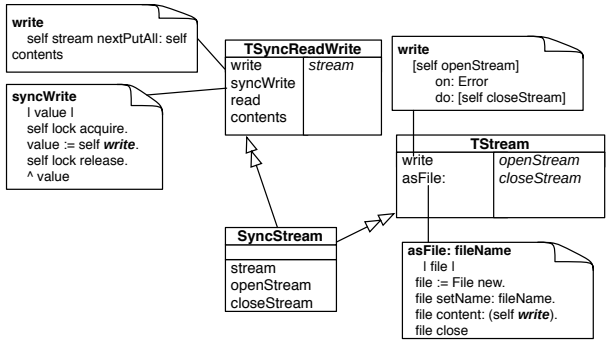

Figure 5. Name clash with traits

2.2 Trait conflict resolution limits

While trait composition lets the class composer resolve conflicts by redefining methods in the composer or excluding them from the composed traits, there are conflicts that are not well handled by traits. Figure 5 illustrates the problem. The class SyncStream is composed of two traits TSyncReadWrite and TStream. Since both traits define the method write, a conflict arises when composing TSyncReadWrite with TStream: the two write methods are different, neither being compatible or composable. They are both used in different non-composable contexts. The default trait conflict resolution is of no help since excluding the method write does not work — each respective trait needs to invoke its specific write method. Redefining the method write in the class does not work either because the behaviours of the two methods are not composable. Even wrapping one of the traits in another trait or class does not solve the problem, as traits can be flattened away at composition time without any means to rename a method. The only solution for the developer may be to rewrite one of the traits which is clearly against the trait goal to support reusable composable abstractions.

3. Freezable traits: adjusting method visibility at composition-time

As was discussed in detail in previous section, composing traits may result in conflicts, and for some compositions the desired solution cannot be expressed without rewriting some of the conflicting methods.

3.1 Adjusting visibility

This paper solves the composition problems of the original traits by introducing slight changes to the original traits model. First we consider now that the host language supports two access modifiers (private and public). Then two new composition operators let a composer change method accessibility at composition time.

With *freezable traits*, a trait developer gives each method a default access modifier of either *private (-)* or *public (+)*, similar to the access modifiers in Java or C++ for example:

- a *private* method is only accessible from within the defining trait. It is therefore bound early to the *defining* trait and *not visible* to the client or the composer. Other methods within the trait referring to the private method will always refer to it, even when they are composed with another class or trait.
- a *public* method is dynamically bound and visible to the client and the composer, behaving as in the original traits model. The method may be overridden in the composer, and sends from methods in the trait to a public method will then result in the execution of the overridden method in the composer.

Unlike most existing languages, the default access modifiers can be changed by the composer at composition time with the help of two new operators: *freeze* (making a public method private) and *defrost* (making a private method public).

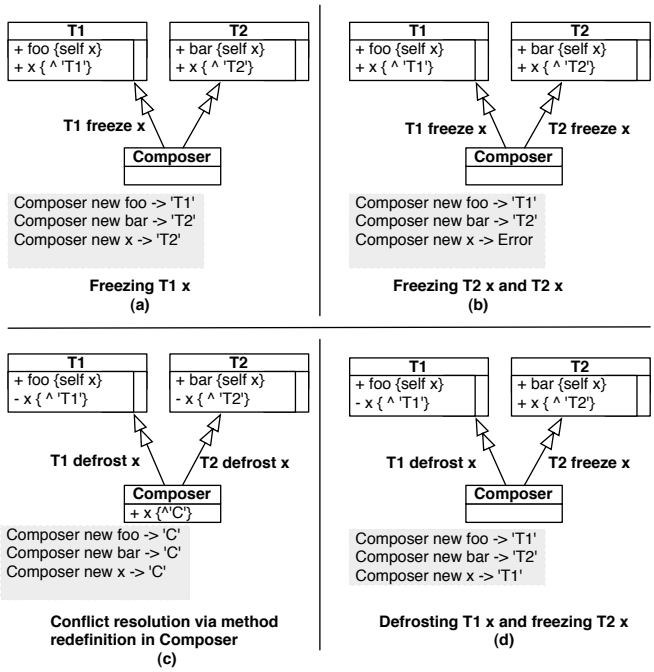

Figure 6. Freeze and defrost semantics illustrated.

Access-modifiers are a well-known mainstream language feature. They are typically seen as simple and well-understood mechanisms, but their exact semantics differ between languages. For example Section 8 shows the semantics of *private* in Ruby, where private methods can be overridden in subclasses, while for example, in Java private methods are not inherited and may not be overridden [GJSB05] (p. 144, section 6.6.8). With freezable traits the semantics of public and private combine notions of visibilty and early/late binding, as explained in detail in the following sections. Their implementation is simple as shown in Section 5.

Note that the client only has access to the public methods resulting from the composition process. Freezable traits follow the original traits philosophy of having few composition operators and putting the composer in charge of the conflict resolution.

Figure 6 shows four often-occurring scenarios when applying traits in the freezable traits model.

- In (a), the two public methods x conflict. To avoid this situation, the composer decides to freeze T1 x. Therefore the x invocation in T1 foo is early bound to T1 x. As a consequence, the method foo *always* invokes T1 x, regardless of whether the Composer class is subclassed or not (*i.e.*, the dynamic type of self differs from Composer).

  In addition T1's x is not visible in the composer, therefore invoking x returns the value 'T2' according to the definition of T2. Since T2's x is dynamically bound, bar invokes it.

- In (b) the two public methods x conflict, and the composer decides to freeze them both. Therefore the invocation of x in T1 foo is bound to T1 x and the one in T2 bar to T2 x. With none of the methods x being visible for the client, it cannot send a message x to an instance of Composer.

- In (c) the two methods x are private. The composer decides to defrost the two methods x. This generates a composition conflict that is then handled by defining a new method x in the class Composer. Invoking x in T1 foo and in T2 bar results in the execution of this new method x, since x is dynamically bound.

- In (d) the method T1 x is private while the method T2 x is public. The composer decides to freeze T2 x and defrost T1 x. Therefore, we end up with the opposite situation from (a).

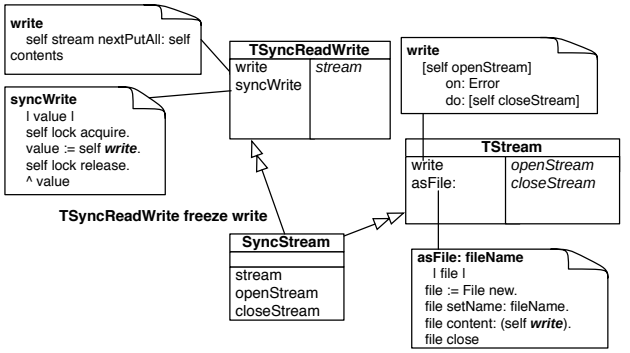

Figure 7. Conflict resolution with the freeze operator.

Using freezable traits, the situation we described in Figure 5 can be easily solved, since the composer can exclude, redefine or freeze the conflicting method. Figure 7 illustrates the solution where one of the conflicting methods is frozen.

The fact that the composer decides the final visibility of methods at composition time implies that a same trait can have different visibilities for its methods in different compositions. For example, Figure 8 illustrates how the trait TSyncReadWrite can be used by two classes, SyncStream and SequenceableStream. The former class uses TSyncReadWrite freeze write, whereas the latter simply uses it. From the point of view of SyncStream, the method write is frozen. TSyncReadWrite only defines one method, syncWrite. From the point of view of SequenceableStream, TSyncReadWrite defines two methods, write and syncWrite, each being publicly accessible from this class.

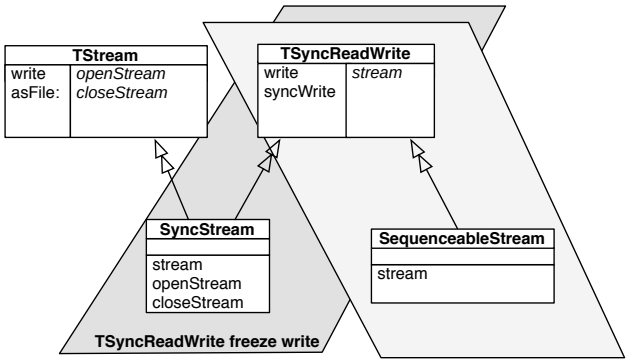

Figure 8. Different visibility policies can be applied to the same trait.

3.2 Freezing in the presence of subtraits

This section extends the notion of freezing to traits that are themselves composed from other traits (subtraits) and thus create trait composition hierarchies. A natural question that arises then is what happens when we start freezing and defrosting when composing such hierarchies. Since the response to this question is not as simple as one might think, this section focusses just on freezing. The next section tackles defrosting.

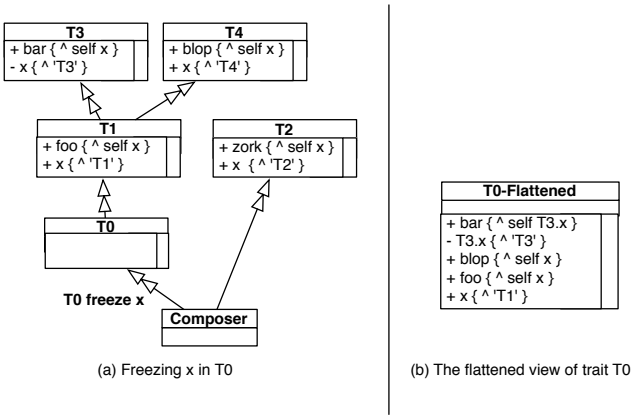

Figure 9. Freezing a trait composed from subtraits.

The previous section explains the basic mechanism of freezing, *i.e.*, making a public method private and therefore making all local calls to this method static. When a trait is itself composed from other traits (its subtraits), and these traits again have subtraits, we have to think about exactly what methods need to be frozen. We present the possible alternatives for the freezing semantics and then discuss our choice.

Figure 9(a) pictures a trait composition where we want to freeze a method x. The result of freezing x could be:

1. All public methods x are frozen, all the way up the trait composition chain. In the given example this would mean that the methods x provided from T4 as well as T1 are made private, and that therefore blop would call x in T4 and foo would call x in T1. There would be no changes for T3.
2. The public method in the *flattened* trait that needs to be frozen is made private, and the callers of this newly private method become static calls.

   Note that in the flattened version of a trait, there is at most one method with the given name to be frozen. In the example given, the flattened trait T0, shown in Figure 9(b), only has one public method x, namely the one provided by T1, which overrides the method x from T4. This method is made private, and the dynamic calls to it in methods blop and foo become static.
3. We could simply disallow freezing a method which is provided by a subtrait and not provided by the composed trait itself. In the example this would mean that there is a composition error since x is not provided by T0 but is provided by T1, a subtrait of T0.
4. We could decide to add explicit references to indicate the exact method to be frozen. So instead of saying Composer uses {T0 freeze x} + {T2} we should write Composer uses {T0 freeze T1.x} + {T2} to freeze the public method x in T1.

We decided in favour of the second option, and say that freezing a method x in a trait T means making at most the public method x of the *flattened* trait T private. This option respects the principle of least astonishment. The result of freezing a trait hierarchy and using it in a composition is that you decide to use it as-is, with all calls going to the methods as though the trait were not composed. Option 1 completely changes the composition since suddenly calls that would result in the invocation of one method suddenly are bound to another one after the freezing. In our example this would mean that both the x provided by T1 as the one provided by T4 would be frozen, and therefore blop would call its private method x and return 'T4' when invoked, instead of 'T1' before the freezing.

Note that using the flattened trait T implies that only non-conflicting methods may not be frozen. For example, if in Figure 6(a) there would be no freeze clause T1 freeze x, then another trait using Composer could not freeze x.

The second option also respects encapsulation: saying that method x is frozen in trait T freezes method x in T regardless of where it comes from. Option 3 is unattractive, since the composer should not care whether a used trait is composite or not. Option 4 introduces unwanted fragility, since aspects of the composition hierarchy are hardcoded into the freezing expressions.

With the chosen semantics, Figure 10 shows that calls to x would result in the same method execution in T0 before and after the composition. The left part shows the composition of

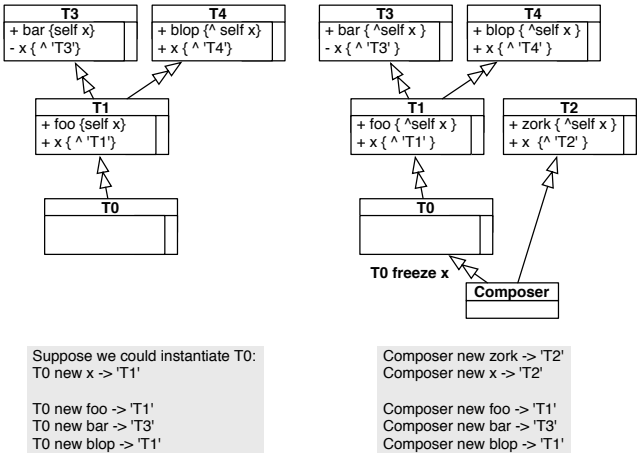

Figure 10. Freezing a trait composed from subtraits results in the flattened trait being used when freezing.

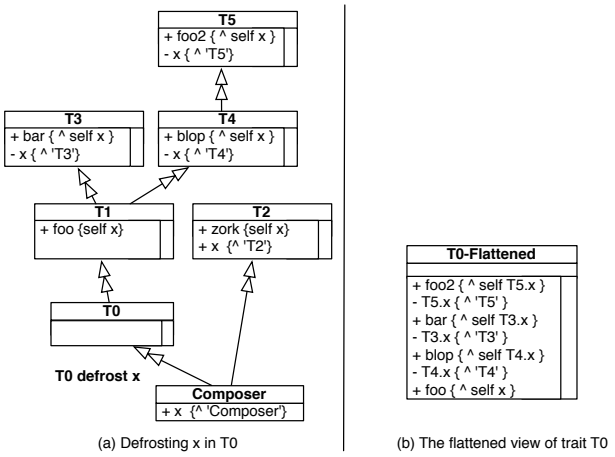

Figure 11. Defrosting in a trait composed from subtraits.

trait T0 by itself. Let's suppose for a moment that we could instantiate a trait and send messages to it, which allows us to investigate the method lookup results. We observe that calls in foo and bar are late bound, and would result in the execution of method x of T1. Method bar is statically bound to the private method x in T3. When trait T0 is used in a composition, as shown in the right part of Figure 10, and x is frozen in T0, the results of sending messages foo, bar and zork are exactly the same as for trait T0 by itself. This is why we dubbed the operation *freeze* in the first place: the existing behaviour before composing is frozen and used as such in the composition.

3.3 Defrosting in the presence of subtraits

The *defrost* operator makes a a previously frozen method public. This section is dedicated to how this operator behaves with a potentially deep hierarchy of composing traits.

Defrost targets private methods, *i.e.*, methods that are not part of the public interface of a trait. One key semantic point arises when several private methods x issued from a trait composition belong to the same composite trait. Defrosting x in such a situation deserves special attention, as discussed in this section.

Figure 11 illustrates a situation where several private x methods are present. Each of T3, T4 and T5 has a private x present also in T0 as illustrated in the flattened representation of T0. Note that these methods are present but not reachable from composers. As in the previous section, we consider several possible semantics for *defrost*:

1. All private methods x are defrosted, all the way up the trait composition chain. In the hierarchy presented in Figure 11, T0 defrost x results in three public x methods, obtained respectively from T3, T4 and T5. These three methods conflict is solved in Composer.
2. All private methods x all the way up the trait composition chain that are not overridden are defrosted. Informally this means that the 'lowest' occurrences in the composition chain of x are made public. If Figure 11 would not contain T3, then T0 defrost x would make T4.x public but not T5.x. In the situation depicted by Figure 11, T0 defrost x will defrost the *two* private methods T3.x and T4.x and reveal two public methods x obtained from T3 and T4, thereby producing a method conflict. T5.x remains private.
3. Only private methods from immediate subtraits can be defrosted by the composer. Since T0 does not freeze any x, it is an error to attempt to defrost x in T0.
4. We could decide to add explicit references to indicate which methods to defrost. In that case, in order to defrost T3's x method we would have to write T0 defrost T3.x.

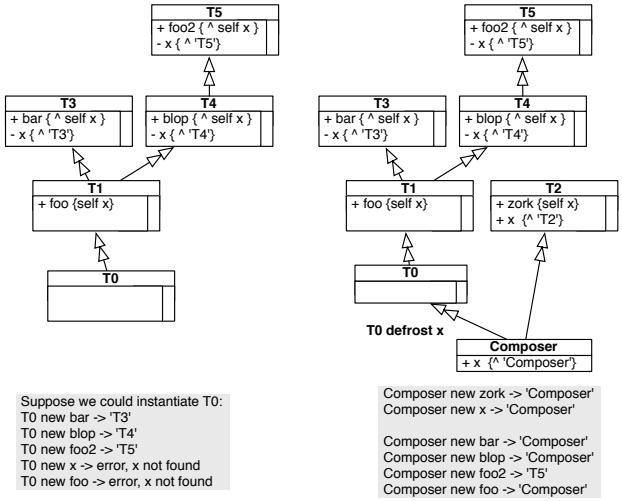

Figure 12. Defrosting in a trait composed from subtraits.

We decided in favour of the second option because it makes *defrost* dual to its counterpart, *freeze*. This means that freezing and then defrosting the same method x in a trait T is equivalent to T. In that case, we have the following relation: T freeze x defrost x = T. It is then possible to deduce T defrost x freeze x = T.

Option 1 does not offer this duality. Whereas T0 defrost x freeze x = T0 since all x in T0 are private, (T0 + T2) defrost x freeze x would not be equal to (T0 + T2) since T2's x would be private. This would also break encapsulation.

Option 3 would not allow us to formulate the expression T freeze x defrost x since T freeze x is a new trait that does not *directly* define x (even privately).

Option 4 uses harcoded prefixes that break encapsulation and would thereby make trait composition fragile with regard to changes in subtrait composition.

It is worth noting that defrosting a method may lead to a conflict as it happens in T0 defrost x. However this situation is easily solved as illustrated in Figure 11. Defrosting x in T0 turns T3 and T4's x public, which results in a conflict. This conflict is solved in Composer by defining a new x. Note that this new version of x is the one invoked by zork.

One characteristic of our design decision is that not every private method is defrosted, only those that are reachable (e.g. that are not overridden). On the figure, the method x issued from T5 may not be defrosted.

4. Operator semantics

To specify the semantics of the new operators we define FREEZABLETRAITS, an extension of SMALLTALKLITE. SMALLTALKLITE is a dynamic language calculus featuring single inheritance, message-passing, field access and updates, and self / super sends. SMALLTALKLITE has already been presented in a previous paper [BDNW07] and is heavily inspired by CLASSICJAVA defined by Flatt *et al.* [FKF98]. We repeated the full description of SMALLTALKLITE in the appendix to aid the reader, but we do not consider it to be a contribution of this paper. We did not use FEATHERWEIGHTJAVA, as in our previous work [NDS06], since FEATHERWEIGHTJAVA is purely functional and does not support super. Moreover, we would like to have a common calculus to express both stateful traits [BDNW07] and freezable traits.

The syntax of FREEZABLETRAITS is presented in Figure 13. Note that we do not provide syntax for declaring methods public or private, since (i) this is not needed to define the semantics of the operators, and (ii) this can be simulated *post hoc* by defining a new trait in which methods to be made private are frozen.

The example described in Figure 5 that presents the stream synchronization problem is written with FREEZABLETRAITS as follows:

```
trait TSyncReadWrite {
    write () { "implementation 1" ... }
    syncWrite () {
        self.lock.acquire()
        let value = self.write() in {
            self.lock.release()
            value }
    }
}

trait TStream {
    write () { "implementation2" ... }
    asFile (fileName) {
        let file = new File () in {
            file.setName (fileName)
            file.content (self.write())
            file.close() }
        }
    }
}
```

4.1 Flattening property

A key feature of traits is that they can be *flattened* [DNS+06]. This means that adding traits to a language does not require a change to the method lookup semantics. As a direct consequence, traits can be compiled away.

We demonstrate the flattening property for traits with the *freeze* and *defrost* operators by defining a flattening function from FREEZABLETRAITS to SMALLTALKLITE.

We distinguish a named trait t from a trait expression τ which may alias, exclude, freeze or defrost methods. A trait t declares a number of methods, but no fields. A trait or a class may use any number of traits, possibly modifying them in a trait expression. A trait expression τ may (i) define an alias m' for an existing method m, (ii) exclude a method m, (iii) freeze a method (*i.e.*, hide a method and statically bind it), (iv) make a (previously frozen) method publicly visible.

Figure 14 presents the flattening function. The flattening is expressed by translating an expression defining a class that references traits to a plain class and method definitions. The translation expands trait expressions to method declarations and is valid if the resulting classes contain no conflicts. The translation is specified in terms of five trait operators (Figure 15). Trait composition ($+$) may generate conflicts if two methods with the same name occur in the composed traits. Class methods take precedence ($\triangleright$) over any used trait methods. Aliasing may generate a conflict if a method has already been defined under the name of the alias. If the method being aliased does not exist, there is no effect. Exclusion simply removes the named trait. Freezing a method removes this method from the interface of a trait, and statically binds sends to it so that may it occur in other methods. Defrosting a method makes it publicly visible and causes sends to it to be dynamically bound.

To be able to fully describe the semantics of the operators we chose to represent how the method visibility changes in the context of our implementation, *i.e.*, a dynamically-typed object-oriented language. Therefore, we represent the special treatment we perform on self-sends (see Section 5). The mechanism to decide whether a message send would

$$
\begin{array}{rcl}
P & = & \mathit{defn}^* e \\
\mathit{defn} & = & \textbf{class}\ c\ \textbf{extends}\ c'\ \{\ f^* \mathit{meth}^* \tau^*\ \} \\
 & | & \textbf{trait}\ t\ \{\ \mathit{meth}^* \tau^*\ \} \\
\tau & = & t \mid \tau\ \textbf{alias}\ m' \to m \mid \tau\ \textbf{minus}\ m \\
 & | & \tau\ \textbf{freeze}\ m \mid \tau\ \textbf{defrost}\ m \\
e & = & \textbf{new}\ c \mid x \mid \textbf{self} \mid \mathsf{nil} \\
 & | & f \mid f{=}e \mid e.m(e^*) \\
 & | & \textbf{super}.m(e^*) \mid \textbf{let}\ x{=}e\ \textbf{in}\ e
\end{array}
\qquad
\begin{array}{rcl}
t & = & \text{a trait name} \\
\mathit{meth} & = & m(x^*)\ \{\ e\ \} \\
c & = & \text{a class name} \mid \mathsf{Object} \\
f & = & \text{a field name} \\
m & = & \text{a method name} \\
x & = & \text{a variable name}
\end{array}
$$

Figure 13. FREEZABLETRAITS syntax.

$$
\begin{array}{rcl}
[\![def1 \cdots defn]\!] & = & [\![def1]\!] \cdots [\![defn]\!] \\
[\![\textbf{trait}\ t\ \{\ \mathit{meth}^* \tau^*\ \}]\!] & = & \emptyset \\
[\![\textbf{class}\ c\ \textbf{extends}\ c'\ \{\ f^* \mathit{meth}^* \tau^*\ \}]\!] & = & \textbf{class}\ c\ \textbf{extends}\ c'\ \{\ f^* \mathit{meth}^* \triangleright [\![\tau^*]\!]\ \} \\
[\![\tau_1 \cdots \tau_k]\!] & = & [\![\tau_1]\!] + \cdots + [\![\tau_k]\!] \\
[\![t]\!] & = & \begin{cases} \mathit{meth}^* & \text{if } \textbf{trait}\ t\ \{\ \mathit{meth}^*\} \in P \\ \mathit{meth}^* \triangleright [\![\tau^+]\!] & \text{if } \textbf{trait}\ t\ \{\ \mathit{meth}^* \tau^+\ \} \in P \end{cases} \\
[\![\tau\ \textbf{alias}\ m' \to m]\!] & = & [\![\tau]\!][m' \to m] \\
[\![\tau\ \textbf{minus}\ m]\!] & = & [\![\tau]\!] - m \\
[\![\tau\ \textbf{freeze}\ m]\!] & = & [\![\tau]\!][m'/m]_{\mathit{self}}\ \ \textbf{alias}\ m' \to m\ \ \textbf{minus}\ m \\
 & & \text{where } m' = \rho_{[\tau\ \textbf{freeze}\ m]}(m) \\
[\![\tau\ \textbf{defrost}\ m]\!] & = & [\![\tau]\!][m/m'_i]_{\mathit{self}}\ \ \textbf{alias}\ m \to m'_i\ \ \textbf{minus}\ m'_i \\
 & & \text{where } M = \rho_{[\tau']}(m) \text{ and } m'_i \in M
\end{array}
$$

Figure 14. Flattening FREEZABLETRAITS to SMALLTALKLITE.

$$
\begin{array}{rcl}
M_1 + M_2 & = & \cdots m_i(x_i^*)\{\top\} \cdots m_j(x_j^*)\{e_j\} \cdots , \\
 & & \forall m_i(\cdots)\{\cdots\} \text{ occurring in both } M_1 \text{ and } M_2 \\
 & & \forall m_j(x_j^*)\{e_j\} \text{ occurring uniquely in one of } M_1 \text{ or } M_2, \\
M_1 \triangleright M_2 & = & \cdots m_i(x_i^*)\{e_i\} \cdots m_j(x_j^*)\{e_j\} \cdots , \\
 & & \forall m_i(x_i^*)\{e_i\} \text{ occurring in } M_1, \\
 & & \forall m_j(x_j^*)\{e_j\} \text{ occurring only in } M_2 \\
M[m' \to m] & = & \begin{cases} M + [m'(x*)\{e\}] & \text{if } m(x*)\{e\} \in M \\ M & \text{otherwise} \end{cases} \\
M - m & = & \begin{cases} \cdots M_{j-1}\ M_{j+1} \cdots & \text{if } M_j = m(\cdots)\{\cdots\} \\ M & \text{otherwise} \end{cases} \\
M[m'/m]_{\mathit{self}} & = & \cdots m_i(x_i^*)\{e_i[m'/m]_{\mathit{self}}\} \cdots , \\
 & & \forall m_i(x_i^*)\{e_i\} \text{ occurring in } M
\end{array}
$$

Figure 15. Trait operations

cause (or not) a frozen method to be invoked, would have to be adapted for a statically-typed language. The operator semantics should be the same in terms of the results in the composer and the client visibility.

The expression **freeze** m renames all the self-sends to m to m', where m' is a new name obtained from the $\rho_{[]}$ hiding function. The scope of this hiding is per trait. Moreover, $\rho_{[]}$ is bijective. Detailed information on $\rho_{[]}$ is given in Section 4.3.

Conversely, **defrost** m renames all the m' self-sends to m, where m' is the "hidden" name for m. It creates a new alias m for m', and excludes m' from the trait. This needs to be done throughout the composition hierarchy, for all m's that are not overridden. Defrosting a method m may reveal several hidden methods m anchored in the composition.

Flattening occurs at composition time (*i.e.*, during the compilation). Before being executed, a program needs to be

$$
\begin{array}{rcl}
\textbf{new}\ c\ [m'/m]_{self} & = & \textbf{new}\ c \\
x\ [m'/m]_{self} & = & x \\
\textbf{self}.m(e_i^*)\ [m'/m]_{self} & = & \textbf{self}.m'(e_i^*[m'/m]_{self}) \\
\textbf{self}.n(e_i^*)\ [m'/m]_{self} & = & \textbf{self}.n(e_i^*[m'/m]_{self}),\ \text{if } n \neq m \\
\mathsf{nil}\ [m'/m]_{self} & = & \mathsf{nil} \\
f{=}e\ [m'/m]_{self} & = & f{=}e[m'/m]_{self} \\
e.m(e_i^*)\ [m'/m]_{self} & = & e[m'/m]_{self}.m(e_i^*[m'/m]_{self}) \\
\textbf{super}.m(e_i^*)\ [m'/m]_{self} & = & \textbf{super}.m(e_i^*[m'/m]_{self}) \\
\textbf{super}.n(e_i^*)\ [m'/m]_{self} & = & \textbf{super}.n(e_i^*[m'/m]_{self}),\ \text{if } n \neq m \\
\textbf{let}\ x = e\ \textbf{in}\ e'\ [m'/m]_{self} & = & \textbf{let}\ x = e[m'/m]_{self}\ \textbf{in}\ e'[m'/m]_{self} \\
\top\ [m'/m]_{self} & = & \top
\end{array}
$$

Figure 16. Self-send substitution

flattened (*i.e.*, traits compiled away). To solve the method conflict, the class SyncStream can be written as follows:

```
class SyncStream extends Object {
    TSyncReadWrite freeze write + TStream freeze write
    ...
}
```

A flattened version of the SyncStream class is:

```
class SyncStream extends Object {
    writeTSyncReadWrite () { "implementation 1" }
    syncWrite () {
        self.lock.acquire()
        let value = self.writeTSyncReadWrite() in {
            self.lock.release()
            value }
    }
    writeTStream () { "implementation 2" }
    asFile (fileName) {
        let file = new File () in {
            file.setName (fileName)
            file.content (self.writeTStream())
            file.close() }
        }
    }
}
```

The two methods write have been renamed to writeTSyncReadWrite and writeTStream, respectively (cf Section 4.3).

4.2 Self sends

When a method has to be (un)frozen, sends that may occur within the same trait have to be renamed. This is achieved with the $[m'/m]_{self}\ m$ operator, presented in Figure 16. It replaces the **self**.$m(\cdots)$ pattern in **self**.$m'(\cdots)$ where m' is a newly generated method name.

Since the scope of a freeze is the trait to which it is applied, super-sends are not renamed.

When the expression TSyncReadWrite freeze write is flattened, the call to write() contained in the syncWrite method is translated into self.writeTSyncReadWrite(). Similarly, for the TStream trait, the call of write() in asFile (fileName) is renamed in **self**.writeTStream().

4.3 The hiding function

Hiding a method could be achieved by either defining a fixpoint [BL91, Section 4.7] or by simply generating a new name for this method and renaming its invocations. Since the defrost operation consists of making this method publicly visible, it requires the original method name, so the hiding function, $\rho_{[]}$, must be bijective.

$\rho_{[]}$ is an arbitrary bijective function, parameterized by trait expressions, that maps method names to be frozen to fresh names. It may make use of any information in the trait expression to achieve the mapping.

Figure 17 describes the hiding function for freezable traits. It recurses over the trait composition. Freezing a method m associates a new name m' to it, and defrosting it simply remove this association. We write a hiding function as a set of bindings, $\{m \to m', n \to n', \ldots\}$. We assume the following operations over such functions:

$$
\begin{array}{l}
\{m \to m'\} + \{n \to n'\} = \{m \to m', n \to n'\} \\
\{m \to m'\} + \{m \to m''\} = \{m \to m', m \to m''\} \\
\{m \to m', n \to n'\} \backslash m = \{n \to n'\} \\
\{m \to m'\} \backslash n = \{m \to m'\} \\
\{m \to m', m \to m'', n \to n'\} \backslash m = \{n \to n'\} \\
\{p \to p', m \to m'\} \triangleright \{m \to m'', n \to n'\} = \\
\qquad \{p \to p', m \to m', n \to n'\}
\end{array}
$$

Application looks up a binding, $\{m \to m'\}(m) = m'$. The result of a lookup may be a set of names if a name is bound more than once, $\{m \to m', m \to m''\}(m) = \{m', m''\}$. Such situation may arise when a name is defrosted.

Freezing a method results in renaming it with a fresh name. Creation of this name occurs in the $\rho_{[\tau\ \textbf{freeze}\ m]}$ clause. We assume that a new name m' is obtained from m parametrised with τ. This implies that the fresh name is recoverable in precisely the same context where it has been originally generated. If we use τ **freeze** m twice in different

$$
\begin{array}{rcl}
\rho_{[\textbf{trait } t \ \{ \ meth^* \ \}]} & = & \emptyset \\
\rho_{[\textbf{trait } t \ \{ \ meth^* \tau_1 \cdots \tau_k \ \}]} & = & \rho_{[\tau_1]} + \cdots + \rho_{[\tau_k]} \\
\rho_{[t]} & = & \rho_{[\textbf{trait } t \ \{ \ meth^* \tau^* \ \}]} \\
 & & \text{where } \textbf{trait } t \ \{ \ meth^* \tau^* \ \} \in P \\
\rho_{[\tau \ \textbf{alias } m' \rightarrow m]} & = & \rho_{[\tau]} \\
\rho_{[\tau \ \textbf{minus } m' \rightarrow m]} & = & \rho_{[\tau]} \\
\rho_{[\tau \ \textbf{freeze } m]} & = & \{m \rightarrow m'\} \triangleright \rho_{[\tau]} \\
 & & \text{where } m' \text{ is a fresh new name} \\
\rho_{[\tau \ \textbf{defrost } m]} & = & \rho_{[\tau]} \backslash m
\end{array}
$$

Figure 17. The $\rho_{[]}$ hiding function.

$$
\begin{array}{rcl}
\rho_{[TSyncReadWrite]} & = & \emptyset \\
\rho_{[TStream]} & = & \emptyset \\
\rho_{[TSyncReadWrite \ \textbf{freeze } write]} & = & \{write \rightarrow writeTSyncReadWrite\} \\
\rho_{[TStream \ \textbf{freeze } write]} & = & \{write \rightarrow writeTStream\} \\
\rho_{[TSyncReadWrite \ \textbf{freeze } write \ + \ TStream \ \textbf{freeze } write]} & = & \{write \rightarrow writeTSyncReadWrite, write \rightarrow writeTStream\} \\
\rho_{[TSyncReadWrite \ \textbf{freeze } write \ \textbf{defrost } write]} & = & \emptyset \\
\rho_{[TStream \ \textbf{freeze } write \ \textbf{defrost } write]} & = & \emptyset
\end{array}
$$

Figure 18. Hiding function for the stream example.

contexts, then m is mapped to two different names. When we defrost m, the correct mapped name is recovered.

By construction, we have:

$$
\begin{array}{rcl}
\rho_{[\tau \ \textbf{freeze } m \ \textbf{defrost } m]} & = & \rho_{[\tau \ \textbf{freeze } m]} \backslash m \\
 & = & (\{m \rightarrow m'\} \triangleright \rho_{[\tau]}) \backslash m \\
 & = & \rho_{[\tau]}
\end{array}
$$

Generating a new method name could consist in appending the name of a trait to a method name. Figure 18 gives some definitions based on the stream example. Something important to keep in mind is the fact that the expression $\rho_{[\ldots]}$ is a function. For instance, we have

$$\rho_{[\textit{TStream } \textbf{freeze } \textit{write}]}(\textit{write}) = \textit{writeTStream}$$

The ρ function returns the list of bindings related to non overridden hidden methods. A hidden method that has been overridden is not captured by ρ, and cannot be defrosted therefore. According to the example given in Figure 12, we have:

$$\rho_{[T0]} = \{x \rightarrow xT3, x \rightarrow xT4\}$$

According to the semantics of ρ, the binding $x \rightarrow xT5$ is not part of $\rho_{[T0]}$: $\rho_{[T5]} = \{x \rightarrow xT5\}$ but $\rho_{[T4]} = \{x \rightarrow xT4\}$.

5. Implementing freezable traits in a dynamically-typed language

We implemented freezable traits in Smalltalk, a dynamically-typed language, since the original traits are fully implemented in Squeak Smalltalk [IKM+97].

As is the case with most dynamic languages, Smalltalk does not provide access modifiers, so our implementation has to add them. Therefore a problem that had to be solved in our implementation was how to introduce statically-bound messages without relying on static types [Wol92, SBD04]. Our solution is based on syntactically distinguishing self-sends from object-sends [TH05,SBD04] (see below). Before presenting our approach we want to stress the difficulties that arise when static types are not available to decide which methods to invoke.

The challenge of introducing method hiding. Following the analysis developed in [SBD04], there exist three possibilities to distinguish which method to execute in absence of static type annotations. We can use:

- the dynamic type (*i.e.*, the class) of the receiver,
- the identity of the receiver, or
- the different kinds of message sends

We chose the third one, and therefore distinguish, self, super and object-sends. Object-sends are always late-bound and self-sends may be either early or late-bound [TH05].

Hiding methods in dynamically-typed languages. Our approach distinguishes between self-sends and object-sends. A self-send is a message send to the pseudo-variable self (this in Java). An object-send is a message that is not sent to self or super.

With this syntactical distinction we have three different message sends:

- Self-sends may have a visibility. They may be early bound and not visible to a client or late bound and visible.
- Object-sends (*i.e.*, invocations that syntactically do not use the keyword self or this) are always late-bound and public.
- Super-sends are static anyway and we do not change their status. The lookup starts in the superclass of the class containing the super expression.

Since our solution is based only on a syntactic distinction, freezable traits could also be applied to other dynamic languages (*e.g.*, Python and Ruby).

Note that Ruby [TH05] also distinguishes calls to private methods syntactically (see Section 8). Syntactically distinguishing private calls establishes the following properties (see [SBD04] for a deeper analysis):

- simple for developers to understand. It is more stable over code changes and modification.
- more stable when confronted with its reflective behaviour.
- easier to implement, since it is based on syntactical difference. Hence it can be implemented efficiently at trait composition time. As such it does not break the flattening property since it does not introduce any run-time penalties.

Implementation. To implement freezable traits we used ByteSurgeon, a bytecode manipulation framework [DDT06]. Using it at composition time, we do not have to recompile methods but just transform method bytecode to change selected self-sends to reflect their visibility status. We then install the resulting method in the class method dictionary of the composer as shown by Figure 19 and 20.

In Figure 19, the class C considers the x methods to be private helpers of their respective traits T1 and T2, therefore it declares them to be frozen. The implementation then:

1. changes all the self-sends to x in the traits to be sent to uniquely identified methods (*e.g.*,T2x in T2 and T1x in T1),
2. copy and install in C the transformed methods containing self-sends that have changed, and
3. add entries to the method dictionary of C using the new names and pointing to the trait methods.

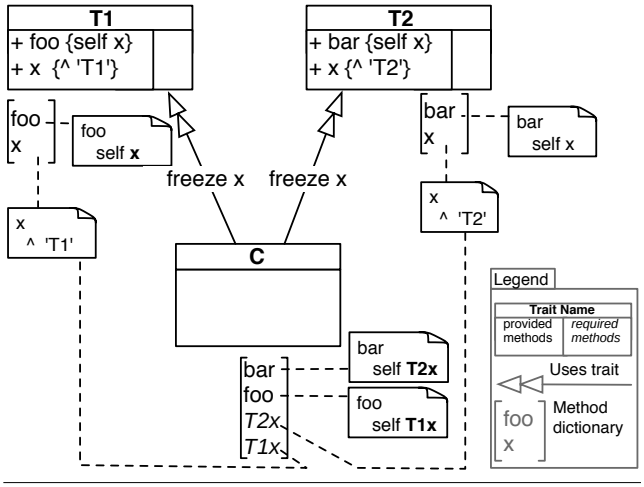

Figure 19. The composer freezes methods at composition time. The early bound (frozen) methods are not visible to the client.

In C the method foo is transformed: its self-send to x now refers to the x method of T1, named T1x which is also installed in the class C and shares the implementation of x. The same happens with bar.

With such a mechanism, several composer classes may use the same traits using different visibility constraints without problem.

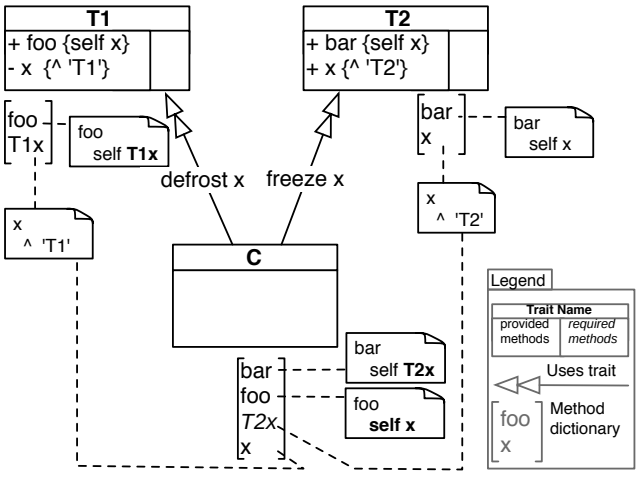

Figure 20. The composer changes visibility at composition time. The early bound (frozen) methods are not visible for the client.

In Figure 20 the class C makes the frozen method x of T1 public while at the same time making the method x of T2 private. Here the method x is frozen, and is not present in the method dictionary of T1 under this name, but is available as a hidden method T1x to the method foo which is calling x via a self-send.

Imagine that in T1 we also have the method zork: arg defined as follows:

```
zork: arg
    arg x
```

In this method, arg x is a normal send (since it is not a self-send). In the context of Figure 20, the expression C new zork: C new will return 'T1', since there is one method x visible and the message arg x is late-bound. On the other hand, if x is frozen or has become frozen as in Figure 19 then the same expression will raise an error since there is no visible x method in C.

6. Renaming vs. hiding

There are three conflict resolution mechanisms in the freezable traits model:

- methods in the composer override methods provided from traits,
- methods can be excluded, and
- methods can be frozen.

Each of these mechanisms solves the conflict by hiding one or more methods. It is as though the methods were not there in the first place, and they are therefore not propagated in the composition. The composing entity has no knowledge of frozen or excluded methods and, even more importantly, neither do its clients or other composing entities.

Conflict resolution mechanisms based on hiding can be contrasted to those based on renaming, even though at first glance they look similar. When one or more conflicting methods are renamed, conflicts are resolved as well.

The key difference between renaming and hiding lies in the fact that renamed methods are part of the composing entity, and clients of other composing entities have access to them, whereas hidden methods are no longer visible. In a system that makes heavy use of trait composition, the presence of conflicting names can impede the effective reuse of traits. Renamed methods will be carried along in the composition, whether they are wanted or not. Exclusion may not be an option if these methods are still needed. With hiding this cannot happen. Methods are never along for the ride unless the composer explicitly wants them to be.

7. Method conflicts revisited

The stateless traits model states that *a conflict arises if two or more traits are combined that provide identically named methods that do not originate from the same trait* [SDNB03, DNS+06]. For freezable traits we note that this formulation is not correct.

To illustrate the problem, Figure 21 shows a trait composition. Composer is composed from two subtraits, T0 and T1 that both are composing trait T. Note that x is being frozen in T0 and T1, and unfrozen by Composer.

With the definition of conflicting methods given above we are in trouble. There are no conflicts between the two methods x, since they are both frozen. There is also no problem for Composer from the two provided methods foo (one through T0 and one through T1) since they originate from the same trait T and are therefore not in conflict. But with freezable traits this poses problems, since the same method foo has a static binding to x (due to the freezing of x when composing T0) but should at the same time have a dynamic binding to x (due to the unfreezing of x when composing Composer). Clearly this should not be allowed.

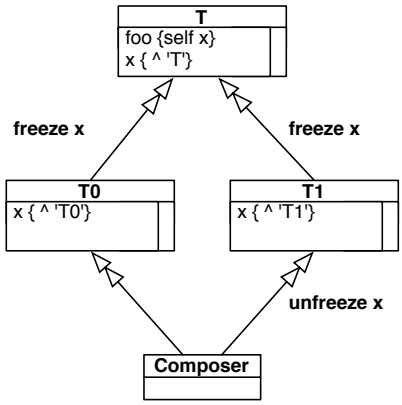

Figure 21. Conflicts between methods obtained via different paths.

In the discussion section of the seminal traits paper [SDNB03] (page 15) a differently formulated definition if conflict is given: *there should be no conflict if the same method is obtained more than once via different paths*. This formulation is correct for the freezable traits model. In the example given it would mean that method foo obtained through T0 is different from the one obtained through T1, because the first has a static binding to x while the second one is dynamically bound. This conflict can be solved by the Composer in different ways, for example by excluding foo from either T0 or T1, and thereby choosing between the static or dynamically bound x.

8. Related work

Qualifier in Java. The Java programming language [GJSB00] provides mechanisms for *access control*, to prevent the users of a package or class from depending on unnecessary details of the implementation of that package or class. Each member (class, interface, field, or method) may have one of those four qualified access:

- if the member or construction is declared *public*, then access is permitted.
- if declared *protected*, then access is permitted only if the access to the member or constructor occurs from (i) within the package containing the class in which the *protected* member or constructor is declared, or from (ii) within the body of a subclass.

- if declared *private*, then access is permitted only within the body of the class that encloses the declaration of the member.
- otherwise, it uses a default access, which is permitted only when the access occurs from within the package in which the type is declared.

At first sight, the *public* and *private* qualifiers seem similar to *defrost* and *freeze* in freezable traits. However *freeze* and *defrost* decouple the definition of methods from the specification of their visibility. The Java *private* method qualifier early binds a method call: the corresponding private method of the message is directly obtained from the class in which the message is sent and is statically determined. The *public* method qualifier makes a method accessible to every object.

The Java type checker is *restrictive*, and does not allow for a method to augment its visibility. For instance, a private or protected method m() in a class A cannot be made public when overridden in a subclass B. The main reason for not allowing a private member to become public is security. In this regard, freezable traits adopt a *permissive* approach, where frozen methods can be made public by being defrosted.

C++. As seen and illustrated in Figure 8, each composer may have a particular view on the traits it uses. For a given method, its visibility is not stated within the trait in which it is defined, but on the composer side. When a composer (*i.e.*, a class or a trait) defines a trait composition, it also specifies the visibility for each method carried by the trait composition.

Letting the composer define its own visibility for the composition methods is one of the major differences between freezable traits and most mainstream composition and visibility mechanisms. More specifically, it is typically not possible that subclasses can inherit a private method, by changing the access modifier to public. Consider the following example program in C++ (the same, minus a number of syntactical differences, holds for Java):

```
class C1 {
private:
     virtual void foo() { };
};

class C2: C1 {
protected:
     virtual void foo() {
          // C1::foo(); // does not compile when uncommented
     };
};
```

While it looks as though the access modifier has been widened, and the private method *foo* becomes protected in the subclass, this is actually not the case: method foo in class C2 is a new method, since method foo from class C1 is not visible. This can be seen when uncommenting one line in the body of C2::foo: the program does not compile because C1::foo is not visible. Note that the inverse (reducing access) may (*e.g.*, C++) or may not (*e.g.*, Java) be possible, depending on the language.

Freezable traits allow the composer (but not the client) to widen or restrict the default access modifiers of existing methods.

C#. C# addresses the problem of unintended name capture in subclasses by early-binding a given method name to a static scope. C# allows the programmer to assign the keyword new (rather than override) to a method to declare that it is used for a different concept than in the superclass and that all calls in the superclass should therefore be statically bound to the local method. The only way to protect internal methods from such unintended name clashes is to explicitly assign the keyword new to the implementation of each of these methods.

Ruby. Ruby is one of the few dynamic languages that offers access qualifiers: methods can be qualified as public, protected and private. Ruby syntactically distinguishes private from public methods, however a private method may be made public in subclasses.

Private methods can only be invoked by sending a message to an *implicit* receiver (*i.e.*, no use of self). However a call to a private method is not statically bound to the method defined in the class but can be overriden in subclasses. The following code illustrates the point: class C defines a private method x. The method foo does not invoke this method since it does not use an implicit receiver but sends the message x to self. This is why C.new.foo raises an error. The method foo2 invokes x with an implicit receiver (*i.e.*, no self). C.new.foo2 executes the private method x and returns 1.

So far this is analogous to the early-binding semantics of freezable traits. Now this is different if a subclass defines a public method x returning 2. D.new.foo2 is interesting since it returns 2 and not 1 even though the method x is private and the method foo2 calls x with an implicit receiver. This shows that Ruby's private methods are dynamically resolved contrary to freezable traits which are statically bound.

```
class C
 def zork(arg) ;  return arg.x ;   end
 def foo ; self.x  end
 def foo2 ;  x ;   end
 private
 def x ; return 1 ;  end
end

class D < C
 public
 def x ;  return 2 ;   end
end

Results:
C.new.foo ==> failed
C.new.foo2 ==> 1
D.new.foo ==> 2
```

```
D.new.foo2 ==> 2
D.new.zork(D.new) ==> 2
D.new.zork(C.new) ==> failed
```

Eiffel. Eiffel [Mey92] is a pure object-oriented language that supports multiple inheritance. Features, *i.e.*, methods or instance variables, may be multiply inherited along different paths. Eiffel provides the programmer mechanisms that offer a fine degree of control over whether such features are shared or replicated. In particular, features may be *renamed* by the inheriting class. It is also possible to *select* a particular feature in the case of naming conflicts. Selecting a feature means that from the context of the composing subclass, the selected feature takes precedence over the possibly conflicting ones.

Eiffel supports a feature adaptation clause, *export*, which allows for an inherited feature to change its export status. A set of features may be exported to a list of classes as illustrated in the following piece of code:

```
class D inherit
        A
            export {X, Y} feature1, feature2 end
        ...
```

The two features, feature1 and feature2, may be invoked by instances of classes X and Y.

Since we defined the semantics of method hiding by renaming self message sends, one might wonder how freezable traits relate to the *rename* operator of Eiffel. First, a class hierarchy *cannot* be flattened in Eiffel as the multiple inheritance and the static type check prevent methods from being linearly ordered. Second, the renaming does not actually hide a method, but renames it and all its references.

Jigsaw. In his PhD thesis, Bracha [Bra92] defined Jigsaw as a minimal programming language in which packages and classes are unified under the notion of a module. A module in Jigsaw is a self-referential scope that binds names to values (*i.e.*, constants and functions). It acts as a class (*i.e.*, an object generator) and as a coarse-grained structural software unit. Modules can be nested, therefore a module can define a set of classes. A number of operators is provided to compose modules. These operators are instantiation, merge, override, rename, restrict, and freeze.

The freeze operator of Jigsaw has the same intent as that of freezable traits, which is to add some privacy to a module or a trait. Despite their similar semantics, two major differences exist. First, in Jigsaw the semantics of freeze is described by means of the Y fix-point operator. This prevents a composition of modules to be flattened. Second, a frozen method cannot be defrosted. In Jigsaw, the interface of a module cannot be widened, whereas with freezable traits, trait interfaces may be narrowed and widened by means of freeze and defrost.

Mixin composition. Van Limberghen and Mens formally describe a mixin model where encapsulation is orthogonal to mixin composition [MvL96]. Their model unifies methods and variables. They propose an operator to encapsulate methods and state locally to the mixin that defines it. When encapsulated, self-sends are early bound. All of this is fairly similar to our model. What is lacking is an inverse operator such as defrost. They do not mention an implementation that would explain how state is represented.

Visibility of modules and packages. In his comparison of module systems, Calliss [Cal91] argues that the name clashing problem ("two same named entities exist in the same region") can be solved either by employing aliasing (which the author refers to "renaming" in his work), or qualified references. With freezable traits, we show that changing visibility is a third option.

One major difference between visibility mechanisms in Java, C#, Ada and freezable traits, is the fact that a visibility policy of a given modular unit is fixed and may not vary in any unanticipated way. With freezable traits, a composer decides which visibility policy to adopt when using a trait.

Object-Oriented Encapsulation. Schaerli et al. proposed encapsulation policies as a way to constrain the interface of an object [SBD04]. With Object-Oriented Encapsulation (OOE), two cases are distinguished: (1) an inheritance perspective where a class can change the way the superclass methods are bound from the subclass perspective and (2) an object perspective where the interface of an object itself may be changed by associating encapsulation policies with object references.

From the inheritance perspective, an encapsulation policy associated with a subclass may be defined to change how methods in the superclass are bound. If a subclass inherits from its superclass using an encapsulation policy that forbids overriding, the subclass may nevertheless re-implement the method with the same name as that in the superclass. In that case, a new method is defined that has the same name as a method in its superclass. OOE is based on the syntactic distinction of three different messages: super-sends, self-send and object-send. Only self-sends can be early bound. This means that for both object-sends and super-sends, the method lookup is the same as in Smalltalk-80 and entirely independent of encapsulation policies. Freezable traits exploit the same syntactic difference between messages and share the same design principle in the sense that the composer controls composition. Contrary to Object-Oriented Encapsulation, we did not introduce new byte-codes and did not change the virtual machine but use compiled method copying and bytecode rewriting. Introducing a new bytecode was not necessary since the goal of OOE was to control object-references and attach to them interfaces (encapsulation policies), therefore it is necessary to distinguish between object-sends and self-sends. The goal of OOE is broader

than that of freezable traits: encapsulation policies could be used to control the visibility of conflicting trait methods but encapsulation policies are more general in terms of providing visibility control in the context of inheritance and for object references. Neither of these issues is a concern for freezable traits.

Scala. Scala [sca, Ode07] is a statically typed programming language integrating object-oriented and functional concepts. It supports classes, traits and mixins. In Scala, a trait is a class that is meant to be added to some other class as a mixin. Classes inherit from other classes and can be composed using mixin composition. This is similar to our approach, with only minor differences (traits in Scala can contain variables and are therefore more similar to stateful traits [BDNW07] than to the stateless traits of this paper).

The major difference regarding conflict resolution between Scala and traits is in the mechanism used. In Scala member resolution is handled by class linearization, and composition conflicts never occur (the automatic linearization takes care of them). In freezable traits method conflicts are automatically triggered at composition time and need to be resolved manually by the composer. A detailed discussion between linearization approaches from different languages and our approach can be found in [DNS+06].

Scala has a number of modifiers that can be used with methods. Methods can be declared private or protected, either on an object basis (object-private or object-protected), or on the more common class basis. There is also the option of declaring methods qualified private or qualified protected, which means that they are accessible only from code within the denoted class or package.

On the other hand Scala does not permit subclasses to change the modifiers of inherited methods, as is possible with freezable traits.

Fortress. The Fortress Programming Language is a general-purpose, statically typed, component-based programming language for producing high-performance software [for]. Functions and methods (collectively called known as functionals) may be overloaded. Within a trait, multiple declarations for the same functional name may coexist in the same scope. In addition several of these may be applied to any particular functional call. Calls to overloaded functionals are resolved by determining the most-specific applicable declaration. Fortress allows functional declarations to be overloaded while ensuring the uniqueness of call declarations for each send. Fortress does not offer visibility qualifiers for methods which are public.

Trait-based metaprogramming. Reppy and Turon have proposed a new trait system [RT06, RT07] called trait-based metaprogramming. Based on the Fisher-Reppy trait calculus [FR03], it adds *deep* operations such as method hiding and renaming to extend the range of conflicts that are solvable by traits.

Hide and *rename* are the deep variants of exclude and alias. The *hide* operation permanently binds a provided method to a trait, while hiding the method's name. A new method with the same name can be introduced as a new provided or required method of the trait, but existing references to the method from other provided methods are statically bound to its implementation at the time of hiding. The *rename* operation changes the name of a method and all its reference in calling methods.

The effect of *hide* is identical to that of *freeze*: both remove a method from a trait's interface and statically binds a method with its invocations on a caller side. However, contrary to our proposal, no counterpart of *hide* is offered.

The use of a type system constitutes the main difference between *trait-based metaprogramming* and *freezable traits*. The main issue that has to be adresssed when *hide* is introduced is to keep the type system sound: the requirements of a trait should remain identical before and after method hiding. To accomplish this, Reppy and Turon "transitively record requirements in the inlining assumptions of any other provided method." The challenge tackled by *freeze* is to be reversible, with the *defrost* operation.

The semantics of trait-based metaprogramming is based on dictionaries (a kind of method dispatch tables) to promote type soundness [RS02]. However, with this design decision the flattening property is sacrificed.

9. Conclusion and future work

In this paper we have presented two new composition operators to define a visibility mechanism for composable behavioural units. Those operators are applied to support unanticipated method conflicts that cannot be resolved with more traditional composition operators such as aliasing and overriding.

After having identified limitations when composing traits, we formulated *freezable traits* which are traits augmented with two operators, *freeze* and *defrost*. A public method may be made private with *freeze*. The dual operation, *defrost* widens the visibility of a method by making a private method public. We demonstrated that freezable traits can be flattened away, preventing any kind of "diamond" situations which multiple inheritance has to deal with. Our proposal includes an implementation suitable for dynamic programming languages such as Ruby, Python and Smalltalk.

In this work, traits do not define variables. A trait is a group of methods that define a behaviour. In our previous work [BDNW07] we augmented a trait with a state declaration. As future work, we plan to combine variable definition with the visibility model described in this paper.

We also plan to use freezable traits in a real-world application, and compare the results with the *stateless* and *stateful* trait models. A possible candidate would be the Collection

hierarchy, since it was previously refactored using stateless traits [BSD03].

Composition of modular units that may conflict in an unanticipated way has been the focus of years of research ranging from classes, modules, and packages just to name a few. Currently, the aspect-oriented community is facing the difficult problem of composing cross-cutting concerns. This paper suggests the use of a visibility mechanism to deal with those situations.

Acknowledgment. We would like to thank the anonymous reviewers for their helpful comments that allowed us to substantially improve the quality of this paper. We also would like to thank Shane Brennan, Damien Cassou and Damien Pollet for their reviews. S. Ducasse gratefully acknowledges the financial support of the french ANR (National Research Agency) for the project "COOK: Réarchitecturisation des applications industrielles objets" (JC05 42872). Oscar Nierstrasz gratefully acknowledges the financial support of the Swiss National Science Foundation for the project "Analyzing, capturing and taming software change" (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008).

References

[BDNW07] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Stateful traits and their formalization. *Journal of Computer Languages, Systems and Structures*, 2007. To appear.

[BL91] Gilad Bracha and Gary Lindstrom. Modularity meets inheritance. Uucs-91-017, University of Utah, Dept. Comp. Sci., October 1991.

[Bra92] Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, Dept. of Computer Science, University of Utah, March 1992.

[BSD03] Andrew P. Black, Nathanael Schärli, and Stéphane Ducasse. Applying traits to the Smalltalk collection hierarchy. In *Proceedings of 17th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'03)*, volume 38, pages 47–64, October 2003.

[Cal91] Frank W. Calliss. A comparison of module constructs in programming languages. *SIGPLAN Not.*, 26(1):38–46, 1991.

[DDT06] Marcus Denker, Stéphane Ducasse, and Éric Tanter. Runtime bytecode transformation for Smalltalk. *Journal of Computer Languages, Systems and Structures*, 32(2-3):125–139, July 2006.

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

[FH92] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, 1992.

[FKF98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183. ACM Press, 1998.

[for] The Fortress language specification. http://research.sun.com/projects/plrg/fortress0866.pdf.

[FR03] Kathleen Fisher and John Reppy. Statically typed traits. Technical Report TR-2003-13, University of Chicago, Department of Computer Science, December 2003.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition*. Addison Wesley, 2000.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification (Third Edition)*. Addison Wesley, 2005.

[IKM+97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of Squeak, A practical Smalltalk written in itself. In *Proceedings OOPSLA '97, ACM SIGPLAN Notices*, pages 318–326. ACM Press, November 1997.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.

[MvL96] Tom Mens and Marc van Limberghen. Encapsulation and composition as orthogonal operators on mixins: A solution to multiple inheritance problems. *Object Oriented Systems*, 3(1):1–30, 1996.

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

[Ode07] Martin Odersky. Scala language secification v. 2.4. Technical report, École Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland, March 2007.

[RS02] Jon G. Riecke and Christopher A. Stone. Privacy via subsumption. *Inf. Comput.*, 172(1):2–28, 2002.

[RT06] John Reppy and Aaron Turon. A foundation for trait-based metaprogramming. In *International Workshop on Foundations and Developments of Object-Oriented Languages*, 2006.

[RT07] John Reppy and Aaron Turon. Metaprogramming with traits. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'2007)*, 2007.

[SBD04] Nathanael Schärli, Andrew P. Black, and Stéphane Ducasse. Object-oriented encapsulation for dynamically typed languages. In *Proceedings of 18th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'04)*, pages 130–149, October 2004.

[sca] The scala programming language. http://lamp.epfl.ch/scala/.

[SD05] Charles Smith and Sophia Drossopoulou. Chai: Typed traits in Java. In *Proceedings ECOOP 2005*, 2005.

[SDNB03] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'03)*, volume 2743 of *LNCS*, pages 248–274. Springer Verlag, July 2003.

[Sla] Slate. http://slate.tunes.org.

[TH05] David Thomas and Andy Hunt. *Programming Ruby*. Addison Wesley, 2nd edition, 2005.

[Wol92] Mario Wolczko. Encapsulation, delegation and inheritance in object-oriented languages. *IEEE Software Engineering Journal*, 7(2):95–102, March 1992.

A. SMALLTALKLITE

We present SMALLTALKLITE, a Smalltalk-like dynamic language featuring single inheritance, message-passing, field access and update, and **self** and **super** sends. SMALLTALKLITE is similar to CLASSICJAVA, but removes interfaces and static types. Fields are private in SMALLTALKLITE, so only local or inherited fields may be accessed.

We base our approach on the object model used by Flatt *et al.* [FKF98] to give a semantics for mixins for Java-like languages. We adapt the CLASSICJAVA model they introduce to develop SMALLTALKLITE, a simple calculus that captures the key features of Smalltalk-like dynamic languages.

A.1 SMALLTALKLITE reduction semantics

The syntax of SMALLTALKLITE is shown in Figure 23. SMALLTALKLITE is similar to CLASSICJAVA, while eliding the features related to static typing. We similarly ignore features that are not relevant to a discussion of traits, such as reflection or class-side methods.

In order to simplify the reduction semantics of SMALLTALKLITE, we adopt an approach similar to that used by Flatt *et al.* [FKF98], namely we annotate field accesses and **super** sends with additional static information that is needed at "run-time". This extended redex syntax is shown in Figure 22. The figure also specifies the evaluation contexts for the extended redex syntax in Felleisen and Hieb's notation [FH92].

Predicates and relations used by the semantic reductions are listed in Figure 25. (The predicates $\text{CLASSESONCE}(P)$ *etc* are assumed to be preconditions for valid programs, and are not otherwise explicitly mentioned in the reduction rules.)

$P \vdash \langle \epsilon, \mathcal{S} \rangle \hookrightarrow \langle \epsilon', \mathcal{S}' \rangle$ means that we reduce an expression (redex) ϵ in the context of a (static) program P and a (dynamic) store of objects $\mathcal{S}$ to a new expression ϵ' and (possibly) updated store $\mathcal{S}'$. A redex ϵ is essentially an expression e in which field names are decorated with their object contexts, *i.e.*, f is translated to $o.f$, and **super** sends are decorated with their object and class contexts. Redexes and their subexpressions reduce to a value, which is either an object identifier or nil. Subexpressions may be evaluated within an expression context E.

$$\begin{array}{rcl} \epsilon & = & v \mid \textbf{new}\ c \mid x \mid \textbf{self} \mid \epsilon.f \mid \epsilon.f{=}\epsilon \\ & \mid & \epsilon.m(\epsilon^*) \mid \textbf{super}\langle o, c\rangle.m(\epsilon^*) \mid \textbf{let}\ x{=}\epsilon\ \textbf{in}\ \epsilon \\ E & = & [\,] \mid o.f{=}E \mid E.m(\epsilon^*) \mid o.m(v^*\ E\ \epsilon^*) \\ & \mid & \textbf{super}\langle o, c\rangle.m(v^*\ E\ \epsilon^*) \mid \textbf{let}\ x{=}E\ \textbf{in}\ \epsilon \\ v, o & = & \mathsf{nil} \mid \mathit{oid} \end{array}$$

Figure 22. Redex syntax

$$\begin{array}{rcl} P & = & \mathit{defn}^*\, e \\ \mathit{defn} & = & \textbf{class}\ c\ \textbf{extends}\ c\ \{\ f^*\mathit{meth}^*\ \} \\ e & = & \textbf{new}\ c \mid x \mid \textbf{self} \mid \mathsf{nil} \\ & \mid & f \mid f{=}e \mid e.m(e^*) \\ & \mid & \textbf{super}.m(e^*) \mid \textbf{let}\ x{=}e\ \textbf{in}\ e \\ \mathit{meth} & = & m(x^*)\ \{\ e\ \} \\ c & = & \text{a class name} \mid \mathsf{Object} \\ f & = & \text{a field name} \\ m & = & \text{a method name} \\ x & = & \text{a variable name} \end{array}$$

Figure 23. SMALLTALKLITE syntax

The store consists of a set of mappings from object identifiers $\mathit{oid} \in \text{dom}(\mathcal{S})$ to tuples $\langle c, \{f \mapsto v\}\rangle$ representing the class c of an object and the set of its field values. The initial value of the store is $\mathcal{S} = \{\}$.

Translation from the main expression to an initial redex is specified out by the $o[\![e]\!]_c$ function (see Figure 24). This binds fields to their enclosing object context and binds **self** to the *oid* of the receiver. The initial object context for a program is nil. (*i.e.*, there are no global fields accessible to the main expression). So if e is the main expression associated to a program P, then $\mathsf{nil}[\![e]\!]_{\mathsf{Object}}$ is the initial redex.

The reductions are summarised in Figure 26.

new c [*new*] reduces to a fresh *oid*, bound in the store to an object whose class is c and whose fields are all nil. A (local) field access [*get*] reduces to the value of the field. Note that it is syntactically impossible to access a field of another object. The redex notation $o.f$ is only generated in

$\prec_P$ Direct subclass

$c \prec_P c' \iff$ **class** c **extends** $c' \cdots \{\cdots\} \in P$

$\leq_P$ Indirect subclass

$c \leq_P c' \equiv$ transitive, reflexive closure of $\prec_P$

$\in_P$ Field defined in class

$f \in_P c \iff$ **class** $\cdots \{\cdots f \cdots\} \in P$

$\in_P$ Method defined in class

$\langle m, x^*, e\rangle \in_P c \iff$ **class** $\cdots \{\cdots m(x^*)\{e\} \cdots\} \in P$

$\in_P^*$ Field defined in c

$f \in_P^* c \iff \exists c', c \leq_P c', f \in_P c'$

$\in_P^*$ Method lookup starting from c

$\langle c, m, x^*, e\rangle \in_P^* c' \iff c' = \mathit{min}\{c'' \mid \langle m, x^*, e\rangle \in_P c'', c \leq_P c''\}$

CLASSESONCE(P) Each class name is declared only once

$\forall c, c',$ **class** $c \cdots$ **class** $c' \cdots$ is in $P \Rightarrow c \neq c'$

FIELDONCEPERCLASS(P) Field names are unique within a class declaration

$\forall f, f',$ **class** $c \cdots \{\cdots f \cdots f' \cdots\}$ is in $P \Rightarrow f \neq f'$

FIELDSUNIQUELYDEFINED(P) Fields cannot be overridden

$f \in_P c, c \leq_P c' \implies f \notin_P c'$

METHODONCEPERCLASS(P) Method names are unique within a class declaration

$\forall m, m',$ **class** $c \cdots \{\cdots m(\cdots)\{\cdots\} \cdots m'(\cdots)\{\cdots\} \cdots\}$ is in $P \Rightarrow m \neq m'$

COMPLETECLASSES(P) Classes that are extended are defined

$\text{range}(\prec_P) \subseteq \text{dom}(\prec_P) \cup \{\mathsf{Object}\}$

WELLFOUNDEDCLASSES(P) Class hierarchy is an order

$\leq_P$ is antisymmetric

CLASSMETHODSOK(P) Method overriding preserves arity

$\forall m, m', \langle m, x_1 \cdots x_j, e\rangle \in_P c, \langle m, x'_1 \cdots x'_k, e'\rangle \in_P c', c \leq_P c' \implies j = k$

Figure 25. Relations and predicates for SMALLTALKLITE

$$P \vdash \langle E[\textbf{new}\ c], \mathcal{S}\rangle \hookrightarrow \langle E[\mathit{oid}], \mathcal{S}[\mathit{oid} \mapsto \langle c, \{f \mapsto \mathsf{nil} \mid \forall f, f \in_P^* c\}\rangle]\rangle \quad [\mathit{new}]$$
where $\mathit{oid} \notin \text{dom}(\mathcal{S})$

$$P \vdash \langle E[o.f], \mathcal{S}\rangle \hookrightarrow \langle E[v], \mathcal{S}\rangle \quad [\mathit{get}]$$
where $\mathcal{S}(o) = \langle c, \mathcal{F}\rangle$ and $\mathcal{F}(f) = v$

$$P \vdash \langle E[o.f{=}v], \mathcal{S}\rangle \hookrightarrow \langle E[v], \mathcal{S}[o \mapsto \langle c, \mathcal{F}[f \mapsto v]\rangle]\rangle \quad [\mathit{set}]$$
where $\mathcal{S}(o) = \langle c, \mathcal{F}\rangle$

$$P \vdash \langle E[o.m(v^*)], \mathcal{S}\rangle \hookrightarrow \langle E[o[\![e[v^*/x^*]]\!]_{c'}], \mathcal{S}\rangle \quad [\mathit{send}]$$
where $\mathcal{S}[o] = \langle c, \mathcal{F}\rangle$ and $\langle c, m, x^*, e\rangle \in_P^* c'$

$$P \vdash \langle E[\textbf{super}\langle o, c\rangle.m(v^*)], \mathcal{S}\rangle \hookrightarrow \langle E[o[\![e[v^*/x^*]]\!]_{c''}], \mathcal{S}\rangle \quad [\mathit{super}]$$
where $c \prec_P c'$ and $\langle c', m, x^*, e\rangle \in_P^* c''$ and $c' \leq_P c''$

$$P \vdash \langle E[\textbf{let}\ x{=}v\ \textbf{in}\ \epsilon], \mathcal{S}\rangle \hookrightarrow \langle E[\epsilon[v/x]], \mathcal{S}\rangle \quad [\mathit{let}]$$

Figure 26. Reductions for SMALLTALKLITE

the context of the object o. Field update [*set*] simply updates the corresponding binding of the field in the store.

When we send a message [*send*], we must look up the corresponding method body e, starting from the class c of the receiver o. The method body is then evaluated in the context of the receiver o, binding **self** to the receiver's *oid*. Formal parameters to the method are substituted by the actual arguments (see Figure 27). We also pass in the actual class in which the method is found, so that **super** sends have the right context to start their method lookup.

super sends [*super*] are similar to regular message sends, except that the method lookup must start in the superclass of class of the method in which the **super** send was declared. When we reduce the **super** send, we must take care to pass on the class c'' of the method in which the **super** method was found, since that method may make further **super** sends. **let in** expressions [*let*] simply represent local variable bindings.

Errors occur if an expression gets "stuck" and does not reduce to an *oid* or to nil. This may occur if a non-existent

$$
\begin{array}{rcl}
o[\![\textbf{new}\ c']\!]_c & = & \textbf{new}\ c' \\
o[\![x]\!]_c & = & x \\
o[\![\textbf{self}]\!]_c & = & o \\
o[\![\mathsf{nil}]\!]_c & = & \mathsf{nil} \\
o[\![f]\!]_c & = & o.f \\
o[\![f{=}e]\!]_c & = & o.f{=}o[\![e]\!]_c \\
o[\![e.m(e_i^*)]\!]_c & = & o[\![e]\!]_c.m(o[\![e_i]\!]_c^*) \\
o[\![\textbf{super}.m(e_i^*)]\!]_c & = & \textbf{super}\langle o, c\rangle.m(o[\![e_i]\!]_c^*) \\
o[\![\textbf{let}\ x{=}e\ \textbf{in}\ e']\!]_c & = & \textbf{let}\ x{=}o[\![e]\!]_c\ \textbf{in}\ o[\![e']\!]_c
\end{array}
$$

Figure 24. Translating expressions to redexes

$$
\begin{array}{rcl}
\textbf{new}\ c\ [v/x] & = & \textbf{new}\ c \\
x\ [v/x] & = & v \\
x'\ [v/x] & = & x' \\
\textbf{self}\ [v/x] & = & \textbf{self} \\
\mathsf{nil}\ [v/x] & = & \mathsf{nil} \\
f\ [v/x] & = & f \\
f{=}e\ [v/x] & = & f{=}e[v/x] \\
e.m(e_i^*)\ [v/x] & = & e[v/x].m(e_i^*[v/x]) \\
\textbf{super}.m(e_i^*)\ [v/x] & = & \textbf{super}.m(e_i^*[v/x]) \\
\textbf{let}\ x{=}e\ \textbf{in}\ e'\ [v/x] & = & \textbf{let}\ x{=}e[v/x]\ \textbf{in}\ e' \\
\textbf{let}\ x'{=}e\ \textbf{in}\ e'\ [v/x] & = & \textbf{let}\ x'{=}e[v/x]\ \textbf{in}\ e'[v/x]
\end{array}
$$

Figure 27. Variable substitution

variable, field or method is referenced (for example, when sending any message to nil). For the purpose of this paper we are not concerned with errors, so we do not introduce any special rules to generate an error value in these cases.