Stateful Traits and their Formalization ⋆

Alexandre Bergel [a] Stéphane Ducasse [b] Oscar Nierstrasz [c]
Roel Wuyts [d]

[a] *LERO & DSG, Trinity College Dublin, Ireland*

[b] *LISTIC – University of Savoie, France*

[c] *SCG, University of Bern, Switzerland*

[d] *IMEC, Belgium, also Professor at Université Libre de Bruxelles, Belgium*

Revised version submitted to Journal of Computer Languages, Systems and Structures,
vol. 34, no. 2-3, 2008, pp. 83-108.

doi: 10.1016/j.cl.2007.05.005

Abstract

Traits offer a fine-grained mechanism to compose classes from reusable components while avoiding problems of fragility brought by multiple inheritance and mixins. Traits as originally proposed are *stateless*, that is, they contain only methods, but no instance variables. State can only be accessed within stateless traits by accessors, which become *required methods* of the trait. Although this approach works reasonably well in practice, it means that many traits, viewed as software components, are artificially *incomplete*, and classes that use such traits may contain significant amounts of boilerplate glue code. We present an approach to stateful traits that is faithful to the guiding principle of stateless traits: *the client retains control of the composition*. Stateful traits consist of a minimal extension to stateless traits in which instance variables are purely local to the scope of a trait, unless they are explicitly made accessible by the composing client of a trait. We demonstrate by means of a formal object calculus that adding state to traits preserves the flattening property: traits contained in a program can be compiled away. We discuss and compare two implementation strategies, and briefly present a case study in which stateful traits have been used to refactor the trait-based version of the Smalltalk collection hierarchy.

⋆ We gratefully acknowledge the financial support of the Swiss National Science Foundation Recast (SNF 2000-061655.00/1), the Cook ANR French projects, and the Science Foundation Ireland and Lero — the Irish Software Engineering Research Centre.

Email addresses: `Alexandre.Bergel@cs.tcd.ie` (Alexandre Bergel),

1 Introduction

Traits are pure units of reuse consisting only of methods [SDNB03, DNS+06]. Traits can be composed to either form other traits or classes. They are recognized for their potential in supporting better composition and reuse, hence their integration in newer 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 been considerable interest in applying traits to statically-typed languages as well [FR03, SD05, NDS06].

Traits make it possible for inheritance to be used to reflect conceptual hierarchy rather than for code reuse. Duplicated code can be factored out as traits, rather than being jimmied into a class hierarchy in awkward locations. At the same time, traits largely avoid the fragility problems introduced by approaches based on multiple inheritance and mixins, since traits are entirely divorced from the inheritance hierarchy.

In their original form, however, traits are *stateless*, *i.e.*, traits are purely groups of methods without any instance variables. Since traits not only provide methods, but may also *require* methods, the idiom introduced to deal with state was to access state only through accessors. The *client* of a trait is either a class or a composite trait that *uses* the trait to build up its implementation. A key principle behind traits is that *the client retains control of the composition.* The client, therefore, is responsible for providing the required methods, and resolving any possible conflicts. Required accessors would propagate to composite traits, and only the composing client class would be required to implement the missing accessors and the instance variables that they give access to. In practice, the accessors and instance variables could easily be generated by a tool, so the fact that traits were stateless posed only a minor nuisance.

Conceptually, however, the lack of state means that virtually all traits are *incomplete*, since just about any useful trait will require some accessors. Furthermore, the mechanism of required methods is abused to cover for the lack of state. As a consequence, the required interface of a trait is cluttered with noise that impedes the understanding and consequently the reuse of a trait. Even if the missing state and accessors can be generated, many clients will consist of "shell classes" — classes that do nothing but compose traits with boilerplate glue code. Furthermore, if the required accessors are made public (as is the case in the Smalltalk implementation), encapsulation is unnecessarily violated in the client classes. Finally, if a trait is ever modified to include additional state, new required accessors will be propagated to all client traits

`stephane.ducasse@univ-savoie.fr` (Stéphane Ducasse), `oscar.nierstrasz@iam.unibe.ch` (Oscar Nierstrasz), `Roel.Wuyts@imec.be` (Roel Wuyts).

and classes, thus introducing a form of fragility that traits were intended to avoid!

This paper describes *stateful traits*, an extension of stateless traits in which a single variable access operator is introduced to give clients of traits control over the visibility of instance variables. The approach is faithful to the guiding principle of stateless traits in which the client of a trait has full control over the composition. It is this principle that is the key to avoiding fragility in the face of change, since no implicit conflict resolution rules come into play when a trait is modified.

In a nutshell, instance variables are private to a trait. The client can decide, however, at composition time to *access* instance variables offered by a used trait, or to *merge* variables offered by multiple traits. In this paper we present an analysis of the limitations of stateless traits and we present our approach to achieving stateful traits. An important property of stateful traits (inherited from stateless version) follows from the way that classes are constructed from traits. The flattening property refers to the fact that in any class defined using traits, the *traits can be inlined* to give an equivalent class definition that does not use traits. We demonstrate this by formally describing stateful traits. Then, we describe and compare two implementation strategies, and we briefly describe our experience with an illustrative case study.

The article is an extension of [BDNW07], presenting a formal model of a Smalltalk-like language, and then showing how stateless traits and statefull traits can be added by "flattening" them down to the base language. The structure of this paper is as follows: First we review stateless traits [SDNB03, DNS+06]. In Section 3 we discuss the limitations of stateless traits. In Section 4 we introduce stateful traits, which support the introduction of state in traits. Section 5 formally describes the semantics of flattening for stateless traits and stateful traits. Section 6 outlines some details of the implementation of stateful traits. In Section 7 we present a small case study in which we compare the results of refactoring the Smalltalk collections hierarchy with both stateless and stateful traits. In Section 8 we discuss some of the broader consequences of the design of stateful traits. Section 9 discusses related work. Section 10 concludes the paper.

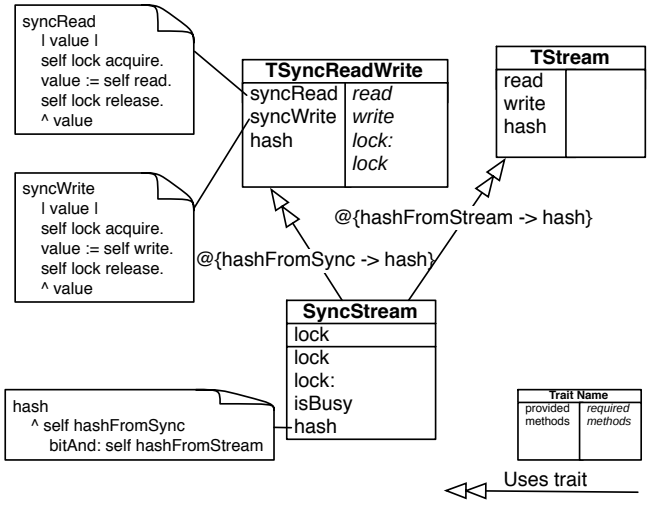

Fig. 1. The class SyncStream is composed of the two traits TSyncReadWrite and TStream

2 Stateless traits

2.1 Reusable groups of methods

Stateless 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.

For example, in Figure 1, 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, where the right column lists required methods while the left one lists the provided methods.

2.2 Composing classes from mixins

The following equation depicts how a class is built with traits:

$$class = superclass + state + trait\ composition + glue\ code$$

A class is specified from a superclass, state definition, a set of traits, and some *glue methods*. Glue methods are defined in the class and they connect the traits together; *i.e.*, they implement required trait methods (often for

accessing state), they adapt provided trait methods, and they resolve method conflicts.

In Figure 1, the class SyncStream defines the field lock and the glue methods lock, lock:, isBusy and hash. The other required methods of TSyncReadWrite, read and write, are also provided since the class SyncStream uses another trait TStream which provides them.

Trait composition respects the following three rules:

- Methods defined in the class take precedence over trait methods. This allows the glue methods defined in a class to override methods with the same name provided by the used traits.
- Flattening property. A non-overridden method in a trait has the same semantics as if it were implemented directly in the class using the trait.
- Composition order is irrelevant. All the traits have the same precedence, and hence conflicting trait methods must be explicitly disambiguated.

With this approach, classes retain their primary role as generators of instances, whereas traits are purely behavioural units of reuse. As with mixins, classes are organized in a single inheritance hierarchy, thus avoiding the key problems of multiple inheritance, but the incremental extensions that classes introduce to their superclasses are specified using one or more traits. In contrast to mixins, several traits can be applied to a class in a single operation: trait composition is unordered. Instead of the trait composition resulting implicitly from the order in which traits are composed (as is the case with mixins), it is fully under the control of the composing class.

2.3 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. Conflicts are resolved by implementing a method at the level of the class that *overrides* the conflicting methods, or by *excluding* a method from all but one trait. In addition traits allow method *aliasing*; this makes it possible for the programmer 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 1, methods in TSyncReadWrite and in TStream are used by SyncStream. The trait composition associated to SyncStream is:
TSyncReadWrite@{hashFromSync→hash} + TStream@{hashFromStream→hash}
This means that 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.

2.4 Method composition operators

The semantics of traits composition is based on four operators: sum, overriding, exclusion and aliasing [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 distinguished conflict method. The + operator is associative and commutative.

The *overriding* 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. This can also be done with methods, as we will discuss in more detail later.

A trait can be constructed by *excluding* 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 @ 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 @ {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 because the body of the aliased method is not changed in any way, so an alias to a recursive method is not recursive.

3 Limitations of stateless traits

Traits support the reuse of coherent groups of methods by otherwise independent classes [DNS+06]. Traits can be composed out of other traits. As a consequence they serve well as a medium for structuring code. Unfortunately stateless traits necessarily encode dependency on state in terms of required methods (*i.e.*, accessors). In essence, traits are necessarily *incomplete* since virtually any useful trait will be forced to define required accessors. This means

that the composing class must define the missing instance variables and accessors.

The incompleteness of traits results in a number of annoying limitations, namely: (i) trait reusability is impacted because the required interface is typically cluttered with uninteresting required accessors, (ii) client classes are forced to implement boilerplate glue code, (iii) the introduction of new state in a trait propagates required accessors to all client classes, and (iv) public accessors break encapsulation of the client class.

Although these annoyances can be largely addressed by proper tool support, they disturb the appeal of traits as a clean, lightweight mechanism for composing classes from reusable components. A proper understanding of these limitations is a prerequisite to entertaining any proposal for a more general approach.

3.1 Limited reusability

The fact that a stateless trait is forced to encode state in terms of required accessors means that it cannot be composed "off-the-shelf" without some additional action. Virtually every useful trait is incomplete, even though the missing part can be trivially fulfilled.

What's worse, however, is the fact that the required interface of a trait is cluttered with dependencies on uninteresting required accessors, rather than focussing attention on the non-trivial hook methods that clients must implement.

Although this problem can be partially alleviated with proper tool support that distinguishes the uninteresting required accessors from the other required methods, the fact remains that traits with required accessors can never be reused off-the-shelf without additional action by the ultimate client class.

3.2 Boilerplate glue code

The necessary additional client action consists essentially in the generation of boilerplate glue code to inject the missing instance variables, accessors and initialization code. Clearly this boilerplate code must be generated for each and every client class. In the most straightforward approach, this will lead to the kind of duplicated code that traits were intended to avoid.

Figure 2 illustrates such a situation where the trait TSyncReadWrite needs to

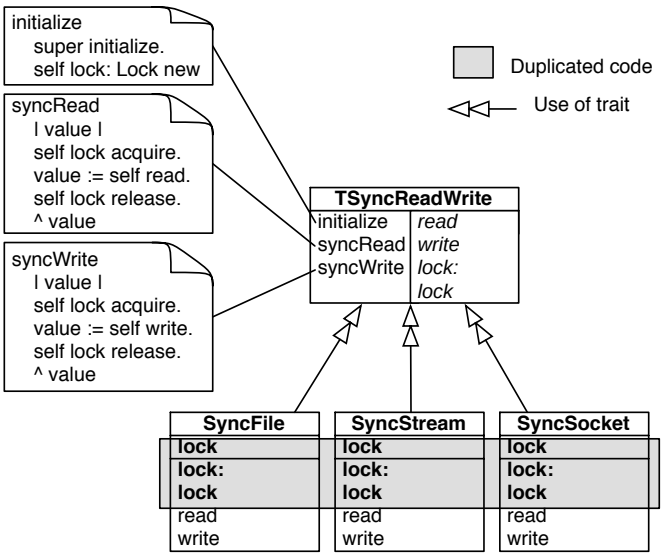

Fig. 2. The lock variable, the lock and lock: methods are duplicated among trait TSyncReadWrite users.

access a lock. This lock variable, the lock accessor and the lock: mutator have to be duplicated in SyncFile, SyncStream and SyncSocket.

Once again, to avoid this situation, tool support would be required (i) to automatically generate the required instance variables and accessors, and (ii) to generate the code in such a way as to avoid actual duplication.

Another unpleasant side effect of the need for boilerplate glue code is the emergence of "shell classes" consisting of nothing but glue code. In the Smalltalk hierarchy refactored using stateless traits [BSD03], we note that 24% (7 out of 29) of the classes in the hierarchy refactored with traits are pure shell classes.

3.3 Propagation of required accessors

If a trait implementation evolves and requires new variables, it may impact all the classes that use it, even if the interface remains untouched. For instance, if the implementation of the trait TSyncReadWrite evolves and requires a new variable numberWaiting intended to give the number of clients waiting for the lock, then all the classes using this trait are impacted, even though the public interface does not change.

Required accessors are propagated and accumulated from trait to trait, therefore when a class is composed of deeply composed traits, a large number of accessors may need to be resolved. When a new state dependency is introduced in a deeply nested trait, required accessors can be propagated to a

large number of client classes. Again, proper tool support can largely mitigate the consequences of such changes, but a more satisfactory solution would be welcome.

3.4 Violation of encapsulation

Stateless traits violate encapsulation in two ways. First of all, stateless traits unnecessarily expose information about their internal representation, thus muddying their interface. A stateless trait exposes every part of its needed representation as a required accessor, even if this information is of no interest to its clients. Encapsulation would be better served if traits resembled more closely abstract classes, where only abstract methods are explicitly declared as being the responsibility of the client subclass. By the same token, a client class using a trait should only see those required methods that are truly its responsibility to implement, and no others.

The second violation is about visibility. In Smalltalk, instance variables are always private. Access can be granted to other objects by providing public accessors. But if traits require accessors, then classes using these traits *must* provide public accessors to the missing state, even if this is not desired.

In principle, this problem could be somewhat mitigated in Java-like languages by including visibility modifiers for stateless traits in Java-like languages. A trait could then require a `private` or `protected` accessor for missing state. The client class could then supply these accessors without violating encapsulation (and optionally relaxing the required modifier). Unlike this approach which is very close to the impementation language, this paper proposes a more principled solution close to the original elegance of the stateless traits model.

4 Stateful traits: reconciling traits and state

We now present stateful traits as our solution to the limitations of stateless traits. Although it may seem that adding instance variables to traits would represent a trivial extension, in fact there are a number of issues that need to be resolved. Briefly, our solution addresses the following concerns:

- Stateless traits should be a special case of stateful traits. The original semantics of stateless traits (and the advantages of that solution) should not be impacted.
- Any extension should be syntactically and semantically minimal. We seek the simplest solution that could possible work.

- We should address the limitations listed in Section 3. In particular, it should be possible to express complete traits. Only methods that are conceptually the responsibility of client classes should be listed as required methods.
- The solution should offer sensible default semantics for trait usage, thus enabling black-box usage.
- Consistent with the guiding principle of stateless traits, the client class should retain control over the composition, in particular over the policy for resolving conflicts. A degree of white-box usage is therefore also supported, where needed.
- As with stateless traits, we seek to avoid fragility with respect to change. Changes to the representation of a trait should normally not affect its clients.
- The solution should be largely language independent. We do not depend on obscure or exotic language features, so the approach should easily apply to most object-oriented languages.

The solution we present extends traits to possibly include instance variables. In a nutshell, there are three aspects to our approach:

(1) Instance variables are, by default, *private* to the scope of the trait that defines them.
(2) The client of a trait, *i.e.*, a class or a composite trait, may *access* selected variables of that trait, mapping those variables to possibly new names. The new names are private to the scope of the client.
(3) The client of a composite trait may *merge* variables of different traits it uses by mapping them to a common name. The new name is private to the scope of the client.

In the following subsections we provide details of the stateful traits model.

4.1 Stateful trait definition

A stateful trait extends a stateless trait by including private instance variables. A stateful trait therefore consists of a group of public methods and private instance variables, and possibly a specification of some additional required methods to be implemented by clients.

Methods. Methods defined in a trait are visible to any other trait with which it is composed. Because methods are public, conflicts may occur when traits are composed. Method conflicts for stateful traits are resolved in the same way as with stateless traits.

Variables. By default, variables are private to the trait that defines them. Because variables are private, conflicts between variables cannot occur when traits are composed. If, for example, traits `T1` and `T2` each define a variable `x`, then the composition of `T1 + T2` does *not* yield a variable conflict. Variables

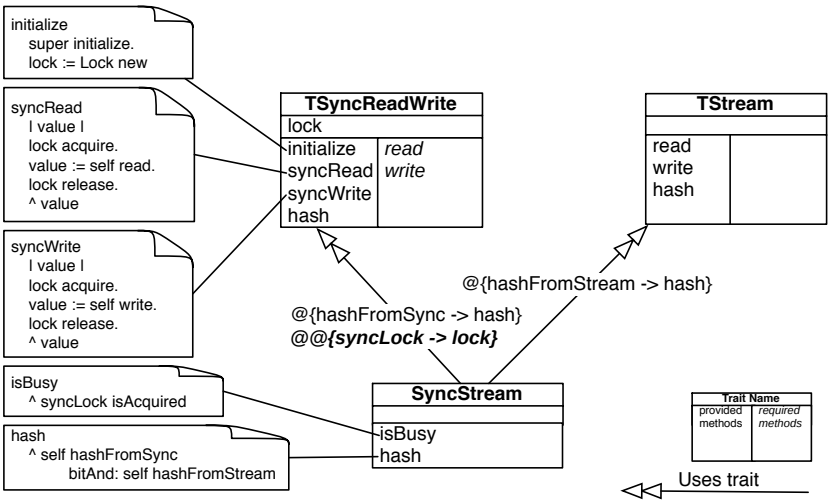

Fig. 3. The class SyncStream is composed of the stateful traits TStream and TSyncReadWrite.

are only visible to the trait that defines them, unless access is widened by the composing client trait or class with the @@ variable access operator.

Figure 3 shows how the example presented in Figure 1 is reimplemented using stateful traits. The class SyncStream is composed of the traits TStream and TSyncReadWrite. The trait TSyncReadWrite defines the variable lock, three methods syncRead, syncWrite and hash, and requires methods read and write.

Note that, in order to include state in traits, we must extend the mechanism for defining traits. In the Smalltalk implementation, this is achieved by extending the message sent to the Trait class with a new keyword argument to represent the used instance variables. For instance, we can now define the TSyncReadWrite trait as follows:

```
Trait named: #TSyncReadWrite
    uses: {}
    instVarNames: 'lock'
```

The trait TSyncReadWrite is not composed of any other traits and it defines a variable lock. The uses: clause specifies the trait composition (empty in this case), and instVarNames: lists the variables defined in the trait (*i.e.*, the variable, lock). The interface for defining a class as composition of traits is the same as with stateless traits. The only difference is that the trait composition expression supports an additional operator (@@) for granting access to variables of the used traits. Here we see how SyncStream is composed from the traits TSyncReadWrite and TStream:

```
Object subclass: #SyncStream
    uses: TSyncReadWrite @ {#hashFromSync →#hash}
```

```
        @@ {syncLock →lock}
    + TStream @ {#hashFromStream →#hash}
instVarNames: ''
....
```

In this example, access is granted to the lock variable of the TSyncReadWrite trait under the new name syncLock. As we shall now see, the @@ operator provides a fine degree of control over the visibility of trait variables.

4.2 Variable access

By default, a variable is private to the trait that defines it. However, the variable access operator (@@) allows variables to be *accessed* from clients under a possibly new name, and possibly *merged* with other variables.

If T is a trait that defines a (private) instance variable x, then T@@{y →x} represents a new trait in which the variable x can be accessed from its client scope under the name y. x and y represent the same variable, but the name x is restricted to the scope of t whereas the name y is visible to the enclosing client scope (*i.e.*, the composing classscope). For instance, in the following composition:

TSyncReadWrite@{hashFromSync →hash} @@{syncLock →lock}

the variable lock defined in TSyncReadWrite is accessible to the class SyncStream using that trait under the name syncLock. (Note that renaming is often needed to distinguish similarly named variables coming from different used traits.)

In a trait variable composition, three situations can arise: (i) variables remain private (*i.e.*, the variable access operator is not used), (ii) access to a private variable is granted, and (iii) variables are merged.

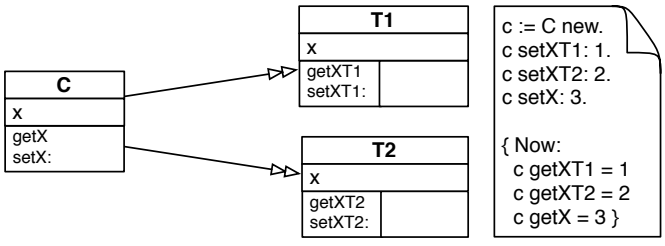

Fig. 4. Keeping variables private: while composed, variables are kept separate. Traits T1, T2 and C have their own variable x.

4.2.1 Keeping variables private.

By default, instance variables are private to their trait. If the scope of variables is not broadened at composition time using the variable access operator, conflicts do not occur and the traits do not share state. Figure 4 shows a case where T1 and T2 are composed without variable access being broadened. Each of these two traits defines a variable x. In addition they each define accessor methods. C also defines a variable x and two methods getX and setX:. T1, T2 and C each have their own variable x as shown in Figure 4.

The trait composition of C is: T1 + T2. Note that if methods would conflict we would use the default trait strategy to resolve them by locally redefining them in C and that method aliasing could be used to access the overridden methods.

This form of composition is close to the module composition approach proposed in Jigsaw [Bra92] and supports a black-box reuse scenario.

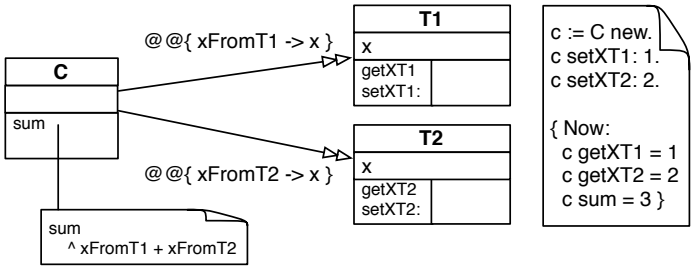

Fig. 5. Granting access to variables: x of T1 and T2 are given access in C.

4.2.2 Granting variable access.

Figure 5 shows how the client class C gains access to the private x variables of traits T1 and T2 by using the variable access operator @@. Because two variables cannot have the same name within a given scope, these variables have to be renamed. The variable x from T1 is accessible as xFromT1 and x from T2 is accessible as xFromT2. C also defines a method sum that returns the value xFromT1 + xFromT2. The trait composition of C is:

T1 @@ {xFromT1 →x} + T2 @@ {xFromT2 →x}

C can therefore build functionality on top of the traits that it uses, without exposing any details to the outside. Note that methods in the trait continue to use the 'internal' name of the variable as defined in the trait. The name given in the variable access operator @@ is only to be used in the client classes. This is similar to the method aliasing operator @.

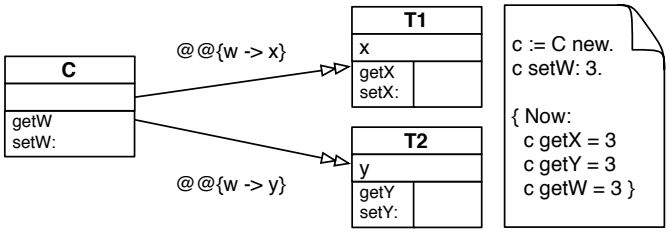

Fig. 6. Merging variables: variables x and y are merged in C under the name w.

4.2.3 Merging variables.

Variables from several traits can be merged when they are composed by using the variable access operator to map multiple variables to a *common name* within the client scope. This is illustrated in Figure 6.

Both T1 and T2 give access to their instance variables x and y under the name w. This means that w is shared between all three traits. This is the reason why sending getX, getY, or getW to an instance of a class implementing C returns the same result, 3. The trait composition of C is:

T1 @@ {w →x} + T2 @@ {w →y}

Note that merging is *fully* under the control of the client class or trait. There can be no accidental name capture since visibility of instance variables is never propagated to an enclosing scope. Variable name conflicts cannot arise, since variables are private to traits unless they are explicitly accessed by clients, and variables are merged when they are mapped to common names.

The reader might well ask, what happens if the client also defines an instance variable whose name happens to match the name under which a used trait's variable is accessed? Suppose, for example, that C in Figure 6 attempts to additionally define an instance variable called w. We consider this to be an error. This situation cannot possibly arise as a side effect of changing the definition of a used trait since the client has full control over the names of instance variables accessible within its scope. As a consequence this cannot be a case of accidental name capture, and can only be interpreted as an error.

4.3 Requirements revisited

Let us briefly reconsider our requirements. First, stateful traits do not change the semantics of stateless traits. Stateless traits are purely a special case of stateful traits. Syntactically and semantically, stateful traits represent only a minor extension of stateless traits.

Stateful traits address the issues raised in Section 3. In particular, (i) there is no longer a need to clutter trait interfaces with required accessors, (ii) clients no longer need to provide boilerplate instance variables and accessors, (iii) the introduction of state in traits remains private to that trait, and (iv) no public accessors need be introduced in client classes. As a consequence, it is possible to define "complete" traits that require no methods, even though they make use of state.

The default semantics of stateful traits enables black-box usage since no representation is exposed, and instance variables by default cannot clash with those of the client or of other used traits. Nevertheless, the client retains control of the composition, and can gain access to the instance variables of used traits. In particular, the client may merge variables of traits, if this is desired.

Since the client retains full control of the composition, changes to the definition of a trait cannot propagate beyond its direct clients. There can be no implicit side effects.

Finally, the approach is largely language-independent. In particular, there are no assumptions that the host language provide either access modifiers for instance variables or exotic scoping mechanisms.

5 Flattening Property

A key feature of stateless traits is that they can be *flattened* [DNS+06]. This means that adding traits to a language does not require any change to the operational semantics of the underlying language, and in particular does not require a change to the method lookup semantics. In principle, traits can be compiled away.

We demonstrate the flattening property for stateful traits by defining a flattening function for a minimal object-oriented language. The approach is similar to that used previously to give a semantics for stateless traits for statically typed object-oriented languages [NDS06]. However, instead of using FeatherweightJava as a the core language, we must use a language with state. We therefore 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 (stateful) ClassicJava model they introduce to develop SmalltalkLite, a simple calculus that captures the key features of Smalltalk-like dynamic languages. We similarly adapt their treatment of mixins to model traits.

We first present SmalltalkLite, a Smalltalk-like dynamic language featuring single inheritance, message-passing, field access and update, and **self** and

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

Fig. 7. SmalltalkLite syntax

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 then extend SmalltalkLite with stateless traits by specifying a flattening function similar to that previously specified for FeatherweightJava with traits [NDS06].

Finally we specify SmalltalkLite with stateful traits by defining a new flattening function that takes fields into account. The key feature of this new flattening function is that fields remain purely local to the traits in which they are defined, unless they are explicitly exposed by an **access** declaration. Conflicts therefore do not arise. Explicitly exposed fields whose names collide are *merged* by the flattening function. Merging can be inhibited by aliasing a field name.

5.1 SmalltalkLite

The syntax of SmalltalkLite is shown in Figure 7. SmalltalkLite is similar to ClassicJava, but eliding the features related to static typing. We similarly ignore features that are not relevant to a discussion of stateful 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 9. 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 8. (The predicates ClassesOnce(P) *etc.*, are assumed to be preconditions for valid programs, and are not otherwise explicitly mentioned in the reduction

$\prec_P$	Direct subclass
	$c \prec_P c' \iff \textbf{class}\ c\ \textbf{extends}\ c' \cdots \{\cdots\} \in P$
$\leq_P$	Indirect subclass
	$c \leq_P c' \quad \equiv \quad$ transitive, reflexive closure of $\prec_P$
$\in_P$	Field defined in class
	$f \in_P c \iff \textbf{class} \cdots \{\cdots f \cdots\} \in P$
$\in_P$	Method defined in class
	$\langle m, x^*, e\rangle \in_P c \iff \textbf{class} \cdots \{\cdots m(x^*)\{e\} \cdots\} \in P$
$\in_P^*$	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
	$\langle c, m, x^*, e\rangle \in_P^* c' \iff c' = min\{c'' \mid \langle m, x^*, e\rangle \in_P c'', c \leq_P c''\}$
ClassesOnce(P)	Each class name is declared only once
	$\forall c, c',\ \textbf{class}\ c \cdots \textbf{class}\ c' \cdots$ is in $P \Rightarrow c \neq c'$
FieldOncePerClass(P)	Field names are unique within a class declaration
	$\forall f, f',\ \textbf{class}\ c \cdots \{\cdots f \cdots f' \cdots\}$ 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
	$\forall m, m',\ \textbf{class}\ c \cdots \{\cdots m(\cdots)\{\cdots\} \cdots m'(\cdots)\{\cdots\} \cdots\}$ is in $P \Rightarrow m \neq m'$
CompleteClasses(P)	Classes that are extended are defined
	range($\prec_P$) $\subseteq$ dom($\prec_P$) $\cup$ {Object}
WellFoundedClasses(P)	Class hierarchy is an order
	$\leq_P$ is antisymmetric
ClassMethodsOK(P)	Method overriding preserves arity
	$\forall m, m', \langle m, x_1 \cdots x_j, e\rangle \in_P c, \langle m, x'_1 \cdots x'_k, e'\rangle \in_P c', c \leq_P c' \implies j = k$

Fig. 8. Relations and predicates for SmalltalkLite

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

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

Fig. 9. Redex syntax

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

Fig. 10. Translating expressions to redexes

expression e in which field names are decorated with their object contexts, *i.e.*, f is translated to $o.f$, and **super** calls 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.

The store consists of a set of mappings from object identifiers $\mathit{oid} \in \mathrm{dom}(\mathcal{S})$ 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 10). 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 11.

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

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

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

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

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

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

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

Fig. 11. Reductions for SmalltalkLite

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

Fig. 12. Variable substitution

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 12). 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

$$
\begin{aligned}
\mathit{defn} &= \textbf{class}\ c\ \textbf{extends}\ c\ \{\ f^{*}\mathit{meth}^{*}\tau^{*}\ \} \\
&\mid\ \textbf{trait}\ t\ \{\ \mathit{meth}^{*}\tau^{*}\ \} \\
\tau &= t\ \mid \tau\ \textbf{alias}\ m' \rightarrow m\ \mid \tau\ \textbf{minus}\ m \\
t &= \text{a trait name}
\end{aligned}
$$

Fig. 13. Adding syntax for traits to SmalltalkLite

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 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.

5.2 Stateless traits

We now add (stateless) traits to SmalltalkLite by (i) specifying an extended syntax for SmalltalkLite with traits (Figure 13), and (ii) specifying a translation from programs with traits back to the core language without traits (Figure 14).

We distinguish a named trait t from a trait expression τ which may alias or exclude 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 define an alias m' for an existing method m, or it may exclude a method m.

We give a semantics to SmalltalkLite with traits by *flattening* traits to plain SmalltalkLite. The translation expands trait expressions to method declarations. The translation is valid if the resulting classes contain no conflicts. (Intermediate trait expressions may contain conflicts, as long as these are resolved by the client classes.)

The translation is specified in terms of four operators over stateless traits (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

$$
\begin{aligned}
[\![def1 \cdots defn]\!] &= [\![def1]\!] \cdots [\![defn]\!] \\
[\![\textbf{trait } t \ \{ \ meth^* \tau^* \ \}]\!] &= \emptyset \\
[\![\textbf{class } c \textbf{ extends } c' \ \{ \ f^* meth^* \tau^* \ \}]\!] &= \textbf{class } c \textbf{ extends } c' \ \{ \ f^* meth^* \triangleright [\![\tau^*]\!] \ \} \\
[\![\tau_1 \cdots \tau_k]\!] &= [\![\tau_1]\!] + \cdots + [\![\tau_k]\!] \\
[\![t]\!] &= \begin{cases} meth^* & \text{if } \textbf{trait } t \ \{ \ meth^* \} \in P \\ meth^* \triangleright [\![\tau^+]\!] & \text{if } \textbf{trait } t \ \{ \ meth^* \tau^+ \ \} \in P \end{cases} \\
[\![\tau \textbf{ alias } m' \to m]\!] &= [\![\tau]\!][m' \to m] \\
[\![\tau \textbf{ minus } m]\!] &= [\![\tau]\!] - m
\end{aligned}
$$

Fig. 14. Flattening SmalltalkLite with stateless traits to SmalltalkLite

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

Fig. 15. Trait operations

already has 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.

5.3 Stateful traits

The syntax for SmalltalkLite with stateful traits is shown in Figure 16. All we change is that trait declarations may include fields, and trait expressions may widen access to a field.

$$
\begin{aligned}
\mathit{defn} &= \textbf{class}\ c\ \textbf{extends}\ c\ \{\ f^*\mathit{meth}^*\tau^*\ \} \\
&\mid\ \textbf{trait}\ t\ \{\ f^*\mathit{meth}^*\tau^*\ \} \\
\tau &= t\ \mid\ \tau\ \textbf{alias}\ m' \to m\ \mid\ \tau\ \textbf{minus}\ m\ \mid\ \tau\ \textbf{access}\ f' \to f
\end{aligned}
$$

Fig. 16. SmalltalkLite with stateful traits syntax

$$
\begin{aligned}
[\![\mathit{def}1 \cdots \mathit{def}n]\!] &= [\![\mathit{def}1]\!] \cdots [\![\mathit{def}n]\!] \\
[\![\textbf{trait}\ t\ \{f^*\ \mathit{meth}^*\tau^*\ \}]\!] &= \emptyset \\
[\![\textbf{class}\ c\ \textbf{extends}\ c'\ \{\ f^*\mathit{meth}^*\tau^*\ \}]\!] &= \textbf{class}\ c\ \textbf{extends}\ c'\ \{\ f^*\mathit{meth}^* \triangleright [\![\tau^*]\!]_\emptyset\ \} \\
[\![\tau_1 \cdots \tau_k]\!]_F &= [\![\tau_1]\!]_F + \cdots + [\![\tau_1]\!]_F \\
[\![t]\!]_F &= \mu(\cdots f_i' \cdots m_j(x_j^*)\{e_j[f_i'/f_i]\} \cdots \triangleright [\![\tau^*]\!]_F) \\
&\quad \text{where} \\
&\quad \textbf{trait}\ t\ \{\cdots f_i \cdots m_j(x_j^*)\{e_j\} \cdots \tau^*\} \\
&\quad \text{and} \begin{cases} f_i' = f_i & \text{if } f_i \in F \\ f_i' \text{ is fresh} & \text{otherwise} \end{cases} \\
[\![\tau\ \textbf{alias}\ m' \to m]\!]_F &= [\![\tau]\!]_F[m \to m'] \\
[\![\tau\ \textbf{minus}\ m]\!]_F &= [\![\tau]\!]_F - m \\
[\![\tau\ \textbf{access}\ f' \to f]\!]_F &= f'\ ([\![\tau]\!]_{F \cup \{f\}} \setminus f)[f'/f]
\end{aligned}
$$

Fig. 17. Flattening SmalltalkLite with stateful traits to SmalltalkLite

A flattened trait now returns a list of fields and methods. The flattening function allows a field to be exposed. Once exposed, this field might be freely renamed. Fields that are not explicitly exposed to the composite entity are alpha-renamed, thus hiding them. $[\![t]\!]_F$ hides all field names except those in F. (See Figure 17.) A field is renamed using the field substitution mechanism (Figure 19), that substitutes all occurrences of the name of a field with a new name. Fields that are not hidden may end up multiply defined, so we explicitly *merge* them. $\mu(f^*\mathit{meth}^*)$ eliminates multiple declarations of any field f, thus guaranteeing the precondition FieldOncePerClass(P) (See Figure 18).

A field declared as accessible within a trait has no special status — any further classes or traits using that trait must again declare it to be accessible or it will be hidden at the next level.

As an example, let's assume we have a trait TColor defining a field color, and two methods changeToWebColor() and isPrimaryColor():

$$\mu(f_i^* meth^*) = f_j'^* meth^* \text{ where } f_j' \in \{f_i\}$$

Fig. 18. Merging fields

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

Fig. 19. Field substitution

```
trait TColor {
    color
    changeToWebColor () { color=... }
    isPrimaryColor () { ...color... }
}
```

The trait TColor defines a coloring concern. A graphical widget that needs to have a colored border line may use TColor a first time to define the color of the widget, and a second time to give a color to the borderline. A class ColoredRectangleWithBorderLine that uses this trait twice can be defined as follows:

```
class ColoredRectangleWithBorderLine extends Object {
    TColor access color → color
    TColor access borderColor → color
        alias changeBorderToWebColor → changeToWebColor
        alias doesBorderUsePrimaryColor → isPrimaryColor
        minus changeToWebColor
        minus isPrimaryColor
    draw () { ... } "use color and borderColor for the drawing"
}
```

Once flattened, the class ColoredRectangleWithBorderLine is equivalent to:

```
class ColoredRectangleWithBorderLine extends Object {
    color
    borderColor
    changeToWebColor() { color=... }
    isPrimaryColor () { ...color... }
    changeBorderToWebColor() { borderColor=... }
    doesBorderUsePrimaryColor () { ...borderColor... }
    draw () { ... } "use color and borderColor for the drawing"
}
```

The expression TColor access borderColor → color is flattened to {borderColor, changeToWebColor () { borderColor=... }, isPrimaryColor () { ...borderColor... }}. Then changeToWebColor is aliased to changeBorderToWebColor and isPrimaryColor to doesBorderUsePrimaryColor. Note that an alias creates a new entry in the method dictionary, leaving the original name accessible. Conflict with the first use of TColor is avoided by removing changeToWebColor and isPrimaryColor.

6 Implementation

We have implemented a prototype of stateful traits as an extension of our Smalltalk-based implementation of stateless traits. [1]

As with stateless traits, method composition and reuse for stateful traits do not incur any overhead since method pointers are shared between method dictionaries of different traits and classes. This takes advantage of the fact that methods are looked up by name in the dictionary rather than accessed by index and offset, as is done to access state in most object-oriented programming languages. However, by adding state to traits, we have to find a solution to the fact that the access to instance variables cannot be linear (*i.e.*, based on offsets) since the same trait methods can be applied to different objects [BBG+02]. A linear structure for state representation cannot be always obtained from a composition graph. This is a common problem of languages that support multiple inheritance. We evaluated two implementations: copy-down and changing object internal representation. The following section illustrates the problem.

6.1 The classical problem of state linearization

As pointed out by Bracha [Bra92, Chapter 7], in implementations of single inheritance languages such as Modula-3 [CDG+92], and more recently in the Jikes Research Virtual Machine [Jik], the notion of virtual functions is supported by associating with each class a table whose entries are the addresses of the methods defined for instances of that class. Each instance of a class contains a reference to the class method table. It is through this reference that the appropriate method to be invoked on an instance is located. Under multiple inheritance, this technique must be modified, since the superclasses of a class no longer share a common prefix.

Since a stateful trait can have private state, and can be used in multiple

[1] See `www.iam.unibe.ch/~scg/Research/Traits`

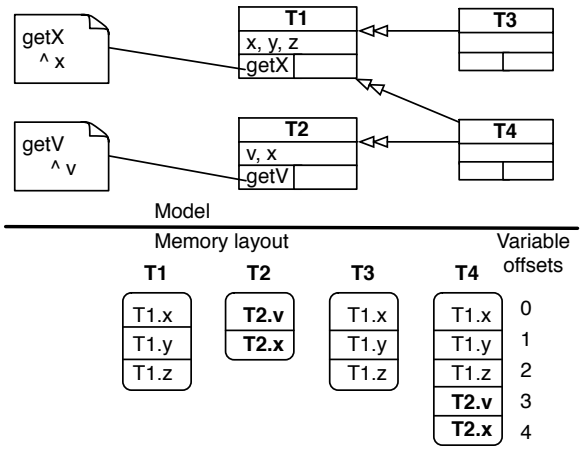

Fig. 20. Problem of combining multiple traits: variable's offset is not preserved.

contexts, it is not possible to have a static and linear instance variable offset list shared by all the methods of the trait and its users.

The top half of Figure 20 shows a trait T3 using T1 and a trait T4 using T1 and T2. T1 defines 3 variables x, y, z and T2 defines 2 variables v, x. The bottom part shows a possible corresponding representation in memory that uses offsets. Assuming that we start the indexing at zero, T2.v has zero for index, and T2.x has one. However, in T4 the same two variables might have indexes three and four. [2] So static indexes used in methods from T1 or T2 are no longer valid. Note that this problem occurs regardless of the composition of trait T4 out of traits T1 and T2 (whether it needs access to variables, whether or not it merges variable x, ...). The problem is due to the linear representation of variables in the underlying object model.

6.2 Three approaches to state linearization

Three different approaches are available to represent non linear state. C++ uses intra-object pointers [SG99]. Strongtalk [BBG+02] uses a *copy-down* technique that duplicates methods that need to access variable with different offset. A third approach, as done in Python [Pyt] for example, is to keep variables in a dictionary and look them up, similar to what is done for methods.

We implemented the last two approaches for Smalltalk so that we could compare them for our prototype implementation. We did not implement C++'s solution because it would require significant effort to change the object representation to be compatible.

[2] We assume that the slots of T2 are added after the ones of T1. In the opposite case the argument holds for the variables of T1.

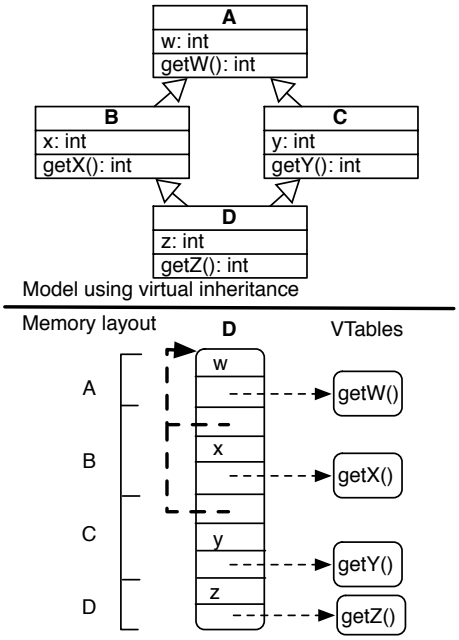

Fig. 21. Multiple virtual inheritance in C++.

6.3 Virtual base pointers in C++

In C++ [SE90], an instance of a class C is represented by concatenating the representations of superclasses of C. Such instance is therefore composed of *subobjects*, where each *subobject* corresponds to a particular *superclass*. Each subobject has its own pointer to a suitable method table. In this case, the representation of a class is not a prefix of the representations of all of its subclasses.

Each subobject begins at a different offset from the beginning of the complete C object. These offsets, called *virtual base pointers* [SG99], can be computed statically. This technique was pioneered by Krogdahl [Kro85, Bra92].

For instance, let's consider the situation in C++ illustrated in Figure 21. The upper part of the figure shows a classical diamond diagram using virtual inheritance (*i.e.*,B and C inherit virtually A, therefore the w variable is shared between B and C). The lower part shows the memory layout of an instance of D. This instance is composed of 4 "sub-parts" corresponding to the superclasses A, B, C and D. Note that C's part, instead of assuming that the state it inherits from A lies immediately "above" its own state, accesses the inherited state via the virtual base pointer. In this way the B and C parts of the D instance can share the same common state from A.

We did not attempt to implement this strategy in our Smalltalk prototype, as it would have required a deep modification to the Smalltalk virtual machine. Since Smalltalk supports only single inheritance, object layout is fundamen-

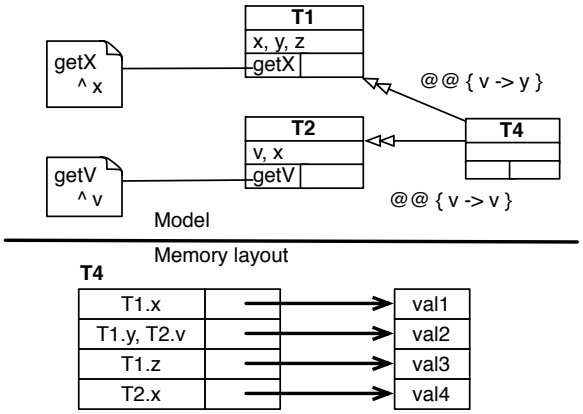

Fig. 22. Structure of objects is similar to a hash table with multiple keys for a same entry.

tally simpler. Accommodating virtual base pointers in the layout of an object would also entail changes to the method lookup algorithm.

6.4 Object state as a dictionary

An alternative implementation approach is to introduce instance variable accesses based on names and not on offsets. The variable layout has the semantics of a hash table, rather than that of an array. For a given variable, its offset is not constant anymore as shown by Figure 22. The state of an object is implemented by a hash table in which multiple keys may map to the same value. For instance, variable y of T1 and variable v of T2 are merged in T4. Therefore, an instance of T4 has two variables (keys), T1.y and T2.v, that actually point to the same value.

In Python [Pyt] the state of an object is represented by a dictionary. An expression such as self.name = value is translated into self.__dict__[name] = value, where __dict__ is a primitive to access the dictionary of an object. A variable is declared and defined simply by being used in Python. For instance, affecting a value to an non-existing variable has the effect to create a new variable. Representing the state of an object with a dictionary is a way to deal with the linearization problem of multiple inheritance.

6.5 Copy down methods

Strongtalk [BBG+02] is a high performance Smalltalk with a mixin-aware virtual machine. A mixin contains a description of its instance variables and class variables, and a method dictionary where all the code is initially stored. One of the problems when sharing code among mixin application is that the

physical layout of instances varies between mixin applications. This problem is addressed by the *copy down* mechanism: (i) Methods that do not access instance variables or super are shared in the mixin. (ii) Methods that access instance variables may have to be copied if the variable layout differs from that of other users of the mixin.

The copy down mechanism favors execution speed over memory consumption. There is no extra overhead to access variables. Variables are linearly ordered, and methods that access them are duplicated and adjusted with proper offset access. Moreover, in Strongtalk, only accessors are allowed to touch instance variables directly at the byte code level. The space overhead of copy-down is therefore minimal. Effective inlining by the virtual machine takes care of the rest, except for accessors which impose no space overhead.

The dictionary-based approach has the advantage that it more directly reflects the semantics of stateful traits, and is therefore attractive for a prototype implementation. Practical performance could however become problematic, even with optimized dictionary implementations like in Python [Pyt]. The copy-down approach, however, is clearly the better approach for a fast implementation. Therefore we decided to adopt it in our implementation of stateful traits in Squeak Smalltalk.

6.6 Benchmarks

As mentioned in the previous section, we adopted the copy-down technique for our stateful traits implementation. In this section we compare the performance of our stateful traits prototype implementation with that of both regular Squeak without traits and that of the stateless traits implementation. We measured the performance of the following two case studies:

- the SyncStream example introduced in the beginning of the paper. The experiment consisted of writing and reading large objects in a stream 1000 times. This example was chosen to evaluate whether state is accessed efficiently.
- a link checker application that parses HTML pages to check whether URLs on a webpage are reachable or not. This entails parsing large HTML files into a tree representation and running visitors over these trees. This case study was chosen in order to have a more balanced example that consists of accessing methods as well as state.

For both case studies we compared the stateful implementation with the stateless traits implementation and with regular Squeak. The results are shown in Table 1.

	Without traits	Stateless traits	Stateful traits
SyncStream	13912	13913	13912
LinkChecker	2564	2563	2564

Table 1
Execution times of two cases for three implementations: without traits, with stateless traits and with stateful traits (times in milliseconds).

As can be seen from the table, no overhead is introduced by accessing instance variables defined in traits and used in clients. This was to be expected: the access is still offset-based and almost no differences can be noticed. Regarding overall execution speed, we see that there is essentially no difference between the three implementations. This result is consistent with previous experiences with traits, and was to be expected since we did not change the parts of the implementation dealing with methods.

7 Refactoring the Smalltalk collection hierarchy

We have carried out a case study in which we used stateful traits to refactor the Smalltalk collection hierarchy. We have previously used stateless traits to refactor the same hierarchy [BSD03], and we now compare the results of the two refactorings. The stateless trait-based Smalltalk collection hierarchy consists of 29 classes which are built from a total of 52 traits. Among these 29 classes there are numerous classes, which we call *shell* classes, that only declare variables and define their associated accessors. Seven classes of the 29 classes (24%) are shell classes (SkipList, PluggableSet, LinkedList, OrderedCollection, Heap, Text and Dictionary).

The refactoring with stateful traits results in a redistribution of the variables defined (in classes) to the traits that effectively need and use them. Another consequence is the decrease of number of required methods and a better encapsulation of the traits behaviour and internal representation.

Figure 23 shows a typical case arising with stateless traits where the class Heap must define 3 variables (array, tally, and sortBlock). The behaviour of this class is limited to the initialization of objects and providing accessors for each of these variables. It uses the trait THeapImpl, which requires all these accessors. These requirements are necessary for THeapImpl since it is composed of TArrayBased and TSortBlockBased which require such state. These two traits need access to the state defined in Heap.

Figure 24 shows how Heap is refactored to use stateful traits. All variables

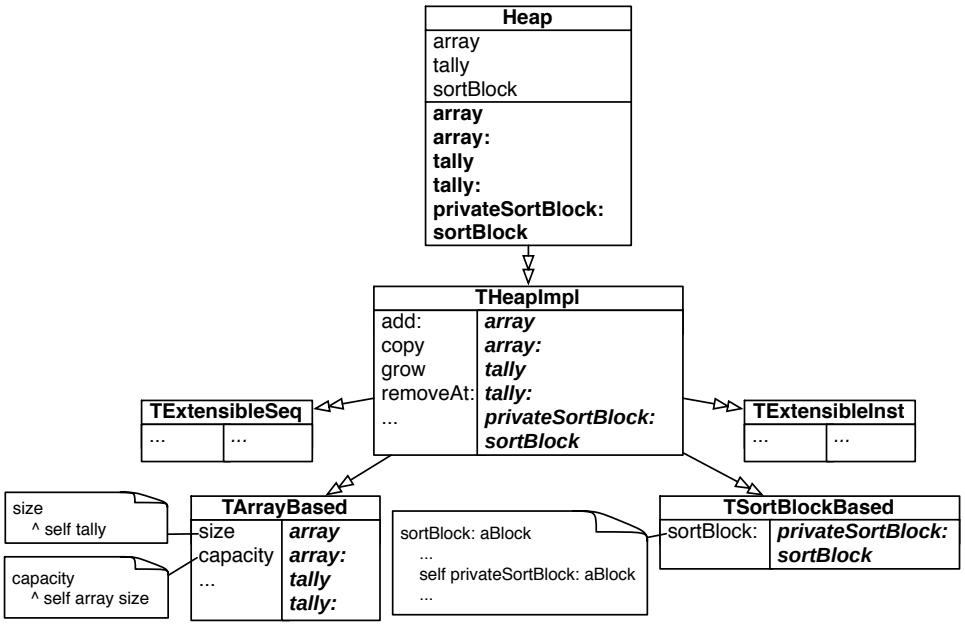

Fig. 23. Fragment of the stateless trait Smalltalk collection hierarchy. The class Heap defines variables used by TArrayBased and TSortBlockBased.

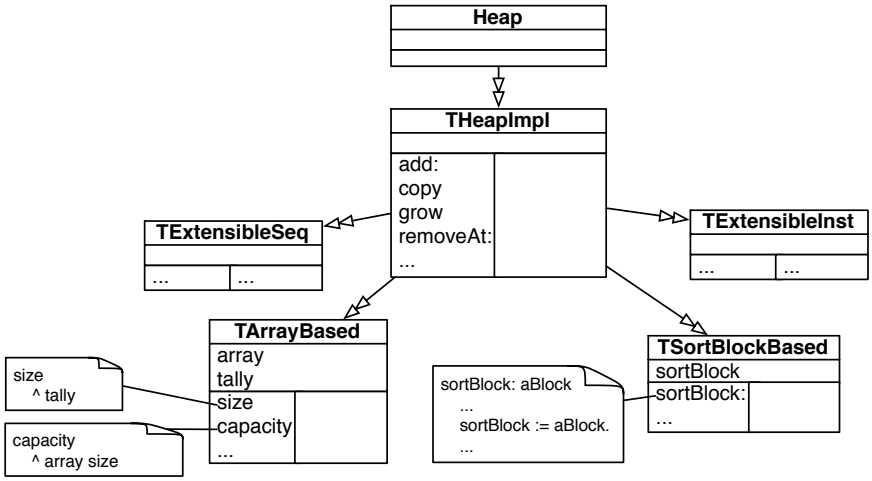

Fig. 24. Refactoring of the class Heap with stateful traits but keeping the trait THeapImpl.

have been moved to the places where they were needed, leading to the result that Heap becomes empty. The variables previously defined in Heap are now defined in those traits that effectively require them. TArrayBased defines two variables array and tally, therefore it does not need to specify any accessors as required methods. The same happens for the variable sortBlock in the trait TSortBlockBased.

If we are sure that THeapImpl is not used by any other class or trait, then we can further simplify this new composition by moving the implementation of

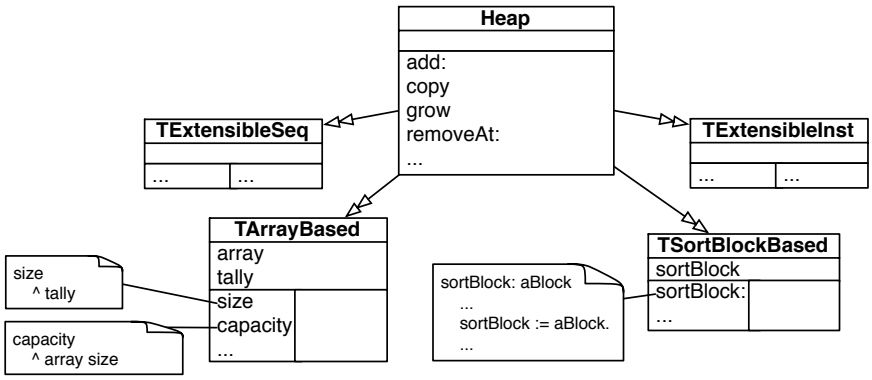

Fig. 25. Refactoring of the class Heap with stateful traits removing the trait THeapImpl.

the trait THeapImpl to Heap and eliminating THeapImpl alltogether. Figure 25 shows the resulting hierarchy. The class Heap defines methods like add: and copy.

Refactoring the Smalltalk class hierarchy using stateful traits yields multiple benefits:

- *Encapsulation is preserved:* Internal representation is not unnecessarily revealed to client classes.
- *Fewer method definitions:* Unnecessary variable accessors are avoided. Accessors that were defined in Heap are removed.
- *Fewer method requirements:* Since variables are defined in the traits that used them, we avoid specifying required accessors. Variable accessors for THeapImpl, TArrayBased, and TSortBlockBased are not required anymore. There is no propagation of required methods due to state usage.

8 Discussion

8.1 Flattening property

In the original stateless trait model [DNS+06], trait composition respects the *flattening property*, which states that a non-overridden method in a trait has the same semantics as if it were implemented directly in the class. This implies that traits can be inlined to give an equivalent class definition that does not use traits. It is natural to ask whether such an important property is preserved with stateful traits. In short, the answer is yes, though trait variables may have to be alpha-renamed to avoid name clashes.

In order to preserve the flattening property with stateful traits, we must en-

sure that instance variables introduced by traits remain private to the scope of that trait's methods, even when their scope is broadened to that of the composing class. This can be done in a variety of ways, depending on the scoping mechanisms provided by the host language. Semantically, however, the simplest approach is to alpha-rename the private instance variables of the trait to names that are unique in the client's scope. Technically, this could be achieved by the common technique of *name-mangling*, *i.e.*, by prepending the trait's name to the variable's name when inserting it in the client's scope. Renaming and merging are also consistent with flattening, since variables can simply be renamed or merged in the client's scope.

8.2 Limiting change impact

Any approach to composing software is bound to be fragile with respect to certain kinds of change: if a feature that is used by several clients changes, the change will affect the clients. Extending a trait so that it provides additional methods may well affect clients by introducing new conflicts. However, the design of trait composition based on explicit resolution ensures that such changes cannot lead to implicit and unexpected changes in the behaviour of direct or indirect clients. A direct client can generally resolve a conflict without changing or introducing any other traits, so no ripple effect will occur [DNS+06].

In stateful traits adding a variable to a trait does not affect clients because variables are private. Removing or renaming a variable may require its direct clients to be adapted only if this variable is explicitly accessed by these clients. However, once the direct clients have been adapted, no ripple effect can occur in indirect clients. By avoiding required method propagation, stateful traits limit the effect of changes.

8.3 About variable access

By default a trait variable is private, thereby enforcing black-box reuse. At the same time we offer an operator enabling the direct client to access the private variables of the trait. This may appear to be a violation of encapsulation [Sny86]. However this approach is consistent with our vision that traits serve as building blocks for composing classes, whether in a black-box or a white-box fashion. Furthermore it is consistent with the principle that the client of a trait is in control of the composition. It is precisely this fact that ensures that the effects of changes do not propagate to remote corners of the class hierarchy.

9 Related work

We briefly review some of the numerous research activities that are relevant to stateful traits.

Self. The prototype based language Self [US87] does not have a notion of class. Conceptually, each object defines its own format, methods, and delegation relations. Objects are derived from other objects by cloning and modification. Objects can have one or more parent objects; messages that are not found in the object are looked for and delegated to a parent object. Self is based around the notion of slots, which unifies methods and instance variables.

Self uses trait objects to factor out common features [UCCH91]. Nothing prevents a trait object from also containing state. Similar to the notion of traits presented here, these trait objects are essentially groups of methods. But unlike our traits, Self's trait objects do not support specific composition operators; instead, they are used as ordinary parent objects.

Interfaces with default implementation. Mohnen [Moh02] proposed an extension of Java in which interfaces can be equipped with a set of default implementations of methods. As such, classes that implement such an interface can explicitly state that they want to use the default implementation offered by that interface (if any). If more than one interface mentions the same method, a method body must be provided. Conflicts are flagged automatically, but require the developer to resolve them manually. State cannot be associated with the interfaces. Scala [sca] also supports traits *i.e.*, partially defined interfaces. While the composition of traits in Scala does not follow exactly the one in stateless traits, traits in Scala cannot define state.

Mixins. Mixins [BC90] use the ordinary single inheritance operator to extend various parent classes with a bundled set of features. Although this inheritance operator is well-suited for deriving new classes from existing ones, it is not necessarily appropriate for composing reusable building blocks. Specifically, because mixin composition is implemented using single inheritance, mixins are composed linearly. This gives rise to several problems. First, a suitable total ordering of features may be difficult to find, or may not even exist. Second,"glue code" that exploits or adapts the linear composition may be dispersed throughout the class hierarchy. Third, the resulting class hierarchies are often fragile with respect to change, so that conceptually simple changes may impact many parts of the hierarchy [DNS+06].

Eiffel. Eiffel [Mey92] is a pure object-oriented language that supports multiple inheritance. Features, *i.e.*, method 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 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.

Despite the similarities between the inheritance scheme in Eiffel and the composition scheme of stateful traits, there are some significant differences:

- *Renaming vs. aliasing* – In Eiffel, when a subclass is created, inherited features can be renamed. Renaming a feature has the same effect as (i) giving a new name to this feature and (ii) changing all the references to this feature. This implies a kind of mapping to be performed when a renamed method is accessed through the static type of the superclass.

  For instance, let's assume a class Component defines a method update. A subclass GraphicalComponent renames update into repaint, and redefines this repaint with a new implementation. The following code illustrates this situation:

```
class Component
feature
    update is
        do
            print ('1')
    end
end
```

```
class GraphicalComponent
inherit
    Component
        rename
            update as repaint
        redefine
            repaint
        end
    repaint is
        do
            print ('2')
        end
end
```

  In essence, the method repaint acts as an override of update. It means that if update is sent to an instance of GraphicalComponent, then repaint is called. This is illustrated in the following example:

```
f (c: Component) is
    do
        c.update
    end
f (create{GraphicalComponent})
==> 2
```

  This is the way Eiffel preserves polymorphism while supporting renaming. In stateful traits, aliasing a method or granting access to a variable assigns

a new name to it. The method or the variable can therefore still be invoked or accessed through its original name.
- *Merging variables* – In contrast to stateful traits, variables can be merged in Eiffel only if they come from a common superclass. In stateful traits, variables provided by two traits can be merged regardless of how these traits are formed.

Jigsaw. Jigsaw [Bra92] has a module system in which a module is a self-referential scope that binds names to values (*i.e.*, constant and functions). A module acts as a class (object generator) and as a coarse-grained structural software unit. Modules can be nested, therefore a module can define a set of classes. A set of operators is provided to compose modules. These operators are instantiation, merge, override, rename, restrict, and freeze.

Although there are some differences between the definition of a Jigsaw module and stateful traits, for instance with the rename operator, the more significant differences are in motivation and setting. Jigsaw is a framework for defining modular languages. Jigsaw supports full renaming, and assigns a semantic interpretation to nesting. In Jigsaw, a renaming is equivalent to a textual replacement of all occurrences of the attribute. The *rename* operator distributes over *override*. It means that Jigsaw has the following property:

(m1 rename a to b) override (m2 rename a to b) = (m1 override m2) rename a to b

Traits are intended to supplement existing languages by promoting reuse in the small, do not declare types, infer their requirements, and do not allow renaming. Stateless traits do not assign any meaning to nesting. Stateful traits are sensitive to nesting only to the extent that instance variables are private to a given scope. The Jigsaw operation set also aims for completeness, whereas in the design of traits we sacrifice completeness for simplicity.

A notable difference between Jigsaw and stateful traits is with the merging of variables. In Jigsaw, a module can have state, however variables cannot be shared between modules. With stateful traits the same variable can be accessed by the traits that *use* it (variables can be accessed by the classes that compose the traits). A Jigsaw module acts as a black-box. A module encapsulates its bindings and cannot be opened. While we value black-box composition, stateful traits do not take such a restrictive approach, but rather let the composer assume responsibility for the composition, while being protected from the impact of changes.

It is worth mentioning typing issues raised when implementing Jigsaw. Bracha [Bra92, Chapter 7] pointed out that the difficulty in implementing inheritance in Jigsaw (which is operator-based) stems from the interaction between structural subtyping and the algebraic properties of the inheritance operators (*e.g.*, *merge* and *override*).

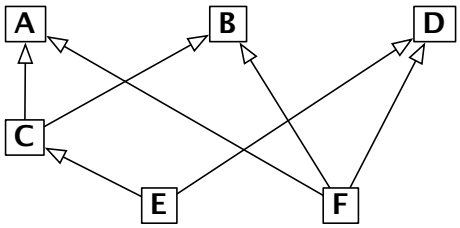

Fig. 26. E and F are structurally equivalent but may have different representations.

For example, let's consider the following classes A, B, C, D, E and F where C is a subclass of A and B. E is a subclass of D and C. F is a subclass of D, A and B. We have $C = AB$, $E = DC$ and $F = DAB$ where in $C_{new} = C_1C_2...C_n$ the superclasses of C_{new} are denoted C_i. (See Figure 26.) Expanding the definitions of all names (as dictated by structural typing), one finds that by associativity $E = F$. This equivalence dictates that all three classes have the same type, so that they can be used interchangeably. This in turn requires that all three have the same representation. However, using the techniques of C++ (Section 6.3), these three classes have different representations. This problem is avoided in traits where a trait does not define a type.

Cecil. Cecil [Cha92] is a purely object-oriented language that combines a classless object model, a kind of dynamic inheritance and an optional static type checking. Cecil's static type system distinguishes between subtyping and code inheritance even if the more common case is when the subtyping hierarchy parallels the inheritance hierarchy. Cecil supports multiple inheritance. Inheriting from the same ancestor more than once, whether directly or indirectly, has no effect other than to place the ancestor in relation to other ancestors: Cecil has no repeated inheritance. Inheritance in Cecil requires a child to accept all of the fields and methods defined in the parents. These fields and methods may be overridden in the child, but facilities such as excluding fields or methods from the parents or renaming them as part of the inheritance are not present in Cecil. This is an important difference with respect to stateful traits.

10 Conclusion

Stateless traits offer a simple compositional approach for structuring object-oriented programs. A trait is essentially a group of pure methods that serves as a building block for classes and as a primitive unit of code reuse. However this simple model suffers from several limitations, in particular (i) trait reusability is impacted because the required interface is typically cluttered with uninteresting required accessors, (ii) client classes are forced to implement boilerplate glue code, (iii) the introduction of new state in a trait propagates required ac-

cessors to all client classes, and (iv) public accessors break encapsulation of the client class.

We have proposed a way to make traits *stateful* as follows: First, traits can have private variables. Second, classes or traits composed from traits may use the *variable access operator* to (i) access variables of the used traits, (ii) attribute local names to those variables, and (iii) merge variables of multiple used traits, when this is desired. The flattening property is preserved by alpha-renaming variable names that clash.

Stateful traits offer numerous benefits: There is no unnecessary propagation of required methods, traits can encapsulate their internal representation, and the client can identify the essential required methods more clearly. Duplicated boilerplate glue code is no longer needed. A trait encapsulates its own state, therefore an evolving trait does not break its clients if its public interface remains unmodified.

Stateful traits represent a relatively modest extension to single-inheritance languages that enables the expression of classes as compositions of fine-grained, reusable software components. An open question for further study is whether trait composition can subsume class-based inheritance, leading to a programming language based on composition rather than inheritance as the primary mechanism for structuring code following Jigsaw design.

Acknowledgment

We also thank Gilad Bracha, Orla Greevy, Nathanel Schärli, Bernd Schoeller and Dave Thomas for their valuable discussions and comments. Thanks to Guillaume Marceau and Robby Findler for their help with the reduction semantics package for PLT Scheme. Thanks to Tom Van Cutsem for having found a typo in our formalization. Thanks to Ian Joyner for his help with the MacOSX Eiffel implementation.

References

[BBG+02] Lars Bak, Gilad Bracha, Steffen Grarup, Robert Griesemer, David Griswold, and Urs Hölzle. Mixins in Strongtalk. In *ECOOP '02 Workshop on Inheritance*, June 2002.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, volume 25, pages 303–311, October 1990.

[BDNW07] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Stateful traits. In *Advances in Smalltalk — Proceedings of 14th International Smalltalk Conference (ISC 2006)*, volume 4406 of *LNCS*, pages 66–90. Springer, 2007.

[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.

[CDG+92] Luca Cardelli, Jim Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 language definition. *ACM SIGPLAN Notices*, 27(8):15–42, August 1992.

[Cha92] Craig Chambers. Object-oriented multi-methods in Cecil. In O. Lehrmann Madsen, editor, *Proceedings ECOOP '92*, volume 615 of *LNCS*, pages 33–56, Utrecht, the Netherlands, June 1992. Springer-Verlag.

[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.

[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.

[Jik] The Jikes research virtual machine. http://jikesrvm.sourceforge.net/.

[Kro85] Stein Krogdahl. Multiple inheritance in Simula-like languages. In *BIT 25*, pages 318–326, 1985.

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

[Moh02] Markus Mohnen. Interfaces with default implementations in Java. In *Conference on the Principles and Practice of Programming in Java*, pages 35–40. ACM Press, Dublin, Ireland, jun 2002.

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

[Pyt] Python. http://www.python.org.

[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.

[SE90] Bjarne Stroustrup and Magaret A. Ellis. *The Annotated C++ Reference Manual*. Addison Wesley, 1990.

[SG99] Peter F. Sweeney and Joseph (Yossi) Gil. Space and time-efficient memory layout for multiple inheritance. In *Proceedings OOPSLA '99*, pages 256–275. ACM Press, 1999.

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

[Sny86] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 38–45, November 1986.

[UCCH91] David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hölzle. Organizing programs without classes. *LISP and SYMBOLIC COMPUTATION: An international journal*, 4(3), 1991.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 227–242, December 1987.