Adding State and Visibility Control to Traits using Lexical Nesting***

Tom Van Cutsem[1]***, Alexandre Bergel[2], Stéphane Ducasse[2], and Wolfgang De Meuter[1]

[1] Programming Technology Lab, Vrije Universiteit Brussel, Belgium
[2] RMoD research group, INRIA Lille, France

Abstract. Traits are reusable building blocks that can be composed to share methods across unrelated class hierarchies. Original traits are stateless and cannot express visibility control for methods. Two extensions, stateful traits and freezable traits, have been proposed to overcome these limitations. However, these extensions introduce complexity and have not yet been combined to simultaneously add both state *and* visibility control to traits.
This paper revisits the addition of state and visibility control to traits. Rather than extending the original traits model with additional operations, we allow traits to be lexically nested within other modules. Traits can then have (shared) state and visibility control by hiding variables or methods in their lexical scope. Although the Traits' "flattening property" has to be revisited, the combination of traits with lexical nesting results in a simple and expressive trait model. We discuss an implementation of the model in AmbientTalk and specify its operational semantics.

1 Introduction

Traits have been proposed as a mechanism to compose and share behavioral units between distinct class hierarchies. They are an alternative to multiple inheritance, the most significant difference being that name conflicts must be explicitly resolved by the trait composer. Traits are recognized for their potential in supporting better composition and reuse. They have been integrated into a significant number of languages, such as Perl 6, Slate [1], Squeak [2], DrScheme OO [3] and Fortress [4]. Although originally designed in a dynamically typed setting, several type systems have been built for Traits [5–8].

Several extensions of the original traits have been proposed to fix their limitations. Stateful traits present a solution to include state in traits [9]. In addition to defining methods, a trait may define state. This state is private by default and may be accessed within the composing entity. Freezable traits [10] provide a visibility control mechanism for methods defined in a trait: a method may either be (i) public and late bound or (ii) private and early bound. This enables the composer to change a trait method's visibility at composition time to deal with unanticipated name conflicts.

Although these extensions have been formally described and implementations were proposed, their main drawback is that the resulting language has too many operators

* Funded by the Interuniversity Attraction Poles Program, Belgian State, Belgian Science Policy
** Accepted to ECOOP 2009
*** Postdoctoral Fellow of the Research Foundation - Flanders (FWO)

and may introduce complex interactions with the host language. For example, freezable traits introduce early bound message dispatch semantics to support method privacy which may conflict with the host language's late bound semantics. Stateful traits introduce private state that may conflict with the host language's visibility rules. Finally, stateful traits must extend the host language's memory layout with a "copy down" technique [11] when linearizing variables whose offset in the memory layout is not constant among different composition locations.

This paper proposes a unique and simpler extension to traits to achieve both state and visibility control (to distinguish public from private state and behavior). We first revisit previous extensions to traits (as defined by among others this paper's second and third author). Subsequently, instead of providing two different visibility control mechanisms – one for state and another one for methods – we use lexical scoping as the sole visibility control mechanism for both state and methods.

Our approach is validated in AmbientTalk, a classless object-based language. In AmbientTalk, traits are plain, first-class objects that can be lexically nested (within other objects or methods). Traits can have private or public state. Unanticipated name conflicts can be reduced because a trait can make methods private by hiding them in its lexical scope. However, there is no mechanism to fully support unanticipated name conflicts, since the composer cannot rename or hide conflicting methods.

The contribution of this paper is a trait model that supports both state and visibility control without the introduction of any new composition operators, in contrast to stateful or freezable traits. Instead, our model relies on the introduction of one feature: lexical nesting. Our simpler model does require more support from its host language than the original one and is therefore not as straightforward to add to existing languages as is the original. Our contribution is validated as follows:

- we describe an existing implementation of our model in the AmbientTalk language.
- we demonstrate the effectiveness of our trait model by using it to structure a non-trivial Morphic-like UI framework.
- we provide an operational semantics to model lexical nesting of objects and the composition of lexically nested traits.

The paper is organized as follows. We first give a brief review of traits and point out some limitations related to state and visibility (Section 2). We then show how lexical nesting may be combined with traits (Section 3). To illustrate the implication of this combination in practice, we discuss a small case study (Section 4). We then formalize our approach by giving an operational semantics (Section 5), the properties of which are subsequently discussed (Section 6). A related work section (Section 7) and a conclusion end this paper (Section 8).

2 Traits and their Limitations

This section provides a brief description of the original Traits model. Readers already familiar with Traits may safely skip Section 2.1 and jump directly to Section 2.2.

2.1 Traits in a nutshell

An exhaustive description of Traits may be found in previous work [12]. This section highlights the most relevant aspects of Traits for the purpose of this paper.

Reusable groups of methods. A trait is a set of methods that serves as the behavioral building block of classes and is a primitive unit of code reuse. Traits cannot define state but may manipulate state via accessor methods.

Explicit composition. A class is built by reference to its superclass, uses a set of traits, defines state (variables) and behavior (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 (*i.e.,* class or trait) using a trait take precedence over trait methods. This allows methods defined in a composer to override methods with the same name provided by used traits; we call these methods *glue methods*.
- Traits may be flattened. In any class composer the traits can be in-lined to yield an equivalent class definition that does not use traits. This helps to understand classes that are composed of a large number of traits.
- Composition order is irrelevant. All the traits have the same precedence, and hence conflicting trait methods must be explicitly disambiguated by the composer.

Conflict resolution. While composing traits, method conflicts may arise. A *conflict* arises if two or more traits are combined that provide identically named methods not originating from the same trait. The composer can resolve a conflict in two ways: by defining a (glue) method that *overrides* the conflicting methods, or by *excluding* a method from all but one trait. Traits allow method *aliasing* 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.

2.2 Issues with freezable traits and stateful traits

Since the original paper *Traits: Composable Units of Behavior* was published at ECOOP 2003 [13], several communities expressed their interest in Traits. In spite of its acceptance, the trait model suffers from several drawbacks.

State. First, state is not modeled: a trait is made of a set of method definitions, required method declarations and a composition expression. At that time, allowing traits to define variables was not considered and was intentionally left as future work. The follow-up work, stateful traits [9], addressed this very issue. It introduces state and a visibility mechanism for variable privacy. Trait variables are private. Variable sharing is obtained with a new composition operation (called @@). Variables defined in different traits may be merged when those traits are composed with each other. This model raises several questions regarding encapsulation since a trait may be proposed as a black box and the composer has means to “open” it up. Since a linear object memory layout is

employed in most efficient object-oriented language implementation, a trait variable may have a different offset in the object layout at different composer locations. To be able to keep using a linear object layout, a technique known as "copy down" has to be employed [11].

Visibility. Second, no visibility control was proposed in the original version of traits. Reppy and Turon [14] proposed a visibility mechanism *à la* Java with public and private keywords. These access modifiers determine the visibility the members will have after they are inlined into a class but cannot be changed by the trait composer. Although having these visibilities for classes seems to be widely accepted, having the very same visibilities for traits seems not appropriate since the range of composition operators is far greater than the one for classes. Van Limberghen and Mens [15] showed that adding public/private/protected properties to attributes in mixins does not offer an adequate encapsulation mechanism. Traits can be composed in more ways than classes or mixins.

Freezable Traits [10] introduced visibility control to fully support unanticipated name conflicts. The composer may change the visibility of trait methods at composition time using two new operators freeze and defrost: the first operator turns a public late-bound method into a private early-bound method while the second reverts a private method into a public one. The problem of Freezable Traits is that it is complex to grasp the consequence of a change. More importantly Freezable Traits are based on the use of early-bound method dispatch. Such a mechanism may not be available in the host language implementation and adding it may radically change the resulting language.

Analysis. The two extensions to traits described above were designed in separation. Combining them into a unique language leads to a complex situation where two different visibility mechanisms coexist. Although doable, this would significantly raise the complexity of the trait model since 3 new operators (@@, freeze, defrost) and two visibility mechanisms would need to be added, which clearly goes against the simplicity of the original model.

In the following Section, we extend traits with both state and visibility control solely by combining them with lexical nesting of objects in the host language. Our model does not introduce any additional composition operators with respect to the original model.

3 Lexically Nested Object-based Traits

In this Section, we discuss how state and visibility control can be added to traits. Our first change to the model is that we no longer represent traits as distinct, declarative program entities, but rather as plain runtime objects (as in Self [16]). More specifically, traits will be represented as objects that close over their lexical environment *i.e.,* as closures. In languages that support closures, such as Scheme, function bodies close over their defining lexical environment. By doing so, the lexical environment may outlive the activation record in which it was created. Thus, a lexically free variable will retain its value and is said to have an indefinite extent. This property enables closures to hide (mutable) variables and auxiliary functions in their lexically enclosing scope.

In the following section we discuss how lexical scoping can and has been reconciled with object-oriented programming. We then introduce lexically nested traits in Section 3.2. Section 3.3 discusses how to compose such traits.

3.1 Objects as Closures

It has long been known that closures can be used to implement objects [17–19]. While closures successfully capture the essence of OOP, the expression of objects as functions and message passing as function calls models objects only as a "second class" abstraction. There do exist a number of object-oriented languages that have introduced first-class support for objects and message passing without losing the benefits of representing objects as closures. Examples of such languages include Emerald [20], ECMAScript (a.k.a. Javascript) [21], Beta [22], E [23] and Newspeak [24].

We introduce traits as closures in the AmbientTalk language [25, 26], an object-based distributed programming language that is closest in style to Miller's E language.

Objects as closures in AmbientTalk. AmbientTalk is a classless, object-based language. Listing 1.1 defines a simple counter abstraction with operations to access, increment and decrement its value. In languages that support (either first or second class) objects as closures, the role of classes as object generators is often replaced by functions that return a new object whenever they are called (cf. E [23], Emerald [20], Scheme [18]). We will name such functions, like the `makeCounter` function defined above, "constructor functions". To create a new counter, one calls the `makeCounter` constructor function.

```
def makeCounter(val) {
  object: {
    def count() { val };
    def inc() { val := val + 1 };
    def dec() { val := val - 1 };
  }
}
```

Listing 1.1. Constructor function for counter objects.

AmbientTalk fully exploits lexical scoping and allows object definitions to be nested within other object definitions or within the scope of a function or method body. In the above listing, the object construction expression `object: {...}` is lexically nested within the function `makeCounter`. The object expression groups the methods and fields of an object and evaluates to a new, independent object. Within the expression one can refer to lexically free variables, such as `val`. Objects close over their lexical environment, such that these variables have an indefinite extent. This allows the counter to keep track of its state using `val`, while keeping this variable completely hidden within its lexical scope. Executing `makeCounter(0).val` will raise an exception.

This simple object model removes the need for special language constructs for object creation (replaced by calling a constructor function), visibility control (hiding

names using lexical scoping), special constructor methods (replaced by constructor functions), static fields or methods (which are free variables of the constructor function) and singletons (by not nesting an object definition within a constructor function).

Lexical nesting and inheritance. AmbientTalk, like Self [27], supports *object-based inheritance* which is a relationship between two objects, a *delegator* and a *delegate*, rather than between a class and a superclass. Object-based inheritance implies that if the delegator receives a message it does not understand, it will delegate this message to its delegate object. Delegating a message to an object is different from sending a message to an object. *Delegation*, as first proposed by Henry Lieberman [28], implies that if a matching method is found in the delegate, in the subsequent method invocation the `self` pseudovariable will refer to the delegator (that is: the object that originally received the message). This property ensures that object-based inheritance properly supports method overriding.

In AmbientTalk, every object has a field named `super` which refers to an object's delegate. There is only one such field in each object; multiple inheritance is not supported. Objects by default delegate to an object named `nil`. Listing 1.2 exemplifies object-based inheritance. It shows the abstraction of a counter that cannot be decremented below zero. The delegate of such a "positive" counter object is a regular counter object. Any message for which a positive counter does not implement a method will be delegated to a regular counter object. The positive counter overrides the `dec` method and, if the counter has not reached zero yet, explicitly delegates the `dec` message to its delegate by means of the syntax `super^dec()`.

```
def isStrictlyPositive(x) { x > 0 };

def makePositiveCounter(val) {
  extend: makeCounter(val) with: {
    def dec() {
      if: isStrictlyPositive(self.count()) then: {
        super^dec()
      }
    }
  }
}
```

Listing 1.2. Object-based inheritance.

Because AmbientTalk allows both lexical nesting and (object-based) inheritance, we must clarify the semantics of looking up identifiers, as there are now two hierarchies of names: the lexical scope (starting with the object itself, ending in the top-level scope via lexical nesting) and the object's inheritance chain (starting with `self`, ending in `nil` via each object's `super` field). An unqualified identifier, such as `val`, is *always* looked up in the object's lexical scope. A qualified identifier, such as `inc` in `c.inc()` or `count` in `self.count()`, is looked up in the receiver's inheritance chain. The major difference with mainstream object-oriented languages is that `m()` is *not* equivalent to `self.m()`. This is

similar to method lookup in Newspeak [24], except that in Newspeak the inheritance chain is still considered if the method is not lexically visible.

The example given above shows how the positive counter abstraction can unambiguously make use of both lexically visible as well as inherited methods. The call to `isStrictlyPositive` is guaranteed to refer to the lexically visible definition. The invocation `self.count()` will find the method defined in the delegate object.

3.2 Lexically Nested Traits

AmbientTalk supports trait-based composition between objects. An object can import zero or more traits when it is defined. This causes the object to acquire all of the methods defined by its imported traits, as if it had defined those methods itself. If an imported method name clashes with another imported name, or with a name explicitly defined by the importing object, an exception is raised when the object is created to signal this conflict. Conflicts should be dealt with by the programmer by either aliasing an imported method name or by excluding it.

In AmbientTalk, traits are regular objects rather than distinct runtime values or static program declarations. Any object can play the role of a trait. Hence, like all objects, traits can make use of lexical nesting to hide private state or auxiliary functions. To the best of our knowledge, this combination hasn't been achieved before: it is a novel property that is not available in other languages with explicit support for trait composition.

Listing 1.3 defines a trait that provides a reusable abstraction for animating arbitrary objects. The example is taken from the implementation of a Morphic-like graphical kernel for AmbientTalk which is discussed in more detail in Section 4. The animation trait is parameterized with the refresh rate between animation calls (in milliseconds). It provides two methods, `start` and `stop`, to start and stop the animation loop. The `start` method triggers the animation loop by sending the message `every:do:` to its `timer`, passing as the second argument an anonymous zero-argument closure to be executed every `refreshRate` milliseconds. Note that the `timer` variable is hidden within the lexical environment of the animation trait object. As such, the variable is private to the trait and will not be visible to the composing clients. This example illustrates that traits can be stateful in AmbientTalk.

```
def makeAnimationTrait(refreshRate) {
  def timer := makeTimer();
  object: {
    def start() { timer.every: refreshRate do: { self.animate() } };
    def stop() { timer.reset() };
  }
}
```

Listing 1.3. A trait as a regular object.

To actually perform the animation, the animation trait requires the composite (*i.e.,* the object using the trait) to define a method named `animate`. The set of methods required

by a trait is implicit in the source code. A method is required by a trait if the trait does not implement the method, yet it is invoked in the code (e.g. by means of a self-send).

Listing 1.4 shows the implementation of a particle morph that uses the above animation trait to move within a given direction at a constant rate (see Section 4 for a more in-depth explanation of a "morph"). We assume that `makeCircleMorph` creates an object that is graphically represented as a circle. The particle morph implements the `animate` method as required by the animation trait. At each step of the animation, the particle morph moves itself at the given `moveRate`.

```
def makeParticleMorph(radius, moveRate, dx, dy) {
  extend: makeCircleMorph(radius) with: {
    import makeAnimationTrait(moveRate);
    def animate() {
      self.move(dx, dy);
    };
  }
}
```

Listing 1.4. Composing an object with a trait.

Composition of traits is performed by means of the **`import`** statement. Because the animation trait is stateful, the particle morph first generates a new instance of this trait (by invoking the `makeAnimationTrait` constructor function) and then imports this new instance. The operational effect of the **`import`** statement is that the particle morph acquires its own, local definitions for the methods `start` and `stop`. How exactly this acquisition of methods takes place is the topic of the following section. For now, it suffices to understand that when `start` is sent to a particle morph, the implementation of the animation trait is invoked and a **`self`**`.animate()` call will invoke the particle morph's `animate` method, as expected.

Conflict resolution. When the composite object is created, the **`import`** statement raises an exception if the composite and the trait define slots with the same name. It is up to the composite to explicitly resolve name conflicts between imported traits, or between an imported trait and itself. The composite can do so by aliasing or excluding imported methods[3]. For example, the particle morph can import the animation trait as follows:

```
import makeAnimationTrait(moveRate) alias start := startMoving exclude stop;
```

In this case, the particle morph will acquire a method named `startMoving` rather than `start`. Because the `stop` method is excluded, the particle morph will not acquire this method such that it cannot be stopped by client code.

[3] Note that aliasing solves name conflicts, but does not guarantee that the intentional behavior of the composed traits is preserved. This issue has previously been addressed [10] and is not further discussed here.

Initialization. As discussed previously, AmbientTalk objects are constructed by calling ordinary functions. All code executed when calling such constructor functions is regarded as initialization code. When objects are used as traits (*i.e.,* constructed as part of an `import` statement), their initialization code is ran in the order in which the `import` statements occur in the code. If more control over the composition of initialization code is required, such code can be transferred to a dedicated trait method that can then be composed, aliased or excluded at will by the composing object.

3.3 Flattening Lexically Nested Traits

We now discuss how exactly a composite object acquires the method definitions of its imported traits. In the original version of the traits model, trait composition enjoys the so-called *flattening property*, which states that the semantics of a class defined using traits is exactly the same as that of a class constructed directly from all of the non-overridden methods of the traits [13]. The intuitive explanation is that trait composition can be understood in terms of copy-pasting the method definitions of the trait into the class definition.

When traits can be lexically nested, trait composition no longer adheres to the flattening property. The reason is that each imported trait method has its own distinct lexical environment upon which it may depend. If the method body of an imported trait method were copy-pasted verbatim into the composing object, lexically free variables of the method may become unbound or accidentally rebound to a variable with the same name in the lexical environment of the composing object. An imported trait method should retain its own lexical environment, which implies that it is not equivalent to a method defined by the composing object with the same method body.

In this Section, we explore an alternative semantics for trait composition based on delegation to avoid the obvious problems related to the flattening property when traits are closures. More specifically, we make use of a language feature of AmbientTalk that has not been explained thus far, which is the explicit delegation of messages between objects. In what follows, we first discuss this language feature. Subsequently, we apply it to implement trait composition.

Explicit delegation As discussed in Section 3.1, an AmbientTalk object delegates any message it does not understand to the delegate object stored in its `super` field. This mechanism is known as object-based inheritance. In this case, delegation of the message happens *implicitly*. However, AmbientTalk also provides a delegation operator `^` that allows objects to *explicitly* delegate messages to objects other than the object stored in their `super` field. This enables objects to reuse code from different objects without resorting to multiple inheritance. Listing 1.5 exemplifies such reuse by extracting the reusable behavior of enumerable collections into a separate object (modeled after Ruby's `Enumerable` mixin module [4]). All reusable methods depend on a required method named `each:`. A collection representing an interval of integers $[min, max[$ reuses this behavior by providing an implementation for `each:`.

[4] http://www.ruby-doc.org/core/classes/Enumerable.html

```
def Enumerable := object: {
    def collect: function {
        def sequence := makeSequence();
        self.each: { |elt| sequence.append(function(elt)) };
        sequence
    };
    def detect: predicate {...};
    ...
}
def makeInterval(min, max) {
    extend: Collection with: {
        // delegate messages to Enumerable to acquire its behavior
        def collect: function { Enumerable^collect: function };
        def detect: predicate { Enumerable^detect: predicate };
        ...
        def each: function { // the method needed by Enumerable
            // apply function to all Integer objects between min and max
            min.to: max do: function
        };
    }
}
```

Listing 1.5. Composition of a reusable object via explicit delegation.

In order to reuse the functionality provided by `Enumerable`, the interval object defines a number of *delegating methods*. The sole purpose of such methods is to explicitly delegate a message to another object. The expression `obj^m()` denotes the delegation of a message `m` to an object `obj`. Recall from Section 3.1 that the difference between a delegation and a regular message send (*i.e.,* `obj.m()`) is that the former leaves the **`self`** pseudovariable unchanged during the invocation of `m()` (*i.e.,* **`self`** is not rebound to refer to `obj`). This property is crucial to enable the kind of reuse exemplified above: the implementation of `Enumerable`'s methods is only partial. It depends on a method call to `each:` (shown underlined) that should generate a sequence of the collection's elements by feeding them to a single-argument function. This method is therefore implemented by the interval object. Because the interval uses explicit delegation to forward the `collect:` and `detect:` messages to `Enumerable`, any occurrence of **`self`**`.each:` in these methods will refer to the implementation provided by the interval object.

The above example shows that explicit delegation allows objects to reuse partially implemented methods via object composition rather than via (object-based) inheritance. The advantage of composition over (multiple) inheritance is that it enables the reuse of methods from multiple objects without introducing ambiguity. Its disadvantage is that the composing object needs to define explicit delegating methods for each method it wants to reuse as part of its interface. Below, we discuss how the definition of these delegating methods can be automated, by defining trait composition in terms of generating delegating methods.

Trait composition by delegating method generation Even though traits cannot be flattened in a language that models traits as closures (as discussed previously), we can attribute a simple semantics to trait composition in terms of explicit delegation. To acquire a method defined in an imported trait object, the composite object can generate a delegating method for it. This has the following desirable properties:

– Because the composite explicitly delegates to the trait object, the trait method is invoked in its proper lexical environment. The lexical environment of the trait's methods is unaffected by the trait composition.
– Because delegation does not alter the binding of **`self`**, this pseudovariable can be used by the trait to invoke its required methods, implemented by the composite.

Given this semantics, if we regard the `Enumerable` object as a trait, the interval object's definitions for the delegating methods can be replaced by a single **`import`** `Enumerable` statement to achieve the same operational effect. Listing 1.6 shows the definition of the particle morph from Section 3.2 where the trait import has been transformed into a set of delegating methods. In this example, we assume `t_` to be a fresh variable name. Note that the animation trait's methods retain their proper lexical environment. Furthermore, this semantics respects the ability of nested traits to close over their lexical environment to encapsulate state and private behavior. For example, the particle morph cannot access the `timer` variable associated with its imported trait.

```
def makeParticleMorph(radius, moveRate, dx, dy) {
  def t_ := makeAnimationTrait(moveRate);
  extend: makeCircleMorph(radius) with: {
    // import is transformed into a set of delegating method definitions
    def start() { t_^start() };
    def stop() { t_^stop() };
    def animate() {
      self.move(dx, dy);
    };
  }
}
```

Listing 1.6. Trait composition is transformed into delegation.

Given the semantics of trait composition by means of explicit delegation, aliasing a method by means of **`alias`** `oldName := newName` generates the delegating method **`def`** `newName() { t_^oldName() }` and the semantics of **`exclude`** `name` is simply that no delegating method for `name` is defined in the importing scope.

Dealing with state. The example objects that we have shown up to now have private state because the fields holding this state are hidden in their lexical environment. It is also possible for objects to declare fields directly as part of their public interface. Since traits are ordinary objects, they may also declare fields in addition to methods. As previously described, when a trait is imported a delegate method is defined for each of its provided methods. In addition, for each field provided by the trait, a field with the same name is defined in the object that imports the trait. Each object that imports a trait with public state will thus have its own copy of that state. As is the case with methods, an exception is raised if the names of imported fields conflict with those defined in the importing object.

3.4 Summary

Objects can be augmented with private state and visibility control by allowing them to close over their environment of definition. We added these properties to traits by similarly representing them as plain objects that close over their environment of definition. However, when representing traits in this way, the traditional way of composing traits by flattening them must be reconsidered. If we were to copy the method bodies of a trait's provided methods directly in the importing scope, their lexical scope would be ill-defined. One way to reconcile trait composition with lexical nesting is by expressing the composition in terms of *delegating methods* that delegate a message to a trait. The composite acquires the delegating method, rather than the method's implementation. The use of delegation allows the trait to execute the method in its proper lexical scope yet access any required methods provided by the composite via self-sends.

Contrary to Stateful and Freezable traits, our model does not introduce any new trait composition operators, thus preserving the simplicity of the original model. The drawback is that our model cannot express certain compositions that can be expressed using Stateful or Freezable traits. For example, contrary to Freezable Traits we provide no operators to deal with unanticipated name conflicts since the composer cannot change the visibility of a trait's provided fields or methods.

4 Case Study: AmbientMorphic

We demonstrate the applicability of lexically nested traits by means of a concrete case study. AmbientMorphic is a minimal implementation of the Morphic user-interface construction framework [29] in AmbientTalk[5]. In Morphic, the basic unit of abstraction is called a *morph*. A morph is an object with a graphical manifestation on the screen. Morphs can be composed into typical user interface widgets, but they can equally be used for rendering more lively applications such as e.g. a simulation of atoms in a gas tank [30]. The goal of the morphic framework is to create the illusion that the graphical "objects" which can be seen on the screen really *are* the objects that the programmer manipulates in the code.

Morphic is an ideal case study for traits because morphs can be decomposed into many different yet interdependent concerns. Typical concerns include drawing and redrawing, resizing, keeping track of which morph has the current focus, determining what morph is currently under the cursor (which is represented by the "hand morph" in Morphic), etc. In our framework, a `Morph` is composed of many small traits that each encode such a concern. Figure 1 depicts a subset of the framework. The entire framework totals 18 traits (13 of which are stateful) and 12 morphs.

Decomposing a morph into separate traits leads to separation of concerns (*i.e.,* increased modularity) and also enables programmers to reuse traits to build other kinds of morphs (*i.e.,* increased reuse). Because our trait model additionally enables state and visibility control, we gain the following benefits:

[5] The framework is included in the open-source AmbientTalk distribution available online at http://prog.vub.ac.be/amop.

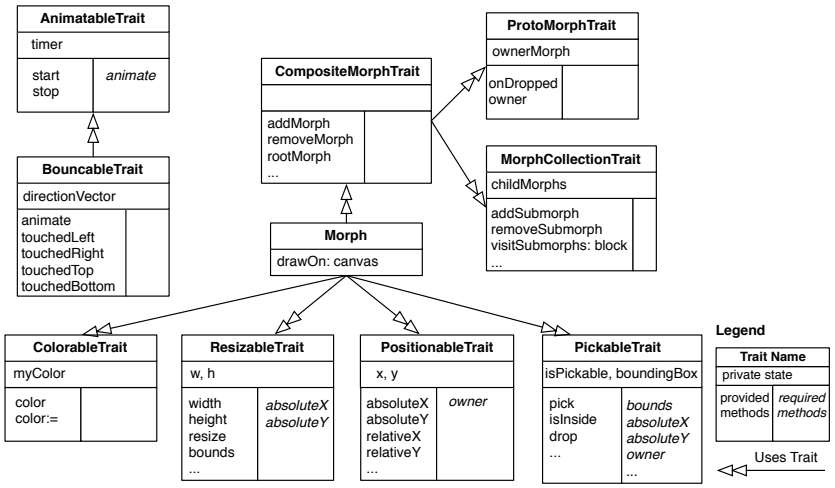

Fig. 1. A selection of the traits used in the AmbientMorphic framework.

1. Since traits can be stateful, the `Morph` object does not need to be polluted with the fields, accessors and mutators that would otherwise be required by the traits.
2. State remains well encapsulated within each trait. If one of the traits modifies the representation of its state, this will not impact the `Morph`.

These benefits would not have been achieved using the original traits model. The first benefit can be achieved with stateful traits and the second with freezable traits. However, these extensions have not before been combined into a single model.

5 Operational Semantics

We now formalize the features presented in Section 3 by providing an operational semantics for three increasingly descriptive calculi: PROTOLITE, LEXLITE, and AMBIENTTALKLITE. PROTOLITE is a core calculus that is gradually extended with the features required for lexically nested traits. The goal of this formalization is to provide the necessary technical description required when one would want to reproduce our implementation of lexically nested traits. This section does not provide any hint on how to make this implementation fast (this is not the focus of this paper), but it conveys the necessary details to realize one.

PROTOLITE is a minimal language that captures the essence of a dynamically typed, object-based programming language. It features ex-nihilo object creation, message passing, field access and update and finally explicit delegation (as discussed in Section 3.3). We chose not to formalize AmbientTalk's support for implicit delegation via **`super`** fields because it is not essential to our discussion on trait composition.

While PROTOLITE allows object definitions to be syntactically nested within other object definitions, objects are not allowed to refer to lexically free fields or methods.

LEXLITE extends PROTOLITE with proper lexically nested objects. This enables nested objects to access lexically visible fields and methods of their enclosing objects, a key property for adding state and visibility control to traits. Finally, AMBIENTTALKLITE extends LEXLITE with support for trait composition.

Related work. A number of calculi that describe delegation or traits have been formulated so far. δ [31] is an imperative object based calculus with delegation. δ allows objects to change their behavior at runtime. Updates to an object's fields can be either lazy or eager. δ also introduces a form of explicit delegation. An object has a number of delegates (which may be dynamically added or removed) and a message may be explicitly directed at a delegate. We chose not to use δ as our starting point because dynamic addition and removal of delegates and methods is not required for our purpose and because it does not support lexical nesting.

Incomplete Featherweight Java [32] is an extension of Featherweight Java with incomplete objects *i.e.,* objects that require some missing methods which can be provided at run-time by composition with another complete object. The mechanism for method invocation is based on delegation and it is disciplined by static typing. The authors extend the class-based inheritance paradigm to make it more flexible.

Bono and Fisher [33] have designed an imperative object calculus to support class-based programming via a combination of extensible objects and encapsulation. Two pillars of their calculus are an explicit type hierarchy and an automatic propagation of changes. Their focus is thus rather different from ours.

5.1 PROTOLITE

Figure 2 presents the syntax of PROTOLITE. This syntax is reduced to the minimum set of syntactic constructions that models object creation, message sending, and message delegation. We use the meta-variable e for expressions, m for method names, *f* for field names and *x* for variable names.

$$
\begin{array}{rl}
e = & \textbf{object}\{\mathit{field}^* \;\; \mathit{meth}^*\} \\
& \mid e.f \mid e.f := e \mid e.m(e^*) \mid e\hat{}m(e^*) \mid \underline{e\langle oid\rangle\hat{}m(e^*)} \\
& \mid e\,;\,e \mid x \mid \textbf{self} \mid \mathsf{nil} \mid \underline{oid} \\
\mathit{meth} = & \; m(x^*) = e \\
\mathit{field} = & \; f := e
\end{array}
$$

Fig. 2. PROTOLITE expressions (run-time expressions are shown underlined)

Objects are created by means of the object creation expression **object**$\{\ldots\}$. An object consists of a set of fields and methods. We assume that all object expressions are valid, *i.e.,* field and method names are unique within an object creation expression. Each

field declaration is associated with an initialization expression, which is evaluated when the object is created. Messages may be sent to the result of an expression using an arbitrary number of arguments ($e.m(e^*)$). Messages may also be delegated ($e\hat{}m(e^*)$). The figure shows run-time expressions, which are underlined, following the notation of Flatt *et. al* [34]. These expressions cannot be formulated in source code. They exist at run-time and may contain annotations to store additional information about the expression (shown using $\langle\rangle$).

Figure 3 shows the evaluation contexts using the notation of Felleisen and Hieb [35]. Evaluation contexts specify the evaluation order on expressions. For example, the receiver expression of a message send must be evaluated before its arguments. The notation $v^*\ E\ e^*$ indicates the evaluation of expressions from left to right. The names v, o and o' are meta-variables that designate object references.

$$
\begin{aligned}
E &= [\,] \mid E.f \mid E.f := e \mid E.m(e^*) \mid E\langle o\rangle\hat{}m(e^*) \\
&\quad \mid o.f := E \mid o.m(v^*\ E\ e^*) \mid o\langle o'\rangle\hat{}m(v^*\ E\ e^*) \mid E\ ;\ e \\
v, o, o' &= \mathsf{nil} \mid \mathit{oid}
\end{aligned}
$$

Fig. 3. PROTOLITE Evaluation contexts

Figure 4 describes the generation of PROTOLITE run-time expressions. Such expressions are generated to annotate objects and replace **self**. This annotation and replacement occurs before evaluating a field initialization expression contained in an object definition and before evaluating a method body.

$$
\begin{aligned}
[\![x]\!]_o &= x & [\![e.f]\!]_o &= [\![e]\!]_o.f \\
[\![\mathbf{self}]\!]_o &= o & [\![e.f := e']\!]_o &= [\![e]\!]_o.f := [\![e']\!]_o \\
[\![\mathsf{nil}]\!]_o &= \mathsf{nil} & [\![e.m(e_i^*)]\!]_o &= [\![e]\!]_o.m([\![e_i]\!]_o^*) \\
[\![e\ ;\ e']\!]_o &= [\![e]\!]_o; [\![e']\!]_o & [\![e\hat{}m(e_i^*)]\!]_o &= [\![e]\!]_o\langle o\rangle\hat{}m([\![e_i]\!]_o^*) \\
& & [\![\mathbf{object}\{\mathit{field}^*\ \ \mathit{meth}^*\}]\!]_o &= \mathbf{object}\{\mathit{field}^*\ \ \mathit{meth}^*\}
\end{aligned}
$$

Fig. 4. Annotating PROTOLITE expressions

Message sending and delegation require a mechanism to bind the **self** pseudo-variable to an arbitrary value (the receiver). The $[\![e]\!]_o$ operator recursively replaces all references to **self** by o in the expression e. References to **self** within a nested object creation expression in e are not substituted because **self** references within this expression should refer to the nested object itself. Delegated message sends need to keep a

reference to the object that performs the delegation. To this end, they are replaced by a run-time expression that is annotated with the value of **self**. This reference is used to replace the **self** variable when the delegated send is executed.

Figure 5 shows the reduction rules of PROTOLITE. Each of these is an elementary reduction, mapping an evaluation context E and a store $\mathcal{S}$ onto a reduced evaluation context and an updated store. Objects are represented as tuples $\langle \mathcal{F}, \mathcal{M} \rangle$ of a map of fields $\mathcal{F}$ and methods $\mathcal{M}$. In the [*object*] reduction rule, fields are initialized to nil when the object is created. Their initial values are subsequently assigned by evaluating the initialization expressions before the object is returned. References to **self** within a field initialization expression are first substituted for the new object.

In the [*delegate*] reduction rule, note that o' (the delegator) rather than o (the receiver) substitutes **self** in the invoked method. In the [*send*] and [*delegate*] rules, before evaluating a method body $e[v^*/x^*]$ substitutes parameters x^* for the argument values v^*. In the method body, a field f of the enclosing object is accessible via **self**.f.

$$\langle E[\textbf{object}\{\textit{field}^*\ \textit{meth}^*\}], \mathcal{S}\rangle \hookrightarrow \langle E[(o.f := [\![e]\!]_o)^*; o], \mathcal{S}[o \mapsto \langle \mathcal{F}, \mathcal{M}\rangle]\rangle \quad [\textit{object}]$$
where $o \notin \mathrm{dom}(\mathcal{S})$ and $\mathcal{F} = \{ f \mapsto \mathsf{nil} \mid \forall f := e \in \textit{field}^* \}$
and $\mathcal{M} = \{ m \mapsto \langle x^*, e' \rangle \mid \forall m(x^*) = e' \in \textit{meth}^* \}$

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

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

$$\langle E[o.m(v^*)], \mathcal{S}\rangle \hookrightarrow \langle E[[\![e[v^*/x^*]]\!]_o], \mathcal{S}\rangle \quad [\textit{send}]$$
where $\mathcal{S}(o) = \langle \mathcal{F}, \mathcal{M}\rangle$ and $m \mapsto \langle x^*, e\rangle \in \mathcal{M}$

$$\langle E[o\langle o'\rangle\hat{}m(v^*)], \mathcal{S}\rangle \hookrightarrow \langle E[[\![e[v^*/x^*]]\!]_{o'}], \mathcal{S}\rangle \quad [\textit{delegate}]$$
where $\mathcal{S}(o) = \langle \mathcal{F}, \mathcal{M}\rangle$ and $m \mapsto \langle x^*, e\rangle \in \mathcal{M}$

$$\langle E[o\ ;\ e], \mathcal{S}\rangle \hookrightarrow \langle E[e], \mathcal{S}\rangle \quad [\textit{seq}]$$

Fig. 5. Reductions for PROTOLITE

5.2 LEXLITE

PROTOLITE does not allow objects to access fields and methods of their enclosing objects. LEXLITE extends PROTOLITE with a new syntax and semantics that allows objects to access lexically visible fields and methods. Figure 6 shows the syntax extensions of LEXLITE with respect to PROTOLITE. The new expressions denote the invocation of a lexically visible method m and the access to a lexically visible field f. LEXLITE supports an additional run-time expression that annotates an object creation expression

$$e = \ldots \mid m(e^*) \mid f \mid \underline{\textbf{object}\{\textit{field}^*\ \textit{meth}^*\}\langle L\rangle}$$

Fig. 6. LEXLITE syntax extensions to PROTOLITE

with a lexical environment L. This annotation is generated when closing over the lexical environment. This is explained in more detail below.

In LEXLITE, receiverless (*i.e.,* lexically resolved) method invocation and field access are interpreted as if the method or field was invoked on the lexically visible object in which the method or field is defined. Also, object definitions must now close over their lexical environment, such that expressions contained in their methods may correctly refer to methods and fields defined in an enclosing lexical environment. We represent the lexical environment as a function $L(n) = o$ mapping a method or field name n to the object o in which that name is defined. Figure 7 shows how each expression e closes over a lexical environment L by means of the transformation $\mathcal{C}_L[\![e]\!]$. Following the convention previously introduced, code generated by this transformation is underlined.

$$\begin{array}{rl}
\mathcal{C}_L[\![x]\!] = x & \mathcal{C}_L[\![e.f := e']\!] = \mathcal{C}_L[\![e]\!].f := \mathcal{C}_L[\![e']\!] \\
\mathcal{C}_L[\![\textbf{self}]\!] = \textbf{self} & \mathcal{C}_L[\![e.m(e_i^*)]\!] = \mathcal{C}_L[\![e]\!].m(\mathcal{C}_L[\![e_i]\!]^*) \\
\mathcal{C}_L[\![\mathsf{nil}]\!] = \mathsf{nil} & \mathcal{C}_L[\![e\hat{}m(e_i^*)]\!] = \mathcal{C}_L[\![e]\!]\hat{}m(\mathcal{C}_L[\![e_i]\!]^*) \\
\mathcal{C}_L[\![e\ ;\ e']\!] = \mathcal{C}_L[\![e]\!]\ ;\ \mathcal{C}_L[\![e']\!] & \mathcal{C}_L[\![f]\!] = \underline{l.f} \text{ where } l = L(f) \\
\mathcal{C}_L[\![e.f]\!] = \mathcal{C}_L[\![e]\!].f & \mathcal{C}_L[\![m(e_i^*)]\!] = \underline{l.m}(\mathcal{C}_L[\![e_i]\!]^*) \text{ where } l = L(m) \\
 & \mathcal{C}_L[\![\textbf{object}\{\textit{field}^*\ \ \textit{meth}^*\}]\!] = \textbf{object}\{\textit{field}^*\ \ \textit{meth}^*\}\underline{\langle L\rangle}
\end{array}$$

Fig. 7. LEXLITE expressions closing over a lexical scope L

Because lexically scoped method invocations and field accesses are transformed into regular method invocations and field accesses when expressions close over their defining lexical environment, no special reduction semantics must be added for these expressions. However, the reduction semantics for [*object*] must be refined such that method bodies now close over the lexical environment in which the object has been defined. This new reduction rule is shown in Figure 8. The other reduction rules for LEXLITE are the same as those defined in Figure 5.

Note that a lexical closure is not defined as a "snapshot" of the lexical environment at the time the object is created. This would work for a functional language, but since LEXLITE is stateful, the closure must refer to the actual enclosing objects such that state changes in those objects remain visible to the nested object. Finally, note that by transforming a receiverless method invocation $m()$ into a receiverful method invocation $l.m()$ on the enclosing object l, within m the binding of **self** will correctly refer to the enclosing object (*i.e., l*) rather than to the nested object that performed the invocation.

$$\langle E[\textbf{object}\{\textit{field}^*\,\textit{meth}^*\}\langle L\rangle], \mathcal{S}\rangle \hookrightarrow \langle E[(o.f := \mathcal{C}_{L'}\ [\![[\![e]\!]_o]\!])^*; o], \mathcal{S}[o \mapsto \langle \mathcal{F}, \mathcal{M}\rangle]\rangle\ [\textit{object}]$$

where $o \notin \text{dom}(\mathcal{S})$ and $\mathcal{F} = \{ f \mapsto \mathsf{nil} \mid \forall f := e \in \textit{field}^* \}$

and $\mathcal{M} = \{ m \mapsto \langle x^*, \mathcal{C}_{L'}[\![e']\!]\rangle \mid \forall\, m(x^*) = e' \in \textit{meth}^* \}$

and $L'(n) = \begin{cases} o & \text{if } n \in \text{dom}(\mathcal{F}) \cup \text{dom}(\mathcal{M}) \\ L(n) & \text{otherwise} \end{cases}$

Fig. 8. Redefined reduction rule for LEXLITE

In LEXLITE, lexical lookup proceeds via a chain of L functions. In order to be well-defined, this lookup must eventually end. Therefore, the top-level expression of a LEXLITE program must close over an empty top-level environment $T(n) = \bot$ before it can be reduced. If one wants to bootstrap the lexical environment with top-level methods and fields, these can be encoded as follows. If e is the top-level expression encoding a program and m is a fresh name then the expression **object**$\{...m() = e...\}.m()$ introduces an explicit top-level environment. All fields and methods declared in this outer object expression can be regarded as top-level in e.

5.3 AMBIENTTALKLITE

AMBIENTTALKLITE extends LEXLITE with explicit support for trait composition. It exhibits the properties of AmbientTalk regarding state and visibility control (cf. Section 3.2) and describes trait composition in terms of generating delegate methods (cf. Section 3.3). Figure 9 shows the syntax extensions of AMBIENTTALKLITE with respect to LEXLITE. An object creation expression may contain **import** declarations to acquire the fields and methods of trait objects. The expression e in the **import** declaration is eagerly reduced to the trait object. A run-time **import** declaration is introduced which is annotated with a mapping $\mathcal{A}$ that maps names to their aliases and a set $\mathcal{E}$ of field or method names to be excluded. Figure 10 shows how these annotations are generated based on the **alias** and **exclude** clauses of the original **import** declaration.

$$\begin{aligned} e &= \ldots \mid \textbf{object}\{\textit{field}^*\ \ \textit{meth}^*\ \ \textit{import}^*\} \\ \textit{import} &= \textbf{import}\ e\ \textbf{alias}\ \textit{alias}^*\ \textbf{exclude}\ n^* \\ &\mid \underline{\textbf{import}\ e\ \langle \mathcal{A}, \mathcal{E}\rangle} \\ \textit{alias} &= n \leftarrow n \\ n &= \text{a method or field name} \end{aligned}$$

$$E = \ldots \mid \textbf{object}\{\textit{field}^*\ \textit{meth}^*\ \textbf{import}\ v\ \langle \mathcal{A}, \mathcal{E}\rangle^*\ \ \textbf{import}\ E\ \langle \mathcal{A}, \mathcal{E}\rangle\ \ \textbf{import}\ e\ \langle \mathcal{A}, \mathcal{E}\rangle^*\}$$

Fig. 9. AMBIENTTALKLITE syntax and evaluation context extensions to LEXLITE

$$[\![\textbf{object}\{\textit{field}^*\ \ \textit{meth}^*\ \ \textit{import}^*\}]\!]_o = \textbf{object}\{\textit{field}^*\ \ \textit{meth}^*\ \ [\![\textit{import}^*]\!]_o\}$$
$$[\![\textbf{import}\ e\ \textbf{alias}\ \textit{alias}^*\ \textbf{exclude}\ n^*]\!]_o = \textbf{import}\ [\![e]\!]_o\langle\mathcal{A},\mathcal{E}\rangle$$
$$\text{where } \mathcal{A}(n) = \begin{cases} n' \text{ if } n' \leftarrow n \in \textit{alias}^* \\ n \ \text{ otherwise} \end{cases}$$
$$\text{and } \mathcal{E} = \{n \mid n \in n^*\}$$

Fig. 10. Annotating AMBIENTTALKLITE expressions

The [*import*] reduction rule in Figure 11 shows how the **import** declarations are expanded into a set of generated fields and delegating methods. Field definitions present in the value being imported, t_i, are copied (as explained in Section 3.3). For each imported method m, a delegating method n is generated which delegates m to t_i. Note that the use of $\mathcal{A}_i$ ensures a field or method renaming if specified. The last two lines indicate the constraint that duplicate field or method names are disallowed. Once the **import** declarations are reduced, a regular LEXLITE object creation expression remains.

$$\langle E[\textbf{object}\{\textit{field}^*\ \ \textit{meth}^*\ \ \textbf{import}\ t_i\langle\mathcal{A}_i,\mathcal{E}_i\rangle^*\}\langle L\rangle], \mathcal{S}\rangle \qquad [\textit{import}]$$
$$\hookrightarrow \langle E[\textbf{object}\{\textit{field}^*\ \ \underline{\textit{ifield}^*}\ \ \textit{meth}^*\ \ \underline{\textit{imeth}^*}\}\langle L\rangle], \mathcal{S}\rangle$$
$$\text{where } \mathcal{S}(t_i) = \langle\mathcal{F}_i, \mathcal{M}_i\rangle$$
$$\text{and } \textit{ifields}_i = \{\ \underline{n := v}\ \mid f \mapsto v \in \mathcal{F}_i,\ f \notin \mathcal{E}_i,\ n = \mathcal{A}_i(f)\}$$
$$\text{and } \textit{imeths}_i = \{\ \underline{n(x^*) = t_i\hat{}m(x^*)}\ \mid m \mapsto \langle x^*, e\rangle \in \mathcal{M}_i,\ m \notin \mathcal{E}_i,\ n = \mathcal{A}_i(m)\}$$
$$\text{and } \textit{ifields}_1 \cap \cdots \cap \textit{ifields}_n \cap \textit{field}^* = \emptyset$$
$$\text{and } \textit{imeths}_1 \cap \cdots \cap \textit{imeths}_n \cap \textit{meth}^* = \emptyset$$

Fig. 11. Additional reduction rule for AMBIENTTALKLITE

The following example illustrates how trait composition is expressed in terms of delegation. The code on the left summarizes the essence of the animation trait example from Section 3.2. The resulting store is depicted on the right. Note the generated `startMoving` method of the *morph* object.

```
animationtrait := object {
    start() = STARTCODE
    stop() = STOPCODE
}

morph := object {
  animate() = ANIMATECODE
  import animationtrait
   alias startMoving<-start exclude stop
}
```

$$\mathcal{S} \equiv \{$$
$$\quad \textit{animationtrait} \mapsto \langle\emptyset, \{$$
$$\quad\quad \text{start} \mapsto \langle[], \text{STARTCODE}\rangle,$$
$$\quad\quad \text{stop} \mapsto \langle[], \text{STOPCODE}\rangle\}\rangle$$
$$\quad \textit{morph} \mapsto \langle\emptyset, \{$$
$$\quad\quad \text{animate} \mapsto \langle[], \text{ANIMATECODE}\rangle$$
$$\quad\quad \text{startMoving} \mapsto$$
$$\quad\quad\quad \langle[], \textit{animationtrait}\hat{}\text{start}()\rangle\}\rangle\}$$

This concludes our description of the operational semantics of trait composition in AmbientTalk. In the following Section, we discuss how state and visibility control for traits are expressed using this operational semantics.

6 Properties and Discussion

State and visibility control. From the operational semantics of AMBIENTTALKLITE we can derive how state and visibility control are expressed through lexical nesting:

- *Public state*. Trait objects can be stateful by declaring public fields which are explicitly copied into the composing object (cf. Figure 11).
- *Lexically hidden state*. Trait objects can depend upon a field of an object in which they are nested. These fields are not copied into the composing object. Rather, the field remains accessible from the original trait method by lexically referring to it (cf. the syntax extension presented in Figure 6). This is possible because all trait methods close over their lexical scope when created (cf. Figure 8).
- *Lexical visibility control*. An object creation expression that is lexically nested within another object creation expression can refer to the fields and methods of the outer expression. However, an object that has a reference to the nested object cannot access these outer fields or methods via that nested object. As expressed in the [*get*] and [*send*] rules (Figure 5), the lexical scope of an object is not involved in external field access or method invocation. As a consequence, outer fields and methods are inaccessible to clients of the inner object.
- *Shared visibility*. When two object creation expressions are lexically nested within the same outer object creation expression, the two inner objects may refer to the same outer field and method declaration. This allows for sharing state and behavior while keeping it private to external clients. In the operational semantics, sharing is expressed in terms of two lexical environments L_1 and L_2 that may both forward to the same lexical environment L_3 when a name n is not found locally (cf. Figure 8).

Limitations. Traits do not by themselves support advanced resolution strategies required to resolve any unanticipated name conflicts. If two traits are composed that intentionally (rather than accidentally) provide a method with the same name, neither exclusion nor aliasing is an appropriate solution to resolve this conflict. One possible solution would be to completely rename one of the two methods (which, unlike aliasing, requires changing the calls to that method in the method bodies of one of the traits). Another solution would be to change the visibility of one of the two methods, such that it effectively becomes private to its trait. Lexically nested traits by themselves support neither renaming nor changing the visibility of imported methods. Hence, dealing with unanticipated conflicts remains an open issue even with lexically nested traits.

Because lexically nested traits can be stateful, they reintroduce the problem of duplicated state in the case of "diamond inheritance". For example, an object may import two traits `A` and `B`, and these two traits both import a third trait `C` which is stateful. The composite object will then acquire `C`'s state twice. To avoid such issues, one must revert to stateless traits that use accessor and mutator methods to manipulate their state, which is deferred to the composite, as in the original Traits model [13].

Cost. In AmbientTalk, a delegated method invocation has the same runtime cost as a normal method invocation. Without additional optimizations, invoking an imported "delegate method" on a composite is about twice as expensive as a normal method call,

because of the additional delegation to the trait. Caching techniques can be used to reduce this overhead, by storing the imported method rather than the delegating method in the composite object's method cache. Improving the performance of our implementation is an area of future work.

Summary. The Trait model supported by AmbientTalk adds both state and visibility control to traits via lexical nesting. Trait composition is made independent of lexical nesting by introducing delegate methods in the composite. Such methods explicitly delegate messages to the imported trait, leaving the lexical environment of the trait's methods intact. Our model does not introduce any additional composition operators.

7 Related Work

Traits in Self. The term *traits* was introduced in the prototype-based language Self to refer to objects that factor out behavior common to objects of the same type [16]. Self traits are not a special kind of object: any object can be a trait and objects "use" a trait by delegating to it using Self's support for object-based inheritance. Like AmbientTalk traits, Self traits exploit delegation to access the "composite" (e.g. for accessing state or invoking an overridden method) by means of late-bound self-sends. Self traits can be stateful, but state is *shared* by all objects using the trait (like class variables shared between all instances).

When multiple inheritance was added to Self, an object was able to specify multiple parent objects, and could thus use traits as mixins. However, since Self relies on object-based (multiple) inheritance to enable the use of (multiple) traits, naming conflicts are not explicitly resolved by the composite object. Resolving such conflicts is instead done by the method lookup algorithm. Self later abandoned multiple inheritance due to its complexity in favor of a simpler "copy-down" approach [30].

AmbientTalk combines the properties of Self traits and Squeak traits. This is illustrated in Figure 12 by means of the example presented in Section 3.1. An `Enumerable` trait defines a `collect:` method and requires an `each:` method which is defined by the composite, in this case a `Range` object that represents an interval.

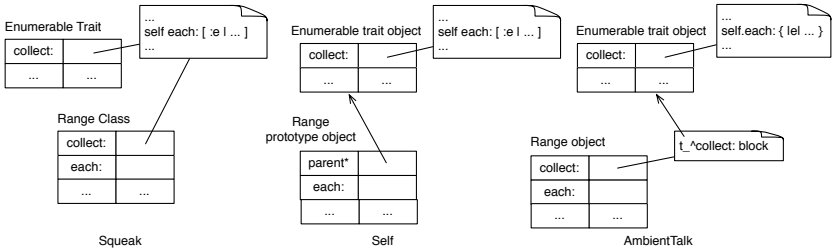

Fig. 12. Comparing trait composition in Squeak, Self and AmbientTalk.

Like Self traits, AmbientTalk traits are object-based: any object can be a trait and both languages use the mechanism of delegation to allow the trait to access the composite. Note that in Self the composite delegates to the trait by means of object-based inheritance (via a so-called "parent slot") while in AmbientTalk delegation to the trait happens by means of the `^` operator in the delegating method.

Like Squeak traits, AmbientTalk traits require an explicit composition operation (**`import`**) during which naming conflicts must be explicitly resolved by the composite. Note how trait composition is compiled away in Squeak by means of the flattening property: the `collect:` method is added to class `Range`. Similarly, in AmbientTalk the trait composition is transformed by adding a delegate method to the composite object. Note that contrary to Self, the trait is not directly accessible from the composite object. Rather, it is referred to by means of a variable private to the delegating method.

Neither Self nor Smalltalk exploit lexical nesting of objects or classes. Similarly, neither language provides a means to hide the visibility of certain methods. Thus, traits defined in these languages have no standard means of controlling visibility. AmbientTalk exploits an object's lexical scope to restrict the visibility of state and behavior. Since a trait is a regular AmbientTalk object, it can continue to use this technique to restrict the visibility of the state or behavior that it needs, but does not want to provide to its clients.

Jigsaw. In his PhD work, Bracha [36] 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). By being an object generator, a module acts as a class 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 are provided to compose modules.

In Jigsaw, modules may refer to each other, and functions defined in one are invokable by others according to some visibility rules. Behavior may be shared by merging two modules, or making one override another one. Using these operators, a mixin and its application are nicely modeled. It appears that the same set of operators is used to express both module specialization and mixin uses.

Traits require a different composition operator than inheritance. Traits cannot be expressed in Jigsaw directly because it imposes an ordering on mixins, while one of the design principles of traits states that the composition order of traits is irrelevant.

CHAI. Smith and Drossopoulou [6] designed the language Chai, which incorporates statically typed traits into a Java-like language. Three different roles for traits in Chai were explored in separate languages: $Chai_1$ (traits may be used by classes), $Chai_2$ (a trait may be a type), and $Chai_3$ (traits play a role at runtime). This third language allows traits to be added to objects at runtime, thus changing their behavior.

The differences with AmbienTalk's traits are significant. AmbientTalk is dynamically typed and its traits are first class values. Any object can be used as a trait simply by importing it into another object. In $Chai_3$ only a trait subtype can be applied to an object. Consequently, a more restricted set of traits may be applied to a given object but type safety is upheld.

8 Conclusion

Traits have originally been presented as groups of reusable methods, without state and a mechanism to control the visibility of provided methods. Extensions have been proposed to add these properties (stateful and freezable traits, respectively) but these models introduce many ad hoc operators that have not before been combined into a unified model supporting both properties. This paper demonstrates that state and visibility control can be added to traits by means of just one linguistic mechanism: lexical nesting.

We have shown that introducing trait composition in a host language that supports lexical nesting requires special attention. Whereas in the original model the flattening property allows trait composition to be implemented by almost literally copying the methods provided by the trait into the composite, this approach fails to hold when traits can be lexically nested, because their methods may refer to lexically free variables. Our approach to solving this problem is based upon delegation of messages. Trait composition is described in terms of generating delegating methods, whose purpose is to delegate a received message to the imported trait object. The actual method invocation is then performed in the proper lexical environment (*i.e.,* that of the trait), but delegation ensures that the trait can still access its required methods via **self** sends. Because methods are generated in the composite, trait composition remains explicit. Name clashes must still be resolved by the composing object, staying true to the original design principles behind traits. Our proposed model has been validated by implementing it in a concrete host language, AmbientTalk, and by describing in detail its operational semantics for a calculus, AMBIENTTALKLITE.

This work started out as an investigation of how traits, a composition mechanism that has been very successfully applied to class-based languages, could be applied to our object-based AmbientTalk language. It became apparent that AmbientTalk's ability to lexically nest objects lead to a simpler trait model since state and visibility control are supported without introducing any additional composition operators. We did need delegation as an additional mechanism to ensure that trait methods can both refer to names in their lexical scope as well as to names provided by the composite. However, our experience tells us that the basic model only relies on lexical nesting, not on the fact that our language is object-based. We think that our model of lexically nested traits can thus be applied more generally to class-based languages as well, provided that they allow classes to be nested and that they provide a solution to the problem of flattening traits in the presence of lexical nesting.

Acknowledgements

We would like to thank Yves Vandriessche who designed and implemented the AmbientMorphic framework. We also like to thank all reviewers and Erik Ernst in particular, for his valuable improvements to the formal semantics.

References

1. : Slate http://slate.tunes.org.

2. Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., Kay, A.: Back to the future: The story of Squeak, a practical Smalltalk written in itself. In: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, ACM Press (November 1997) 318–326
3. Flatt, M., Finder, R.B., Felleisen, M.: Scheme with classes, mixins and traits. In: AAPLAS 2006. (2006)
4. : The Fortress language specification http://research.sun.com/projects/plrg/fortress0866.pdf.
5. Fisher, K., Reppy, J.: Statically typed traits. Technical Report TR-2003-13, University of Chicago, Department of Computer Science (December 2003)
6. Smith, C., Drossopoulou, S.: Chai: Typed traits in Java. In: Proceedings ECOOP 2005. (2005)
7. Liquori, L., Spiwack, A.: FeatherTrait: A modest extension of Featherweight Java. ACM Transactions on Programming Languages and Systems (TOPLAS) **30**(2) (2008) 1–32
8. Reppy, J., Turon, A.: Metaprogramming with traits. In: Proceedings of European Conference on Object-Oriented Programming (ECOOP'2007). (2007)
9. Bergel, A., Ducasse, S., Nierstrasz, O., Wuyts, R.: Stateful traits and their formalization. Journal of Computer Languages, Systems and Structures **34**(2-3) (2007) 83–108
10. Ducasse, S., Wuyts, R., Bergel, A., Nierstrasz, O.: User-changeable visibility: Resolving unanticipated name clashes in traits. In: Proceedings of 22nd International Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM Press (October 2007) 171–190
11. Bak, L., Bracha, G., Grarup, S., Griesemer, R., Griswold, D., Hölzle, U.: Mixins in Strongtalk. In: ECOOP '02 Workshop on Inheritance. (2002)
12. Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., Black, A.: Traits: A mechanism for fine-grained reuse. ACM Transactions on Programming Languages and Systems (TOPLAS) **28**(2) (March 2006) 331–388
13. Schärli, N., Ducasse, S., Nierstrasz, O., Black, A.: Traits: Composable units of behavior. In: Proceedings of European Conference on Object-Oriented Programming (ECOOP'03). Volume 2743 of LNCS., Springer Verlag (July 2003) 248–274
14. Reppy, J., Turon, A.: A foundation for trait-based metaprogramming. In: International Workshop on Foundations and Developments of Object-Oriented Languages. (2006)
15. Mens, T., van Limberghen, M.: Encapsulation and composition as orthogonal operators on mixins: A solution to multiple inheritance problems. Object Oriented Systems **3**(1) (1996) 1–30
16. Ungar, D., Chambers, C., Chang, B.W., Hölzle, U.: Organizing programs without classes. LISP and SYMBOLIC COMPUTATION **4**(3) (1991)
17. Reddy, U.: Objects as closures: abstract semantics of object-oriented languages. In: LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming, New York, NY, USA, ACM (1988) 289–297
18. Abelson, H., Sussman, G.J., Sussman, J.: Structure and interpretation of computer programs. MIT electrical engineering and computer science series. McGraw-Hill (1991)
19. Dickey, K.: Scheming with objects. AI Expert **7**(10) (October 1992) 24–33
20. Black, A., Hutchinson, N., Jul, E., Levy, H.: Object structure in the Emerald system. In: Proceedings OOPSLA '86, ACM SIGPLAN Notices. Volume 21. (November 1986) 78–86
21. International, E.C.M.A.: ECMA-262: ECMAScript Language Specification. Third edn. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland (December 1999)
22. Madsen, O.L., Moller-Pedersen, B., Nygaard, K.: Object-Oriented Programming in the Beta Programming Language. Addison Wesley, Reading, Mass. (1993)

23. Miller, M.S.: Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA (May 2006)
24. Bracha, G.: On the interaction of method lookup and scope with inheritance and nesting. In: 3rd ECOOP Workshop on Dynamic Languages and Applications. (2007)
25. Dedecker, J., Cutsem, T.V., Mostinckx, S., Theo D'Hondt, W.D.M.: Ambient-oriented programming in ambienttalk. In Thomas, D., ed.: Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP '06). Volume 4067., Springer-Verlag (2006) 230–254
26. Van Cutsem, T., Mostinckx, S., Boix, E., Dedecker, J., De Meuter, W.: Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks. Chilean Society of Computer Science, 2007. SCCC '07. XXVI International Conference of the (Nov. 2007) 3–12
27. Ungar, D., Smith, R.B.: Self: The power of simplicity. In: Proceedings OOPSLA '87, ACM SIGPLAN Notices. Volume 22. (December 1987) 227–242
28. Lieberman, H.: Using prototypical objects to implement shared behavior in object oriented systems. In: Proceedings OOPSLA '86, ACM SIGPLAN Notices. Volume 21. (November 1986) 214–223
29. Maloney, J.H., Smith, R.B.: Directness and liveness in the morphic user interface construction environment. In: UIST '95: Proceedings of the 8th annual ACM symposium on User interface and software technology, New York, NY, USA, ACM (1995) 21–28
30. Smith, R.B., Ungar, D.: Programming as an experience: The inspiration for self. In Olthoff, W., ed.: Proceedings ECOOP '95. Volume 952 of LNCS., Aarhus, Denmark, Springer-Verlag (August 1995) 303–330
31. Anderson, C., Drossopoulou, S.: delta an imperative object based calculus. In: Proceedings of USE 2002. (2002)
32. Bettini, L., Bono, V.: Type Safe Dynamic Object Delegation in Class-based Languages. In: Proc. of PPPJ, Principles and Practice of Programming in Java, ACM Press (2008)
33. Bono, V., Fisher, K.: An imperative, first-order calculus with object extension. In: Proceedings of the 12th European Conference on Object-Oriented Programming, London, UK, Springer-Verlag (1998) 462–497
34. Flatt, M., Krishnamurthi, S., Felleisen, M.: Classes and mixins. In: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press (1998) 171–183
35. Felleisen, M., Hieb, R.: The revised report on the syntactic theories of sequential control and state. Theor. Comput. Sci. **103**(2) (1992) 235–271
36. Bracha, G.: The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance. PhD thesis, Dept. of Computer Science, University of Utah (1992)