Modular Composition and State Update in Plaid

Jonathan Aldrich
School of Computer Science
Carnegie Mellon University
aldrich@cs.cmu.edu

Karl Naden
School of Computer Science
Carnegie Mellon University
kbn@cs.cmu.edu

Éric Tanter
PLEIAD Laboratory
Computer Science
Department (DCC)
University of Chile
etanter@dcc.uchile.cl

ABSTRACT

At the core of the Plaid typestate-oriented programming language is the ability to change the representation of an object at run-time. As such, the semantics of the state change operation impact how Plaid programs are structured and how objects are composed in Plaid's trait-based composition system. We describe the challenges with respect to designing a modular state change operation and suggest two options. Secondly, we explore the issues both designs create for explicit conflict resolution and sketch a potential solution which eliminates the flattening property to allow conflicts to be resolved only when they come into scope.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*

General Terms

Design, Languages, Theory.

Keywords

Typestate, State, Trait, Composition, Plaid, Semantics.

1. INTRODUCTION

As the software industry has matured, software development has shifted away from implementing data structures and algorithms and towards building systems by tying reusable components together with application-specific logic. This change has yielded great gains in productivity, but fully taking advantage of component-based software engineering requires better models for reusable components. A central challenge is expressing the correct protocol for using components: for example, initializing them before use, cleaning them up when done, and not using them afterwards.

To address these challenges, we are developing a new typestate-oriented [2] programming language called Plaid[1]. Typestate-Oriented Programming extends the standard object-oriented paradigm by making the potential states of an object first class and providing a novel state change operation to transition between states. This operation can add, remove, and update methods and fields, thus affecting both the representation and the behavior of the object. For example, the open() method on a file abstraction in Plaid would use the state change operation to add a read() method to the object. In languages such as Java, this change of behavior occurs only implicitly because the read() method would always be available and simply have different (exceptional) behavior before open() was called. We believe that providing a way to make state change explicit allows code to more directly follow design. In turn, this may enhance documentation, reduce errors, help developers diagnose problems more quickly, and improve engineering productivity when developing component-based software.

Composition Challenges. In this paper, we introduce the state change operation and Plaid's trait-based [15] composition model via an example. An evaluation of the example reveals several issues which stem from the interaction between program structure, state change, and composition. We explore these and propose preliminary solutions for the following questions:

- How can the semantics of Plaid's state change operation support a modular program structure?
- How can trait conflicts caused by state changes be resolved explicitly under these semantics?

We discuss the most closely related work at relevant points in the paper, leaving a discussion of the broad area of related work for the end.

2. COMPOSITION AND STATE CHANGE

Basic Composition in Plaid. To illustrate the challenges in integrating trait-based state composition and state change, consider the example of a Turing Machine implemented in Plaid (taken from [1]). The syntax of Plaid is Java-like, but replaces class declarations with **state** declarations, requires keywords for **method** and immutable field (**val**) declarations, and is expression-oriented (thus not requiring an explicit return statement).

We model the tape of the Turing machine as a sequence of Cell objects, each of which keeps track of its left and

[1] `http://www.plaid-lang.org/`

MASPEGHI '10, June 22, 2010, Maribor, Slovenia

right neighbors on the tape. Each cell is composed with a Symbol (Zero or One) representing the symbol currently printed at that location on the tape. Zero and One are each a **case of** Symbol, meaning they inherit the declarations in Symbol and code can test whether any Symbol is a Zero or One as with Java's instanceof test. We create a Cell with an expression like **new** Cell **with** Zero. Here, the **with** operator is state composition, and is exactly analogous to trait composition. The resulting object has all the members of both Cell and Zero.

```
state Cell {
    method getLeft() {left}
    method getRight() {right}
    val left;
    val right;
}
state Symbol { }
state Zero case of Symbol {
    method writeZero { }
    method writeOne() {
        val lt = getLeft();
        val rt = getRight();
        this <− Cell with One { left = lt; right = rt }
    }
    method printVal() { System.out.print("0") }
}
state One case of Symbol {
    method writeZero() {//analogous to Zero.writeOne()}
    method writeOne() { }
    method printVal() { System.out.print("1") }
}
```

Listing 1: Modeling Tape Cells

The state change operator (<−) is used in the writeOne() method. This operation updates the target object to match the state expression to the right of the arrow. In this example, the state change updates the state of the cell from Cell **with** Zero to Cell **with** One. Semantically, this updates the implementations of the writeOne(), writeZero(), and printVal() methods to those defined in One, but leaves the left and right fields as they were before.

A Second Look. A closer look at this example reveals some subtleties in the interaction between state change and composition. While the code places the Cell and Symbol in different hierarchies in an attempt to allow individual reuse, the state change breaks this separation. The current semantics of Plaid specify that a state change removes *all existing state information* from the target object on the left before adding everything from the right hand state expression. While these semantics are clean and simple, they prevent reuse by tightly coupling the implementations of states that will be used together. In this example, the Zero and One could not be used except when composed with a Cell.

The core issue with these semantics for state update is that they require that the entire composition of an object be explicitly reconstructed as part of a state change. However, for a given state to be reusable in multiple contexts, it must be able to deal abstractly with the portions of the object's state which are orthogonal to the state it is changing. In the above example, the Cell and Symbol states were designed to be separate by implementing them as distinct states. A modular semantics for the state change operation would support that separation.

2.1 Towards Modular State Update

We now explore the design of possible semantics for a modular state update. The idea is to consider an individual state as a relevant unit of granularity for state updates.

Restricted Update. In this first variant of modular state update, the state change operator only affects that part of the state that "corresponds" to the new state. State correspondance is defined in terms of state refinement as specified by the **case of** extension operator. When an object is updated with a new state (eg. One in our example), then the least common ancestor(s) of the current object and the new state in the **case of** forest are considered the *root(s) of the update* (eg. Symbol). That part of the state is replaced by the new state. Any other state (eg. Cell) is considered orthogonal, and therefore not affected by the update.

In our example, this means that the writeOne() method of the Zero state can be simply defined as:

```
method writeOne() { this <− One }
```

It is no longer necessary to explicitly reconstitute the Cell portion of the object. Because Cell is orthogonal to the added state One (that is, it is not part of the same **case of** tree), the members associated with it remain. The only effect of the method then is to replace the Zero state and all its associated members with the One state and all its members in the object's representation.

Note that because states can be composed (eg. using **with**) it is possible that the new state is comprised of several parts. In this case, there may be multiple roots, one for each of the **case of** trees involved in the update. For instance:

```
this <− Cell with One { ... }
```

has the same effect as previously: both the Cell and Symbol states are roots of update, so the whole object is replaced by the new state.

Projected Update. While the previous design has the advantage of simplicity, it does not make it possible to use a state change to discard certain states that are unrelated to the added state (in the sense of not being part of the same **case of** tree). Only states that are declared to be mutually exclusive with the new state are discarded. In particular, it makes full object update more cumbersome because one needs to be sure to specify a new state that covers all facets of the current state.

The notion of a *projected update* represents an alternative design which offers more flexibility at the cost of a slight increase in complexity. In a projected update, the programmer explicitly specifies the parts of an object that are affected by the state change. We introduce the projection operator, "|", which is parametrized by a list of states. For instance:

```
this|Symbol <− One
```

updates the Symbol part of the object to the One state. With projected update, it is straightforward to get rid of an

orthogonal state. For instance, suppose that a Cache state is associated to the object and it should be discarded when changing to One:

```
this|Symbol,Cache <− One
```

Under these semantics, the update operator without projection refers to the full-blown state update discussed in the first section:

```
this <− One
```

updates the entire object to just the One state; in that case, it would discard the Cell state.

2.2 Plaid's Design

The current specification of the Plaid language [11] specifies Restricted Update as the design which will be implemented in future versions of Plaid. This choice was mostly pragmatic: it provides a fairly lightweight syntax and simple semantics that covers a use case we are confident we will need. On the other hand, it seems reasonably likely that in some cases we will need the Projected Update semantics also. Our plan is to gain experience with the language before committing to this additional complexity.

3. COMPOSITION CONFLICTS

By dealing with members of an object differently during state change depending on which state they were defined in, we have introduced a complication to conflict resolution. We use our running example to introduce Plaid's conflict resolution operators, show how unresolvable conflicts can arise, and sketch a possible solution.

White Box Conflict Resolution. Plaid provides standard trait operators to resolve conflicts in the case where the structure of the combined states is known. Take, for instance, a new Symbol Three implemented as a binary pair:

```
state Pair {
    method getLeft() {l}
    method getRight() {r}
    val l;
    val r;
}
state Three case of Symbol =
          Pair {l = 1; r = 1} with {
    method printVal() {
        val theVal = 2∗getLeft() + getRight();
        System.out.print(theVal)
    }
}
```

Composing Cell **with** Three would result in a conflict because both states define the getLeft() and getRight() methods. Since the methods are semantically different, we do not want to use aliasing and exclusion operations, but rather we'd like to use Reppy and Turon's deep renaming operation [14] that redirects existing calls in Three to the new method name:

```
Cell with Three [getLeft()−> getLeftVal(),
                 getRight()−> getRightVal()]
```

Three's getRight() and getLeft() methods can also be seen as implementation details which should be hidden from the public interface of Three. Therefore, Plaid also provides Reppy and Turon's hide operator [14]. The statement

```
Cell with Three [− getLeft(), − getRight()]
```

resolves the conflict by removing Three's getLeft() and getRight() methods from the public interface of the composed object. However, they remain visible to methods defined in Three.

Black Box Conflict Resolution. The conflict resolution operators are only effective when all the members of an object are fully known. With state change semantics proposed above, this will not always be the case. Take for example a new method defined in the Zero state:

```
method writeThree() { this <− Three }
```

This code would then create a conflict:

```
val x = new Cell with Zero;
x.writeThree()
```

As both Three and Cell define both a getLeft() and a getRight() method, the state change from Zero to Three in the writeThree() method would result in an object with conflicting implementations of these methods. Under the semantics detailed above, there is no way in general to resolve these conflicts at the time of the state change because the programmer cannot refer to the members of orthogonal states like Cell. Providing the ability to do so would only bring us back to the tight coupling of states which we previously rejected because it prevented modularity.

Instead, we propose to have states define their own namespaces as a way to push conflict resolution to the point where the composition was defined. In our example, this would mean that during the call to writeThree(), the members of the Cell state would not be in scope and thus no conflicts would appear immediately as a result of the state change. Once the writeThree() call returned, however, all members of both the Cell and Three states would be visible, creating an implicit composition of Cell **with** Three. The conflicts could then be resolved explicitly at the call site using white box conflict resolutions operators:

```
x.writeThree() [− Three.getLeft(), − Three.getRight()]
```

We believe that this design follows the spirit of conflict resolution in traits because it pushes explicit resolution of conflicts to the code which introduced the composition of Cell and Three while also maintaining modularity. While this simple example belies the complexity of the general case of this scheme, we believe it to be a promising line for investigation. As such, we leave a precise characterization of the solution, including how scopes are defined, for future work.

4. RELATED WORK

Typestate. The idea of typestate-oriented programming [2] has multiple precedents. Typestate was proposed as an abstraction of the operations currently available on an object, which may change as the program executes [16]. Most prior

work on typestate has focused on statically verifying correct typestate usage using various techniques, including program analysis [9, 5]. In another approach, Bierhoff and Aldrich developed a type system [4] to track typestate. Their system includes the notion of separating updates to orthogonal portions of the typestate which corresponds to our Restricted Update. Typestate-oriented programming generalizes typestate verification by combining a type-based approach to checking typestate usage with states as first-class entities at run time.

Runtime Support for States. The Actor model [10] was one of the first programming models to treat states in a first class way. An Actor can accept one of several messages; in response, it can perform computation, create other actors, send messages, and finally determine its next state—i.e. how to respond to the next message received. Our state-orientated approach draws inspiration from actors, but in this paper we do not consider concurrency.

Smalltalk [12] introduced a `become` method that allows an object to exchange state and behavior with another object, which can be used to model state changes in a first-class way. In a related approach, the Self language [19] allows an object to change the objects it delegates to (i.e. inherits from), also providing a way to model state changes.

The concept of a state is related to that of a *role* played in interactions with other object. While most research in the area uses roles to describe different (simultaneous) views of an object, Pernici proposed state-like roles where objects can transition from one role to another [13].

From the object modeling point of view, the closest work to Plaid is Taivalsaari's proposal to extend class-based languages with explicit definitions of logical states (modes), each with its own set of operations and corresponding implementations [17]. Our proposed object model differs in providing explicit state transitions (rather than implicit ones determined by fields) and in allowing different fields in different states. Another similar system is Fickle, a language in which objects can change from one "state class" to another at run time [8].

Traits. The notion of traits originated in Self [19]: trait objects are used to factor out common features [18]. However, traits in this setting are but a mere design pattern: a trait is just a parent object, and there are no composition operators to address potential conflicts. Our proposal, while still prototype-based, is rather inspired by Schaerli's traits [15], in particular with respect to composition operators. Schaerli's traits are stateless, but an extension has been proposed to deal with state [3]. Traits in Plaid are stateful as well.

A major departure of our work from much of the literature on traits is the lack of the flattening property. According to this property, traits are just a composition facility which is compiled away and does not affect the semantics of the resulting program. Because Plaid's traits are the units at which state update is performed and define namespaces they become semantically important at runtime. Other systems which build more complexity into trait-based systems [14, 7] have previously found the need to dispense with the flattening property. Our work follows these in trading the simplicity offered by the flattening property for more flexibility and power in the runtime system. In this way, our system is also closely related to Bracha's Jigsaw modularity framework [6] which first introduced advanced operators such as hide.

Acknowledgments

We thank members of the Plaid research group for their feedback and contributions to the design of Plaid and its composition system. This research was supported by DARPA grant #HR00110710019 and NSF award #CCF-0811592.

5. REFERENCES

[1] J. Aldrich. Holy States can Save the World! In *Proc. SIGBOVIK*, 2010.

[2] J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. Typestate-Oriented Programming. In *Proc. Onward!*, 2009.

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

[4] K. Bierhoff and J. Aldrich. Modular Typestate Checking of Aliased Objects. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications*, 2007.

[5] E. Bodden, L. Hendren, P. Lam, O. Lhotak, and N. A. Naeem. Collaborative Runtime Verification with Tracematches. *Oxford Journal of Logics and Computation*, 20(3):707–723, 2010.

[6] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity, and Multiple Inheritance*. PhD thesis, The University of Utah, March 1992.

[7] T. V. Cutsem, A. Bergel, S. Ducasse, and W. D. Meuter. Adding state and visibility control to traits using lexical nesting. In *Proc. European Conference on Object-Oriented Programming*, 2009.

[8] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Fickle: Dynamic Object Re-classification. In *Proc. European Conference on Object-Oriented Programming*, 2001.

[9] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective Typestate Verification in the Presence of Aliasing. *Transactions on Software Engineering and Methodology*, 17(2), 2008.

[10] C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular Actor Formalism for Artificial Intelligence. In *Proc. International Joint Conference on Artificial Intelligence*, 1973.

[11] Jonathan Aldrich. *The Plaid Language: Dynamic Core Specification, version 0.1.2*, 2010. `http://plaid-lang.googlecode.com/hg/docs/spec/versions/plaid-spec-v0_1_2.pdf`.

[12] A. C. Kay. The Early History of Smalltalk. *SIGPLAN Notices*, 28(3), 1993.

[13] B. Pernici. Objects with Roles. In *Proc. Conference on Office Information Systems*, 1990.

[14] J. Reppy and A. Turon. Metaprogramming with traits. In *Proc. European Conference on Object-Oriented Programming*, pages 373–398, July-August 2007.

[15] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable Units of Behaviour. In *Proc. European Conference on Object-Oriented Programming*, 2003.

[16] R. E. Strom and S. Yemini. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, 1986.

[17] A. Taivalsaari. Object-Oriented Programming with Modes. *Journal of Object-Oriented Programming*, 6(3):25–32, 1993.

[18] D. Ungar, C. Chambers, B.-W. Change, and U. Hölzle. Organizing programs without classes. *Lisp and Symbolic Computation*, 4(3):223–242, July 1991.

[19] D. Ungar and R. B. Smith. Self: The Power of Simplicity. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications*, 1987.