Resource-Based Programming in Plaid

Jonathan Aldrich

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

Abstract

Many modern programming challenges center on the correct handling of abstract *resources* whose use is constrained in some way. These constraints include initialization before use, resource cleanup, safe coordination among threads, and usage protocols. Unlike class-based languages, the resource-based programming language Plaid models interfaces, representation, and behavior using *states*, and an object's state can change. Plaid's gradual, linear logic-based type system and runtime system will track both the state of an object and aliases to it, ensuring that clients access objects safely in both sequential and concurrent programs.

1. Motivation: Resources

As the software industry has matured, software development has shifted away from data structure and algorithm implementation and towards building systems by tying resuable components together with application-specific logic. This change has yielded great gains in productivity, but reusing these components can also be difficult. A central issue is managing *resources*: stateful objects whose use is constrained in some way. These constraints include initialization before use, resource cleanup, safe coordination among threads, and other usage protocols. They can be complex: the ResultSet class from Java's JDBC library has 33 states and 200 methods with state constraints. Protocols are also widespread, including many stateful data structures and I/O libraries. They also cause problems for developers in practice: Jaspan found that a substantial fraction (near 20%) of the understandable postings in an ASP.NET help forum were related to protocol constraints [2].

Existing programming systems have minimal support for resources. In Java, finally blocks assist with resource cleanup on exceptional paths, but it is easy to forget to add finalizers. Other researchers have investigated analyses and type systems for avoiding race conditions or verifying typestate properties [5]. However, no language we know of explicitly models resources and their usage constraints in the object model and native type system. In this paper we sketch a design for such a language and argue that explicitly modeling resources in the language can help programmers use them better.

2. The Plaid Language

First-Class States in Plaid. We are developing a new programming language, Plaid, which is designed to facilitate programming with first-class resources. Plaid supports resources mainly via *typestate-oriented programming* [1], a new programming paradigm that extends object-oriented programming with first-class states. Rather than be a member of a fixed class, each object has a *typestate* that is changeable. Unlike typestate analyses, states in Plaid are first-class, meaning they define not just an interface but also representation and behavior.

```
1  state Buffer
2      comprises EmptyBuffer, FullBuffer { }
3
4  state EmptyBuffer case of Buffer {
5      method void put(unique Element ≫ none e)
6              [EmptyBuffer ≫ FullBuffer] {
7          this ← FullBuffer { elem = e };
8      }
9  }
10
11 state FullBuffer case of Buffer {
12     requires unique Element elem;
13     method unique Element get()
14             [FullBuffer ≫ EmptyBuffer] {
15         val e = elem;
16         this ← EmptyBuffer { };
17         e;
18     }
19 }
```

Figure 1. A Buffer Implementation in Plaid.

For example, consider the Buffer implementation in Figure 1. The Buffer state is abstract; the **comprises** clause ensures that every Buffer is either an EmptyBuffer or a FullBuffer. EmptyBuffer has a single operation put, which accepts an argument e. put is implemented with a state change primitive (the left arrow), indicating that the object **this** should be transitioned into the FullBuffer state.

The FullBuffer state requires a field elem, and so when we transition from EmptyBuffer to FullBuffer we must provide a value for this field—conveniently, we can use the element the client passed to put. FullBuffer has a single operation get, which reads the value in the elem field, transitions the receiver object back to the EmptyBuffer state, and returns the old value of the field.

Note that the implementation in Figure 1 is clearer and less error-prone than an equivalent Java implementation. Since Java does not support states explicitly, we'd have to have a single class Buffer that has an elem field as well as get and put methods. Then the protocol that get can only be called when the buffer is full will have to be enforced with some kind of run-time check; such checks are often omitted in practice either by accident or because they are costly. Furthermore, in Java we will have to set the element field to some sentinel value (e.g. null) when the buffer is empty, and then we must be very careful to associate the potential null value with the state of the buffer, otherwise we may get a null pointer exception. In contrast, Plaid's explicit support for states means that methods and fields are only present in the states where they make sense. If the programmer uses the buffer incorrectly then the system can give a good error message, either at compile time or at run time, saying that get cannot be called on an EmptyBuffer.

A State Tracking Type System. We would like the Plaid compiler to provide pragmatic help in using objects correctly, by analogy with type systems in languages like Java. In this case, however, the compiler must track the type of a buffer flow-sensitively as it will change when an element is added or removed.

Each method therefore declares how it changes the state of the receiver (and of each argument, if relevant). The declaration goes in brackets after the method declaration, with ≫ showing the direction of state change. For example, the signature of put declares that the receiver changes from the EmptyBuffer to the FullBuffer state.

Permission-based Types. Tracking states is easy if there is only one reference to an object, but in the presence of aliasing it becomes difficult because the state of the object can change due to an effect through an alias. Our type system is thus based on a linear logic, and we provide various permissions that express whether there can be aliasing or not. For example, **unique** means that we have the only reference to an object, while **share** means that there may be other aliases. A **unique** permission can be converted into an **immutable** permission (destroying the original **unique**), which allows other aliases but ensures that the object cannot be modified through them. Linear logic ensures that a variable's permission is a sound description of how it is aliased.

In Figure 1, the argument to put is **unique** and is in the Element state. When the method returns, however, no permission to that element is returned to the caller, because we have created a **unique** field reference to that element in the FullBuffer state, and returning a permission back to the caller would violate the uniqueness invariant. That is, the caller may still have a reference in scope but it does not have permission to use it. This is indicated with the **none** permission.

Permissions default to **unique** for mutable states; for example, the state transition of the receiver of put is really [**unique** EmptyBuffer ≫ **unique** FullBuffer]. We also allow the declaration of **immutable** states, which means the default permission for objects of that state is **immutable**. This allows programs without aliasing, or purely functional programs without mutation, to be expressed without any additional permission annotation overhead.

Gradual Types and Dynamic Checks. We are designing Plaid to be gradually typed [3], so in fact type declarations like those in Figure 1 are optional. If present, they can help developers understand interfaces better and find errors earlier, but if they are absent then Plaid just falls back on run-time checking. Developers can also insert casts to assert that an object is in a particular permission-based type, or use pattern matching to test the state of an object. The combination of gradual typing, typestate checking, casts, and permission-based types is interesting—consider, for example, how a cast to **unique** might be checked!

Concurrency by Default. Plaid's permissions not only help track state, they also allow the natural concurrent execution of programs [4]. If we have a function fill that fills a buffer, we can create two **unique** buffers and if we call:

```
1 fill(buf1);
2 fill(buf2);
3 combine(buf1, buf2);
```

Plaid's compiler will automatically execute the two calls to fill in parallel, since they operate on different state. The compiler will wait until both fills are complete before calling combine as this requires the same resources as fill.

Of course, in some cases we want to concurrently access shared state. Our model allows this if the programmer is explicit about her intention to "split" the permission to the buffer, presumably having considered the consequences of any possible interference. In the example below, a producer and consumer access a shared buffer concurrently. The Plaid compiler will ensure that the producer and consumer acquire a lock before accessing the shared data, helping to avoid the worst conflicts.

```
1 split (buf) produce(buf) || consume(buf);
```

Overall, our model supports a dataflow style of concurrency, but we integrate the dataflow concurrency with mutable, potentially shared state using permissions.

3. Status and Future Work

We have developed a prototype compiler for Plaid, which already compiles simple dynamically-typed Plaid programs; we plan an open-source public release[1] concurrent with PLDI. In the summer of 2010, we will be building a typechecker and extending the compiler to support concurrent execution.

Many research questions remain:

- The buffer above stores a fixed unique Element; how can we make the buffer generic in the type and permission?
- How can we provide efficient runtime support for representation changes, casts, and gradual types in Plaid?
- Can we keep the permission-based type system simple enough to use in practice?
- Can a compiler automatically come up with an efficient approach to concurrent execution?
- Do programmers find Plaid natural, and does it help them avoid errors and become more productive?

We look forward to answering these questions, and we believe that as programming becomes more and more centered on resources, the ideas in Plaid will be of significant help in implementing and using resource abstractions easily and correctly.

Acknowledgments

The author acknowledges the Plaid research group for their contributions to the design and vision for Plaid. This research was supported by DARPA grant #HR00110710019 and NSF award #CCF-0811592.

References

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

[2] C. Jaspan. *Proper Plugin Protocols*, 2010. Carnegie Mellon University Thesis Proposal.

[3] J. G. Siek and W. Taha. Gradual typing for objects. In *ECOOP'07: 21st European Conference on Object-Oriented Programming*, 2007.

[4] S. Stork, P. Marques, and J. Aldrich. Concurrency by Default: Using Permissions to Express Dataflow in Stateful Programs. In *Proc. Onward!*, 2009.

[5] 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.

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