Efficient Implementation of the Plaid Language

Sarah Chasins

Swarthmore College, Carnegie Mellon University

schasi1@cs.swarthmore.edu, schasins@andrew.cmu.edu

Abstract

The Plaid language introduces native support for state abstractions and state change. While efficient language implementation typically relies on stable object members, state change alters members at runtime. We built a JavaScript compilation target with a novel state representation, which enables fast member access. Cross-language performance comparisons are used for evaluation.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Optimization

General Terms Languages

Keywords Plaid, states, state change

1. Plaid and Abstract State

Consider a simple file object. When it is open, it has read and close methods. When it is closed, it can only be opened. In essence, this single file object has the methods of two distinct classes during these different phases of its use. Yet in typical object-oriented languages, this state information is never directly expressed.

The Plaid language introduces a model in which object state is made explicit[2] [6]. The practice of maintaining implicit state information is pervasive in program design, whether in the case of a simple file or in objects composed of six states simultaneously, or even nested states. By introducing abstract states and explicit state change, Plaid makes these transitions salient to users, facilitating code that depicts object structure more clearly. Without the need to write one's own state checks, code is neater and more compact. Further, where programmers forget to write state checks, the runtime can indicate that a member is unavailable, rather than permit continued execution and possible data corruption.

In Listing 1, simple Plaid code lays out the design of the `File` state whose state space appears in Figure 1. `OpenFile` and `ClosedFile` are substates of `File`; method `close` uses the `<-` operator to cleanly transition between them.

Though state check elimination may seem trivial in the case of a file with two substates, the advantages are clear in the context of a more complex state, such as the `Car` state pictured in Figure 2. This state would be created in Plaid using `case of` to declare the specializing substates (such as `Braking` and `NotBraking`), and the `with` operator to compose independent dimensions (such as `DrivingStatus` and `CleanStatus`). In fact, any state space that can be modeled with statecharts [5] can be expressed in Plaid. In supporting the complex state spaces that can be represented in Plaid — the `Car` state space in Figure 2, the Java JDBC library with its 33 distinct abstract states [3] — the elimination of state checks makes code shorter, cleaner, and less susceptible to programmer errors.

```
state File {
   val filename; }
state OpenFile case of File {
   val filePtr;
   method read() {...}
   method close() { this <- ClosedFile;} }
state ClosedFile case of File {
   method open() { this <- OpenFile;} }
```

Listing 1. Plaid declaration of File state and two substates.

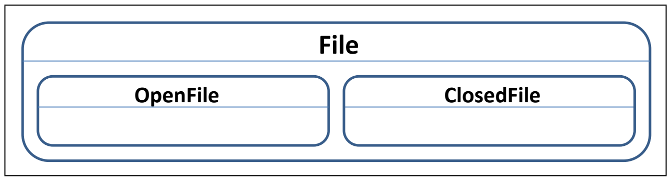

Figure 1. The statechart for the simple file state declared in Listing 1.

2. Related Work

While maintaining state and permitting state change is excellent from the perspective of a programmer, the challenges for implementation are significant. If an object's members do not change, there is never a need to allocate additional space for an object that has already been declared. More importantly, a stable set of members means there can be a standard,

SPLASH'11 Companion, October 22–27, 2011, Portland, Oregon, USA.
ACM 978-1-4503-0940-0/11/10.

consistent way to access particular members throughout program execution, for all instances of a given class. Thus, efficient language implementation has traditionally required that every object have a stable set of members throughout its life cycle. However, state is a useful abstraction only if it is accompanied by state transition — and state transition allows members to change at runtime.

Before the work presented in this paper, the only existing implementation of Plaid was a Plaid to Java compiler which relies heavily on reflection. At runtime, every representation of a Plaid object contains a map of objects that represent members; each member object is essentially a wrapper for a value or for the body of a function. To enact a method call, the runtime first searches for an object with the appropriate name in an object's member map. If the member is found, calling `invoke` on the member object completes the process. If a matching member is not found, the runtime next searches the map of the parent state, if one exists. The runtime repeats this process until the member is found, or until there are no more states to check.

Some dynamic languages like Self [7] provide mechanisms for modifying an object's class at runtime. Self permits directly adding or removing members during execution, and also allows the programmer to mark slots as parent slots. An object inherits all the members of the object stored in a parent slot. Since a parent slot can be mutable, a user can manipulate an object's state by changing the object in a parent slot. During execution, if a message is passed to an object that cannot handle that message, it is passed next to any parent objects. This approach could be used to implement state change, and indeed it corresponds quite closely to the implementation scheme of the Plaid to Java compiler.

3. Member Access and State Change

To date, implementations of object reclassification generally suggest the same intuitive representation for state at runtime. The intuitive representation entails maintaining a target language object for each state and substate: one for `File`, another for `OpenFile`, another for `ClosedFile`. Pointers to the states that currently compose the Plaid object would render state change a simple process. Adding and removing states would be as simple as adding and removing nodes in a tree.

However, consider that a file will commonly be opened once, read many times, and closed once. It is easy to bring to mind many such examples, and it is a rare program that will require more state transitions than method calls or field reads. This being the case, it is essential that member access be fast. In the naive solution above, a call to `close()` would require first searching the `File` object for the desired member, then the `OpenFile` object, before finding the appropriate method. While this may not greatly affect execution time in the case of a simple `File` object, consider the complex `Car` state pictured in Figure 2. The process for finding the

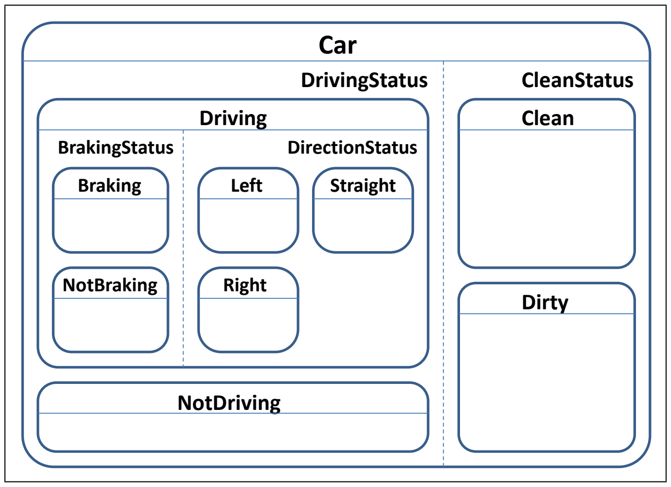

Figure 2. The statechart for a complex car state.

`stopBraking()` method of `BrakingState` would require a search through five to nine objects.

Aside from prioritizing state change over the more common method access, this naive implementation would make the execution time of member access dependent on how a state is composed — more component states and more substates would increase overhead. This discourages the use of the state abstractions Plaid introduces. It is clear that keeping a tree of component objects will not produce an efficient implementation.

4. Implementation

In formulating a new implementation, JavaScript was a natural target for compilation. As a prototype-based language with first-class objects, it shares some features with Plaid that make translation from one to the other clear and intuitive. Given the language's current ubiquity on the web, a JavaScript implementation of Plaid should be useful, and given the carefully optimized virtual machines, a JavaScript implementation should be fast.

The central design question was the matter of how to represent a Plaid object in JavaScript, taking into account the conflicting demands of member usage and state transitions. Efficient member access requires that all methods and fields be part of a single JavaScript object. However, efficient state change would most naturally be implemented by storing members in multiple objects. With only naive solutions in mind, decreased efficiency in one realm seems bound to accompany improved efficiency in the other.

The need for fast method calls and field lookup motivates the central restriction we placed on our implementation: any member of a Plaid object must be a member of a single corresponding JavaScript object. If JavaScript object `f` represents a `File` in the `OpenFile` state, a call to `read` should produce the code `f.read()` in the compiled code, rather than trigger a search through `File` and `OpenFile` objects. With

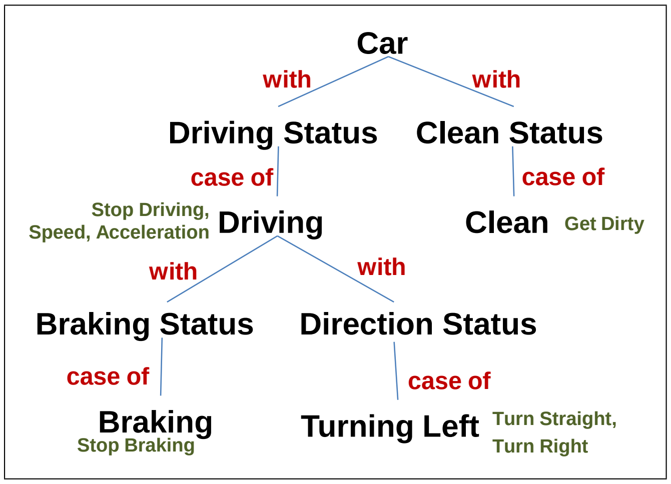

Figure 3. A visual representation of the JavaScript metadata tree for an instance of the `Car` state pictured in Figure 2.

JavaScript's first-class functions, it is trivial to copy all the members of an object's component states and substates to a single object. By maintaining a single JavaScript object with all available members, we ensure that the execution time of member accesses and method calls does not depend on the number of component states that form a Plaid state, or on the depth of the state hierarchy.

However, with this accomplished, how can state change be enacted? When states are added or removed, the runtime must be able to identify which members should be added or removed along with those states. With all of `File`'s members and all of `OpenFile`'s members in a single JavaScript object, there is no clear way to distinguish between `File` and `OpenFile` methods.

Maintaining a metadata field within JavaScript representations of Plaid objects yielded a solution. In each JavaScript representation of a Plaid object, a metadata tree details all current states, their unique names, the members associated with them, and their relationships to each other. The runtime updates these trees as objects are transitioned between states. See Figure 3 for a visual representation of a `Car` object's JavaScript metadata tree.

State change entails a traversal of the current and target trees – in accordance with Plaid semantics [6] – to identify members to be added or removed. With JavaScript's first-class treatment of objects, it is trivial to update members. If `f` is an object, the code `delete f["close"]` removes member `close`. The code `f["open"]=fileOpen` sets the `open` member of `f` to `fileOpen`, whether it is a variable or a function. With member revisions completed, a metadata update completes the process.

Consider the case depicted in Figure 4, which shows the steps entailed in transitioning a `Car` object from the `Braking` to the `NotBraking` state. Let Left be the metadata

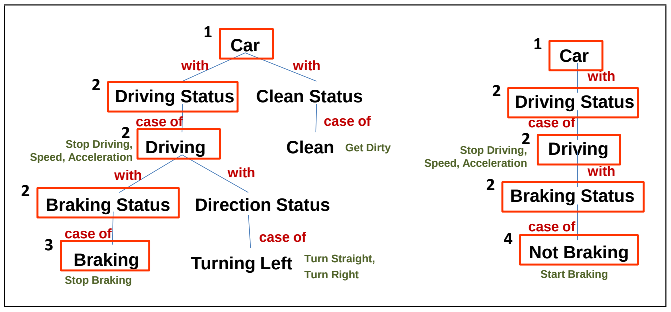

Figure 4. The steps of transitioning a `Car` object from the `Braking` to the `NotBraking` state.

on the left. Let Right be the metadata on the right. Metadata traversal proceeds as follows:

1. Match the unique name of the state at Right's root.
2. Continue matching children as long as all of Right's states appear in Left.
3. Identify an or-state that appears in Left but not Right. These are mutually exclusive. `Braking` must be removed from Left.
4. Identify the state in Right that necessitated the removal of the or-state. `NotBraking` must be added to Left.

The metadata objects depicted in Figure 4 thus provide all the information necessary for state change. Traversing them as in Figure 4 reveals which states must be added and removed. Since the metadata indicates that `Braking` and `NotBraking` each have an associated method, the runtime also identifies that it must add and remove members from the object to which this metadata belongs. The runtime therefore removes `StartBraking` from and adds `StopBraking` to not only the metadata, but also the object that represents the Plaid object.

The essential features of our runtime representation are:

- It takes exactly *one* JavaScript object to represent each Plaid object.
- A Plaid object's one corresponding JavaScript object has all the Plaid object's currently available members, and no other additional members.
- Each Plaid object's corresponding JavaScript object also has a metadata element, which encodes all the information necessary for performing state change on the object.

5. Preliminary Results

To evaluate the performance of this implementation, we translated the Splay and Richards benchmarks from the V8 JavaScript benchmark suite[1] into Plaid. These Plaid benchmarks were compiled to JavaScript, after which the resultant code was timed alongside the original V8 JavaScript ver-

[1] http://v8.googlecode.com/svn/data/benchmarks/v6/run.html

sions, the results of which appear in Tables 1 and 2. These preliminary results — produced with a still non-optimizing compiler — reveal that run time for code compiled from Plaid ranged from 2.6 times the run time of the original JavaScript implementation (for the Splay benchmark) to 8.7 times the run time of the original JavaScript implementation (for the Richards benchmark).

Even with the current non-optimized compilation strategy, the JavaScript compiler far surpasses the performance of the pre-existing Java compiler. Table 3 displays the results of running the two Plaid compilers on the same Plaid benchmark. Execution time for the Java code was 49 times the execution time of the code produced by our JavaScript compiler. As planned optimizations move forward, the performance of the JavaScript compiler should compare even more favorably.

	JavaScript	Plaid to JavaScript
Average Run Time (ms)	760	1950
Standard Deviation	30	70

Table 1. Execution time for the Splay benchmark. A comparison of the original JavaScript program and the same program translated to Plaid and compiled to JavaScript.

	JavaScript	Plaid to JavaScript
Average Run Time (ms)	19	166
Standard Deviation	2	5

Table 2. Execution time for the Richards benchmark. A comparison of the original JavaScript program and the same program translated to Plaid and compiled to JavaScript.

	Plaid to Java	Plaid to JavaScript
Average Run Time (ms)	76,000	1,570
Standard Deviation	2,000	40

Table 3. A comparison of run times for the same Plaid program, executed using the Plaid to Java compiler and the Plaid to JavaScript compiler.

6. Contribution

Substantial work has gone into optimizing compilation for dynamically typed languages, and early Scheme [1] and Self [4] innovations continue to furnish valuable insights. As languages like Python and Ruby meet with steadily greater success, the importance of such work is only growing.

It is in developing state change that Plaid introduces a new challenge for implementation, one that has not yet been addressed in the field. Our implementation offers a novel representation for state at runtime, a representation shaped by the need to facilitate efficient member access. Even in its non-optimized form, our compiler produces code 49 times faster than code produced with the naive implementation scheme suggested by past works on object reclassification.

If this research successfully optimizes state change, it will be the first implementation to do so. This will be an important step in establishing state-based languages' usefulness for practical purposes, and in making state abstraction a viable option for language users and designers.

7. Future Work

There remain many pressing questions on the topic of how to efficiently compile a language that supports state change. This research will go on to investigate further refinements of the JavaScript implementation. To start, we hope to implement several state-related optimizations.

First, before a new Plaid object can be created, the runtime must execute a computationally expensive unique members check, which ensures that no members of the new object will share the same name. While this check is important in instantiating a state that has never yet been used, there may be an advantage to tracking already-instantiated states and eliminating the unique members check from their instantiation process.

Second, some programs frequently perform state change between the same states. For these programs, caching the results of state change may improve performance. If state change is enacted multiple times from a receiver with metadata tree A to a state with metadata tree B, the members to be added and removed are necessarily the same each time the transition is executed. If the runtime determines during state change that the receiver and target trees appear in a cached receiver-target pair, the runtime should not have to traverse the same pair of metadata trees again.

Additional future work will center on implementations for target languages without prototype support — for instance, a traditional object-oriented language like Java. With classes able to inherit members from only a single superclass, how can a Plaid state be created as a composition of two other states? How can states be allowed to transition, and their members with them? While slow solutions come readily to mind, efficient ones do not. These and many other questions will provide fertile ground for continued research.

References

[1] N. Adams, D. Kranz, R. Kelsey, J. Rees, P. Hudak, and J. Philbin, *ORBIT: an optimizing compiler for scheme*, In *Proc. Symposium on Compiler Construction*, 1986.

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

[3] K. Bierhoff, and J. Aldrich, *Lightweight object specication with typestates*, In*Proc. ESEC/FSE*, 2005.

[4] C. Chambers and D. Ungar, *Making pure object-oriented languages practical*, In *Proc. OOPSLA*, 1991.

[5] D. Harel, *Statecharts: A visual formalism for complex systems*, In *Sci. Comput. Program*, 1987.

[6] J. Sunshine, K. Naden, S. Stork, J. Aldrich, and E. Tanter, *First-Class State Change in Plaid*, In *Proc. OOPSLA*, 2011.

[7] D. Ungar, and R. B. Smith, *Self: The power of simplicity*, In *Proc. OOPSLA*, 1987.