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. State-Based Languages

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] [4]. 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 a `File` state. `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 states, the advantages are clear in the context of a more complex state space.

```
1 state File {
2    val filename; }
3 state OpenFile case of File {
4    val filePtr;
5    method read() {}
6    method close() { this <- ClosedFile;} }
7 state ClosedFile case of File {
8    method open() { this <- OpenFile;} }
```

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

2. Implementation

While maintaining state and permitting state change is excellent from the perspective of a programmer, the challenges for implementation are significant. A Java implementation of Plaid has already been created. However, it relies heavily on inefficient reflection. At runtime, every representation of a Plaid object contains a map of members, and each member is itself an object. Method calls require finding the member in the map, and then calling a method on the member object, which is essentially a container for the body of the function. Performance is unsurprisingly slow.

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 at this juncture 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.

An intuitive representation entails maintaining a JavaScript object for each state and substate: one for `File`, another for

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

`OpenFile`, another for `ClosedFile`. Pointers to the states that currently compose the Plaid object would render state change a simple process. 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.

The need for fast method calls and field lookup motivates an alternative approach, one that places members at the level of the JavaScript instance object. That is, any member of a Plaid object must be a member of the JavaScript object that represents it. If JavaScript object `f` represents a `File` in the `OpenFile` state, a call to `read` should produce the code `f.read()` at runtime, rather than trigger a search through `File` and `OpenFile` objects. However, with this accomplished, how can state change be enacted? When one `case of` state is removed and another added, how is the runtime to identify the appropriate modifications?

Maintaining a metadata field within JavaScript representations of Plaid objects yielded a solution. A tree details all current states, the members associated with them, and their relationships to each other. State change entails a traversal of the current and target trees – in accordance with Plaid semantics [4] – to identify members to be altered. 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.

3. 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 versions, the results of which appear in the first two charts of Figure 1. These preliminary results – produced with a still non-optimizing compiler – reveal a range of slowdowns, from a 158.4% increase in Splay running time, to a 768.8% increase in Richards running time.

Even with the current naive compilation strategy, the JavaScript compiler far surpasses the performance of the pre-existing Java compiler, as is evident in the third chart in Figure 1. Our implementation produces code that runs in 2.1% the amount of time required by the compiled Java code, with an average running time of 76,279.4 milliseconds for the Java Richards, and an average running time of 1,566.5 for the JavaScript Richards. As planned optimizations move forward, the performance of the JavaScript compiler should compare even more favorably.

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

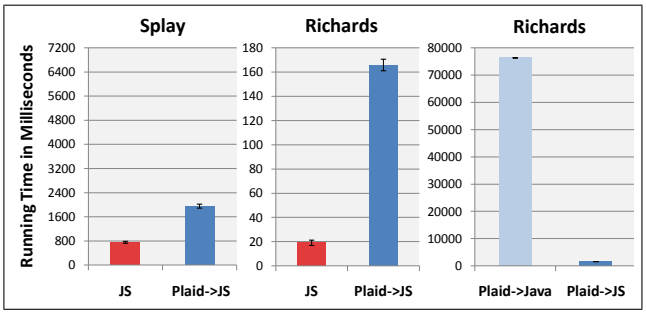

Figure 1. Average running times and the standard deviations appear above. To compare JS to compiled JS, each benchmark was run 500 times in SpiderMonkey. To compare compiled Java to compiled JS, each benchmark was run 100 times within a loop.

4. Contribution

Substantial work has gone into optimizing compilation for dynamically typed languages, and early Scheme [1] and Self [3] 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. 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.

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

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