orphan:

General Type State Notes

Immutability

Using Typestate to control immutability requires recursive immutability propagation (just like sending a value in a message does a recursive deep copy). This brings up interesting questions:

1. should types be able to opt-in or out of Immutabilizability?
2. It seems that 'int' shouldn't be bloated by tracking the possibility of immutabilizability.
3. We can reserve a bit in the object header for reference types to indicate "has become immutable".
4. If a type opts-out of immutabilization (either explicitly or implicitly) then a recursive type derived from it can only be immutabilized if the type is explicitly marked immutable. For example, you could only turn a struct immutable if it contained "const int's"? Or is this really only true for reference types? It seems that the immutability of a value-type element can follow the immutability of the containing object. Array slices need a pointer to the containing object for more than just the refcount it seems.

Typestate + GC + ARC

A random email from Mike Ferris. DVTInvalidation models a type state, one which requires recursive transitive propagation just like immutable does:

"For what it is worth, Xcode 4 has a general DVTInvalidation protocol that many of our objects adopt. This was a hard-won lesson dealing with GC where just because something is ready to be collected does not mean it will be immediately.

We use this to clean up held resources and as a statement of intent that this object is now "done". Many of our objects that conform to this protocol also assert validity in key external entry points to attempt to enforce that once they're invalid, no one should be talking to them.

In a couple cases we have found single-ownership to be insufficient and, in those cases, we do have, essentially, ref-counting of validity. But in the vast majority of cases, there is a single owner who _should_ be controlling the useful lifetime of these objects. And anyone else keeping them alive after that useful lifetime is basically in error (and is in a position to be caught by our validity assertions.)

At some point I am sure we'll be switching to ARC and, as we do, the forcing function that caused us to adopt the DVTInvalidation pattern may fall by the wayside (i.e. the arbitrary latency between ready to be collected and collected). But I doubt we would consider not having the protocol as we do this. It has been useful in many ways to formalize this notion if only because it forces more rigorous consideration of ownership models and gives us a pragmatic way to enforce them.

The one thing that has been a challenge is that adoption of DVTInvalidation is somewhat viral. If you own an object that in invalidate-able, then you pretty much have to be invalidate-able yourself (or have an equivalent guaranteed trigger to be sure you'll eventually invalidate the object)... Over time, more and more of our classes wind up adopting this protocol. I am not sure that's a bad thing, but it has been an observed effect of having this pattern."

Plaid Language notes

http://plaid-lang.org/ aka http://www.cs.cmu.edu/~aldrich/plaid/

This paper uses the hybrid dynamic/static approach I chatted to Ted about (which attaches dynamic tags to values, which the optimizer then tries to remove). This moves the approach from "crazy theory" to "has at least been implemented somewhere once": http://www.cs.cmu.edu/~aldrich/papers/plaid-oopsla11.pdf

It allows typestate changes to change representation. It sounds to me like conjoined discriminated unions + type state.

Cute typestate example: the state transition from egg, to caterpillar, to pupae, to butterfly.

It only allows data types with finite/enumerable typestates.

It defines typestates with syntax that looks like it is defining types:

```
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; }
}
```

Makes it really seem like a discriminated union. The stated reason to do this is to avoid having "null pointers" and other invalid data

around when in a state where it is not valid. It seems that another reasonable approach would be to tag data members as only being valid in some states. Both have tradeoffs. Doing either of them would be a great way to avoid having to declare stuff "optional/?" just because of typestate, and even permits other types that don't have a handy sentinel. It is still useful to define unconditional data, and still useful to allow size-optimization by deriving state from a field ("-1 is a closed file state" - at least if we don't have good integer size bounds, which we do want anyway).

It strikes me that typestate declarations themselves (e.g. a type can be in the "open" or "closed" state) should be independently declared from types and should have the same sort of visibility controls as types. I should be able to declare a protocol/java interface along the lines of:

```
protocol fileproto {
  open(...) closed;
  close(...) opened;
}
```

using "public" closed/opened states. Insert fragility concerns here.

It supports multidimensional typestate, where a class can transition in multiple dimensions without having to manually manage a matrix of states. This seems particularly useful in cases where you have inheritance. A base class may define its own set of states. A derived class will have those states, plus additional dimensions if they wanted. For example, an NSView could be visible or not, while an NSButton derived class could be Normal or Pressed Down, etc.

Generics: "mechanisms like type parameterization need to be duplicated for typestate, so that we can talk not only about a list of files, but also about a list of *open* files".

You should be allowed to declare typestate transitions on "self" any any by-ref arguments/ret values on functions. In Plaid syntax:

```
public void open() [ClosedFile>>OpenFile]
```

should be a precondition that 'self' starts out in the ClosedFile state and a postcondition that it ends up in the OpenFile state. The implementation could be checked against this contract.

Their onward2009 paper contains the usual set of aliasing restrictions and conflation of immutable with something-not-typestate that I come to expect from the field.

Their examples remind me that discriminated unions should be allowed to have a 'base class': data that is common and available across all the slices. Changing to another slice should not change this stuff.

'instate' is the keyword they choose to use for a dynamic state test.