I keep thinking about parallel blocks although I know I probably shouldn't; but so long as I write these notes while rustc builds, everybody wins, right?

Anyhow, pcwalton and dherman yesterday pointed out to me that `const` is not exactly one of the most beloved features of C++: "const-ification" is no fun, and if we're not careful, Rust could walk right down that path. To some extent my reaction is, "Well, something's gotta' give." You can't have modular static race freedom without some way to know what function will write what. But nonetheless they are quite correct.

I think the problem with `const` is that it's not the default, despite the fact that most operations are reads. So perhaps we could make `const` the default for Rust: I think this would fit with the general philosophy of making mutation explicit. In practice what this would mean is that declarations like `x: @T`, `&&x: T`, `x: [T]` would all be declaring a pointer `x` to read-only (i.e., `const`) data. This would be a deep const. A pointer of type `@mut T` would permit modification of the data it points at. (I am not quite sure what to do with `~`; since there is no fear of aliasing, permitting mutation implicitly makes sense to me but it is somewhat inconsistent).

This makes subtle changes in the meaning of existing types. For example, `[T]` no longer means an immutable array but rather what today would be written as `[const T]`. We could add an explicit `imm` qualifier if we wanted to allow for immutable data: it would be usable not only on arrays but also on records and the like, so you could write `[imm T]` or `@imm T`. I won't talk about `imm` further, though; the details of its addition are left as an exercise to the reader (always wanted to write that...).

An important difference between `mut` and the other qualifiers is that `mut` is shallow where the others are deep. For example, given a variable `x: @mut T` where the type `T` is defined as:

```
type T = {
    f: @U, g: @mut U
};
```

`x.f` has type `@U` and `x.g` has type `@mut U`. But if we have a non-`mut` variable `y: @T`, then both `y.f` and `y.g` have type `@U` (the `mut` on the field `g` is lost because `y` is a const pointer).

One final note: in comparison to C, I have only discussed the `mut` qualifer in pointer types. For example, `@mut T` and `[mut T]`. Types like `const int` that you can find in C don't make sense to me: it's a value, it can't be modified, only

overwritten. That seems like a property of the lvalue not the type. But if you want to interpret things in the C way, then you could say that types are mutable by default, but pointers are const by default. So the type `int` is a mutable int (i.e., a variable which can be changed), but the type `@int` is a const pointer to an int. I don't think this is a very important point, personally; it sounds like it's deep but I think it's not.

Ok, this is just a sketch, but it's enough for me to remember what I was thinking later. I am pretty sure this actually hangs together rather nicely (I thought about it a lot on the train and went through various iterations). I am not sure how it would feel to program in, but I suspect it would be very nice. More to come in later posts I'm sure.