I recently implemented a new hashtable module for Rust. This was actually the first piece of code written in Rust which I started from scratch. I ran into some difficulties. Some of these are just things you have to get used to; some might be worth trying to correct. Just not entirely sure which problem falls into which category yet.

Cyclic types

Due to Rust's structural types, types cannot reference themselves. In the hashtable module I was defining, I wanted a linked list of entries. In C I would write something like:

```
struct entry {
    unsigned hash;
    void *key;
    void *value;
    entry *next;
};
```

The Rust equivalent I ended up with is:

```
type entry<copy K, copy V> = {
    hash: uint,
    key: K,
    mutable value: V,
    mutable next: chain<K, V>
};
```

It's great that I can use generic types and specify which fields are mutable and which are not. But, for the `next` pointer, I would have liked to simply say `option::t<@entry<K,V>>` (actually, I would *like* to say `option<@entry<K,V>>`, but that's another story). However, that would create a cyclic type, so I cannot, and instead I must introduce a tag very similar to option, `chain`:

```
tag chain<copy K, copy V> {
    present(@entry<K, V>);
    absent;
}
```

I suppose that another option would be to use a type like `list<@entry<K,V>>`, however the `std::list` implements standard, immutable functional lists, which aren't as efficient when removing items from the list.

The mutable keyword is too long and I have to write it too often

It should be `mut` or something. It doesn't feel very Rust-like. And to make it worse I have to repeatedly tell the type checker that a field is mutable each time I define an instance of a record.

Defining records vs tags

The syntax for defining a record vs defining a tag is different in some small

ways. Compare the record definition for `entry` we saw above:

```
type entry<copy K, copy V> = {
    hash: uint,
    key: K,
    mutable value: V,
    mutable next: chain<K, V>
};
```

to the tag definition for `chain`:

```
tag chain<copy K, copy V> {
    present(@entry<K, V>);
    absent;
}
```

The record definition separates the fields with `,`, has an `=` and a trailing `;`. The tag separates the variants with `;` and has no trailing `;`. I know why these things are the way they are (the record type definition is really declaring a type alias, etc), but I find it surprisingly hard to remember which goes where.

Alias analysis

Rust includes an alias analysis that is intended to prohibit you from using mutation to invalidate outstanding references that you have. The idea is sound but the rules that are enforced are a bit confusing. It felt like the compiler was arbitrary forcing me to introduce random local variables at miscellaneous points in the code, even though I know there is a logical set of rules behind it. Sometimes this was inconvenient, such as when I wanted to invoke a function in the `when` clause of an `alt` arm; that particular case eventually resulted in me breaking apart a function into several functions because the code got too messy when I had to introduce the temporaries.

Nested functions do not inherit scope

I found it pretty annoying that I had to constantly repeat parameters—particularly type parameters—because there isn't really a good way of having many functions that share the same common set of upvars.

Basically, I want to write something like:

```
class hashmap<copy K, copy V> {
    priv type entry = { ... };
    priv tag chain { ... }
    priv let buckets: [mutable chain];
    priv fn search_tbl(k: K, h: uint) { ... }
    fn get(k: K) -> V { ... }
    fn remove(k: K) -> option::t<V> { ... }
}
```

Instead I have to write:

```
type entry<copy K, copy V> = { ... };
tag chain<copy K, copy V> { ... }
```

```
type tbl<copy K, copy V> = {
    buckets: [mutable chain<K,V>],
    ...
}
fn search_tbl<copy K, copy V>(t: @tbl<K,V>, k: K, h: uint) { ... }
fn get<copy K, copy V>(t: @tbl<K,V>, k: K) -> V{ ... }
fn remove<copy K, copy V>(t: @tbl<K,V>, k: K) -> option::t<V> { ... }
```

Notice anything different? In the first version, I only had to define the type variables once. Furthermore, I didn't have to keep listing the table everywhere. And, because the types were defined insde of the class scope, I didn't have to repeat the `K` and `V` each time I referenced them (note: this goes beyond the current OOP proposals, which don't permit nested types). I find the second version fairly hard to read because there is so much redundant information. And I didn't even define the object yet! (Because objects in Rust have no self pointer, they can't invoke their own methods, so you basically always end up defining a module of functions and a type that holds the object's data, then glueing it all together with an object at the end)

Update: Self calls are possible with objects in Rust, but that is the only thing one can do with the `self` pointer. Nonetheless one still defines sets of functions that operate over shared data in order to achieve encapsulation.

References

One thing I am not terribly fond of is passing by-reference. Right now, if you want to accept a pointer argument but you don't want to specify precisely where that pointer is allocated, you can do it via an immutable reference. This is in fact the default mode for most arguments (except those of scalar and boxed type, I think). This means that in a function like:

```
fn get<K,V>(t: tbl<K,V>, k: K) { ... }
```

The parameter `t` is actually passed by immutable reference. Now, suppose I want to call `get()`:

```
fn call_get(t: @tbl<uint,uint>) {
    get(*t, 3);
}
```

Here, I had to dereference `t`, giving me the type `tbl<uint,uint>`. What is surprising, though, is that even though I have dereferenced `t`, it is still passed being passed by reference. This is basically the same as in C++. I didn't like it then and I don't really care for it now. On the caller side, it looks like I made a copy of `t` and passed it in by value. But in fact the callee is aliasing `t`, so if the callee changes the mutable fields of `t`, it will affect the caller's copy too. Basically I found that I really had to think hard and examine each method pair

to determine whether a given value would be passed by pointer or by value. This seems like something that should be effortless to decide.

And yet...

Lest I be perceived as complaining too much, I had a good time coding up my little hashtable. Rust does a pretty damn good job of giving you control without requiring you to worry over every last thing. Basically, programming in Rust captures a lot of the good parts of coding in C or C++, without some of the real headaches. I think we've got things to improve but the basics feel right to me. Having only recently joined the project, I am amazed how well it works: the infrastructure feels surprisingly mature. It's definitely good to be a member of a good team!