I am trying to mop up some of the remaining work for regions now. One of the big remaining areas is dealing with function and iface types. This proposal is certainly influenced by my previous proposals. However, we have backed away from the idea of dynamically-sized types for vectors and so I will do the same here.

## The design

My current design includes the following kinds of function types (written as I expect them to commonly be written; some details are omitted):

- `&fn(S) -> T`
- `@fn(S) -> T`
- `~fn(S) -> T`

Similar types apply to iface types (using the iface `to_str` as an example):

- `&to_str`
- `@to_str`
- `~to_str`

These types work like their corresponding pointer types, so a value of type `@fn(S) -> T` or `~fn(S) -> T` can be "borrowed" into a `&fn(S) -> T`. (Today, there is a subtyping relationship)

Similarly, the explicit syntax for creating closures will become `@fn(s: S) -> T { ... }` and `~fn(s: S) -> T { ... }`. I will make it legal to omit the argument and return types and infer them from context. The closures produced from such expressions are always copying closures, in that they copy or move values from their environment when created.

The sugared notation `|s| expr` as well as the `do` and `for` loop notation will use inference to decide which of the three kinds of closures is produced. If a sugared closure resolves to a `&fn`, then a "by-reference" closure will be produced. That is, one which can modify the values in its enclosing environment. Otherwise, a copying closure is created. (As today)

## Borrowed closures

Borrowed closures in their most explicit form are written `&r/fn(S) -> T`. They are like `fn()` today but they are associated with a region `r` (the notation for region pointers is changed from `&r.T` to `&r/T`, for those of you following along). The region type system will ensure they do not escape, so we can lift the various restrictions about using borrowed closures only within parameters and so forth.

This will require some tweaks to various analyses (notably borrowck) that take advantage of the limited possibilities of the current by-reference closures. I don't think this will be terribly difficult.

Including an explicit region also *enables* a lot of very useful things. Closures which reference their environment can be stored in other data structures. Further, we should be able to make use of the region bound to allow borrowed closures to deinitialize variables in their environment—see below for more details.

### Shared closures

Shared closures, written like `@fn(S) -> T`, are exactly like the `fn@(S) -> T` that we use today. They consist of a shared, boxed environment which may close over arbitrary data. They cannot close over borrowed pointers.

### Unique closures

Unique closures in their most explicit form are written like `~fn:K(S) -> T`, where `K` is a set of bounds like those found on type parameters. If `K` is omitted it defaults to `send`. They cannot close over borrowed pointers and, furthermore, all data that they do close over must conform to the given bound. I expect it will rarely be necessary to write an explicit bound, but it may be useful sometimes to write `~fn:send copy(S) -> T`, to indicate that the data that is closed over is not only sendable but also copyable. Unlike all other closure types, unique closures are themselves sendable provided that `send` is listed among the bounds `K`.

### Representation, bare functions

The representation of function pointers will remain a pair `(code, env)` of pointers. This is not as pure as my prior proposals but has some advantages, not the least of which being that it will require less code churn to achieve. It should also make bare functions more efficiently represented.

The bare function type goes away entirely, to be replaced with an inference technique similar to what Lindsey did with integer literal inference.

### Preventing capture

There is one annoying complication. As they have no region bounds, we need to prevent the `@fn()` and `~fn()` types from closing over borrowed pointers. I plan to do this by introducing a new kind `owned` which means "copyable and does not contain borrowed pointers". Functions can only close over owned data. The reason that a kind is necessary is due to the possibility of functions

like the following:

```
iface foo { fn foo(); }
fn generator<A: copy foo>(a: A) -> @fn() {
    @fn() { a.foo() }
}
```

This function closes over `a`. So now if we invoked `generator()` with `A` bound to a type like `&int`, it would close over region data. Any time the returned closure is used, it would make use of that region data. Because the region does not appear in the type of the closure (`@fn()`), the type system would be powerless to prevent this borrowed pointer from leaking out beyond its lifetime. Bad.

In my current design, the function above would be illegal. One would have to write:

```
iface foo { fn foo(); }
fn generator<A: owned foo>(a: A) -> @fn() {
    @fn() { a.foo() }
}
```

Note that the bound `owned` replaced `copy`. Without this bound it would not be legal to copy `a` into the closure. This bound also prevents `generator()` from being called with a borrowed pointer.

This is actually a simplified approach. Another option might be to add region bounds on all `fn` types. We could then allow `fn` types to close over borrowed pointers. I hestitate to do this though because I think that the region bounds on `@fn` and `~fn` would want different defaults than all other region references. For example, all region references appearing in a function signature default to a region parameter, including those that appear within shared boxes and so forth (this will not be a common occurrence, I think, but of course it can happen). So if I write:

```
fn some_func(x: &int, y: @&int) { ... }
```

then both `x` and `*y` have the same lifetime (the default lifetime parameter). If we were consistent, then, it would mean that a function like this:

```
fn some_func(x: &int, f: @fn(int)) { ... }
```

would imply that the lifetime of `x` and the region bound of `f` would both be the same as well (the default lifetime parameter). But this is likely not what you wanted; probably you planned to store `f` into a data structure or something and actually wanted a region bound of `&static` (that is, does not close over region data).

Basically, the difference between an `@fn` type and other shared types with regard to regions is that we never know what an `@fn` closes over. With a

normal shared type, we only add region bounds when regions are actually used: but for `@fn` we'd have to choose a different default.

Maybe this is not so bad, but even if we added region bounds to `@fn`, we'd *still* need the `owned` kind, or else we'd need to be able to add region bounds to type variables. This all seems like layers and layers of complexity that won't be necessary, as a `@fn` type that closes over borrowed pointers isn't all that useful (there are places I can imagine it being useful, though).

Anyway this part is actually easily tweaked if it seems more expressiveness is warranted.

## Goals that are achieved by this proposal

**Stack-allocated and sendable iface instances.** I've wanted this for a long time.

**More flexibility with stack-allocated closures.** The current requirement that all closures which go into data structures be heap-allocated is sometimes a real drag. It means that things like the visitor pattern we implement in rustc can't make use of borrowed pointers—though this would also be ameliorated with traits, I suppose. But anyhow.

**Prefix-sigils.** Postfix sigils like `fn@` are so Rust 0.2.

**Support for sendable, copyable unique closures.** Now that `send` no longer implies `copy`, we need to distinguish these things.

**Fewer closure types.** We currently have way too many (5, I think). With this proposal there are still three, but they look and feel like one. Mostly this is achieved through better inference techniques for native functions and eliminating the "any closure" type (which, admittedly, I introduced).

## Goals that *might* be achieved by this proposal

**Moving within closures.** Actually I only *think* this can be done safely. This is a fairly complex topic. The basic goal is to be able to write something like this `fold()` operation (here I am using some non-implemented things, like unary move, in place of the current `<-` and last use):

```
fn fold<A,B>(iter: &iterable<A>, b0: B, op: &fn(B, &A) -> B) -> B {
    let b = move b0;
    for iter.each |a| {
        b = op(move b, a);
    }
    ret b;
}
```

The interesting part is that we move the intermediate value `b` out of the stack

frame in the call to `op()` and then replace it with a new value before the next iteration of the loop. Note that the loop body here is actually a closure.

This may seem trivially safe, but given the possibility of recursion it is not so. We need to prove that the closure never recurses while some upvars are uninitialized. I *think* regions give us the tools to do that: we can see that the region of `op` is bigger than the method body itself (it's a region parameter, in fact). This implies that `op()` cannot reach the closure which is executing, which is defined with a tighter region. However, there are some complications I haven't thought through—for example, what if we provided the closure as an argument to `op()`? This will turn out being quite complex in the end I fear. Maybe there is a better way.

### Goals that are *not* achieved by this proposal

**One-shot closures.** We still have no support for one-shot closures. It is becoming increasingly clear that these would be helpful: it is common to want to move some value into the closure and then move it out again, and the current unique closure design does not permit this. You can work around it with a series of painful swaps and calls to `option::unwrap()`, but it's inefficient and inelegant.

pcwalton ran an idea by me which basically converted today's `fn~` into a one-shot closure: if you wanted to call the closure mutiple times, you had to copy it. This has a certain appeal but it doesn't work with the system as it is today nor the system I proposed just now, because `fn~` (at least today) can be upcast to `fn`. We'd have to make `fn~` not borrowable.

I was initially enthusiastic about this idea but am now less so. It seems to be a strike against orthogonality: `~fn()` types would behave differently from all other closure types. I'd rather give them a different name, like procedures or even `~fn1()`, to make the distinction more clear. Not sure what is best here.