Fn Types in Rust, Take 3

As some of you may recall, the fate of function types in Rust has been somewhat uncertain. It all began when I realized that we could not soundly permit closures to recurse, which implies that closures must be tracked in a linear fashion. This caused interactions with the prior plans we had for dynamically sized types, and led to several alternative proposals. The most radical involved keeping only one closure type for borrowed closures and then using macros and object types to represent all other use cases.

In a recent discussion, we decided that this plan might be too radical; not because it was inexpressive, but rather because usability might suffer. Therefore our current plan is to reorganize fn types into two groups, which I will call *closures* and *procedures*. These are in addition to "bare function" types, which describe a function with no environment at all. The full set of "callable" types would therefore become as follows:

```
fn(T, U) -> R      // Function (no environment)
|T, U| -> R        // Closures (borrowed environment)
proc(T, U) -> R    // Procedures (call once, owns its environment)
```

The type `|T, U| -> R` is called a *closure*. A closure is a function that has full access to the local variables from the stack frame that created it. The closure can read and modify those variables freely. Closures are cheap to create because they are allocated on the stack; the type system guarantees that the closure will not outlive the stack frame that created it.

A *procedure* `proc(T, U) -> R` is intended to be the means for specifying task bodies. Procedures are similar to closures in that they bundle up a function with an environment containing values taken from the creating environment, but they also have two crucial differences. First, a procedure can only be run once (this reflects the fact that tasks only execute once). Also, procedures allocate their environment on the heap, rather than using the parent's stack frame. This means that the lifetime of a procedure is independent from the stack frame that created it – the procedure can be returned or sent between tasks. (For what it's worth, I prefer the name `thunk` to `proc`, but I seem to be the only one that does.)

Bare functions and ABIs

The simplest form of callable is just a "bare function", which is a function

pointer with no environment at all. Such functions can be represented as a single pointer at runtime. All top-level function items in Rust are bare functions, as are external functions found in C libraries.

The full type of a bare fn includes two optional prefixes that I did not mention:

```
<'a, ..., 'z> extern "ABI" fn(T) -> U
```

- `'a ... 'z` are *named lifetime parameters* that may appear in the argument or return type. While it is not so unusual to use named lifetime parameters in function declarations, it is *very* rare to see them in function *types*. The current codebase contains no instances of function or closure types that specify named lifetime parameters, except in tests.
- The `extern "ABI"` clause specifies an ABI. If the ABI is omitted, then it defaults to `Rust`, and hence `fn(T) -> U` is the type of a normal fn item.

As an example, consider this code to compute fibonacci:

```
fn fibonacci(n: uint) -> uint {
    match n {
        0 => 0,
        1 => 1,
        _ => fibonacci(n-1) + fibonacci(n-2)
    }
}
```

The type of `fibonacci` is `fn(uint) -> uint`.

As another example, consider this code to return a reference to a field of an input struct:

```
struct Point { x: int, y: int }
fn x_coord<'a>(p: &'a Point) -> &'a int {
    &p.x
}
```

The type of `x_coord` is `<'a> fn(&'a Point) -> &'a int`. You may wonder why the `<'a>` moved from after the `fn` keyword to before; the reason is both for consistency with other types like closures and possibly in the future with traits. See the “Future Directions” section below for a fuller discussion.

Examples of using closures

The most common fn type would be a closure. Closures are used for callbacks that will always occur within the lifetime of the current function. Because closures are always linked to the current stack frame, they are able to read and

manipulate local variables for that stack frame. Here is a silly example where a function `call_twice` is invoked with a closure that increments the local variable `count`:

```
let mut count = 0;
call_twice(|| count += 1);
assert_eq!(count, 2);
```

The definition of `call_twice` would look like:

```
fn call_twice(f: ||) {
    f();
    f();
}
```

Here the closure type `||` indicates a closure with no arguments that returns the unit type `()`. A more realistic example would be `map`:

```
fn map<T,U>(slice: &[T], f: |&T| -> U) {
    let mut result = ~[];
    for v in slice.iter() {
        result.push(f(&slice[i]));
    }
    result
}
```

At runtime, a closure would consist of two pointers, one to the function and one for the environment. The environment for a closure would always be allocated on the stack and would be populated with pointers to the borrowed local variables. Thus closures would always be associated with a lifetime representing the stack frame in which the closure was allocated.

Advanced closure types: named lifetime parameters and environment bounds

The full closure type in all its detail would be written

```
<'a, ..., 'z> |T1, ..., TN|:K -> R
```

In addition to the argument types (`T1...TN`) and the return type, there are two additional buts of information here: the named lifetime parameters `<'a, ..., 'z>` and the closure bounds `K`:

- Named lifetime parameters are used to tie together the lifetime of an input and the lifetime of the return value. As stated before, this syntax should be rarely needed.

- The closure bounds `K` describe the closed-over variables in the environment. They consist of a lifetime `'r` and a set of optional built-in traits (`Send` and `Freeze`, to start, but possibly `Isolate` and `Share` and others in the future). It would be very unusual to specify the bounds manually; if omitted, the bounds would default to a fresh lifetime and no traits.
  The two cases where bounds would be specified are (1) placing closures into a struct, where all lifetimes must be explicitly named and (2) specifying data-parallel APIs. In the first case, a struct definition that contains a closure, you would expect to write something like the following:

```
struct ClosureAndArg<'a> {
    closure: |int|:'a -> int,
    arg: int,
}
```

  This would indicate a struct that contains two fields, `closure` and `arg`, where the lifetime of `closure` is `'a`. The second use case, data-parallel APIs, remains a bit speculative, but one might imagine writing types like `|uint|:Share -> uint` to indicate a closure that only closes over data that can safely be shared between multiple parallel worker threads.

Closures vs RAII

Closures are best used for functions that may be invoked more than once. If you know that a closure will be called exactly once, it is better to use RAII for this pattern. This is somewhat limited by the current rules on destructors, which were described and motivated in this previous blog post, but we have plans to generalize those (to be described in an upcoming blog post). Some of our existing code that uses closures will want to be converted. Basically any place where you have the pattern of a function that does some stuff, calls a closure, and then does some more stuff. I'll try to write more on this topic in another blog post.

Procedures and do syntax

The main use case for procedures is for bundling up a function and the data that it touches into a convenient unit that can be shipped to another task and executed there. More generally, where a closure represents a callback that always occurs within the current frame, a procedure represents a callback that that will be invoked in some other context. Because procedures cannot rely on the enclosing frame to exist, they capture any variables that they access *by*

value, meaning that the values of those variables are copied/moved into the procedure as appropriate.
A procedure would be created using the `proc` keyword. Here is a simple example of starting a new task:

```
var (channel, port) = comm::stream();
task::start(proc {
    loop {
        // Here we send values across `channel`, which was
        // captured from the environment.
        channel.send(produce_next_value());
    }
});
loop {
    consume_next_value(port.receive());
}
```

The form `proc { ... }` creates a procedure of type `proc()` – in other words, a procedure with no arguments and a unit return type (the return type here would be inferred).
One could also write `proc(a, b) { ... }` to create a procedure that takes two arguments. The resulting type would be `proc(A,B)`, where `A` and `B` are the inferred types for the parameters (`proc(a: A, b: B) -> () { ... }` would be the fully explicit form).
The current keyword `do` would be used as syntactic sugar for invoking a function and providing a `proc` as the final argument. Therefore, `task::start(proc { ... })` could also be written:

```
do task::start {
    ...
}
```

Presumably `do foo |a, b| { ... }` could be used as today for the case where the proc expects arguments.
Any values used in the proc body are implicitly captured by value, just as today. This means that their current value is copied from the environment; depending on the type of the value, this may mean that the creator of the proc loses access to the value (for example, if the value has a destructor, it will be moved into the proc and no longer accessible from the creator).

Like closures, procedure types may have bounds or named lifetime parameters, so the full procedure type would include optional named lifetime parameters and bounds:

```
<'a, ..., 'z> proc(T1, T2):K -> U
```

- Named lifetime parameters are used to tie together the lifetime of an input and the lifetime of the return value. As stated before, this syntax should be rarely needed.
- The bounds `K`. By default, the bounds on a procedure are `'static+Send`, meaning that the procedure only closes over sendable data. This is the bounds you want if you plan to send the procedure to another task, which will almost always be the case. But if for some reason we wanted weaker bounds, one might write e.g. `proc():Freeze`, to indicate a procedure that only closes over freezable (but not necessarily sendable) data. Of course such a procedure could not itself be sent between tasks.

Future directions

Eventually we would like to eliminate closures and procedures as distinct types and instead rely on the trait system instead; this cannot be done today because the current trait system is lacking a few necessary features (bound lifetimes, “varargs”-like type parameters). We believe this can be done in a backwards compatible way by making the syntax proposed here into syntactic sugar.

At a high-level, the idea is that we would define a trait for callable things like closures or bare functions (procedures would require a slightly different trait, see below). This trait would look something like:

```
trait Fn<A,R> {
    fn call(&mut self, args: A) -> R;
}
```

Now you could represent a closure type like `|int, uint| -> float` as the object type `&mut Fn<(int, uint), float>` (in fact, the closure type would be syntactic sugar for the object type).

What is nice about this approach is that you can write generic functions that are parameterized over closure types in order to generate a specialized variant. The specialized variant will have static linkage and a few other advantages. For example, `map` as we saw before might be written:

```
fn map<T,U,F:Fn<&T,U>>(slice: &[T], f: F) {
    let mut result = ~[];
```

```
    for v in slice.iter() {
        result.push(f.call(&slice[i]));
    }
    result
}
```

Some complications

There are some limitations in the trait system that prevents this from working today. One is that traits today must have a fixed number of type parameters, so we can only express functions of a single argument. You can workaround this either by expressing multiple arguments as tuples (inefficient), by having multiple `Fn` traits like `Fn1`, `Fn2`, etc (hacky, but similar to Scala), or by having "variable arity" type parameters where the single parameter `A` expands to many "inline" types (complicated, but similar to C++).

Another problem is the matter of lifetime parameters. Today, a fn type like `|&T| -> U` means a closure that takes a pointer to `T` with some caller-specified lifetime. The point is that the lifetime of the argument will potentially be different each time the closure is called. Unfortunately trait and object types do not currently support bound lifetimes. It may help to illustrate what I mean using named liftime parameters: the type `|&T| -> U` is equivalent to `<'a> |&'a T| -> U`. The trait equivalent would be `<'a> Fn<&'a T, U>` except that we don't support a syntax like that currently (this example is also another reason why lifetime binders go before the `fn` keyword: if we were to write `fn<'a>`, I at least would expect to write `Fn<'a>` – but that would be supplying `'a` as a value to `Fn`).

Working out this example pointed out another minor complication. I had previously assumed that the call notation `f(&slice[i])` would become an overloaded operator controlled by the trait `Fn`, but that excludes procs, because they require a distinct trait (`Proc`). Perhaps the answer is just not to allow `call` notation except on `||` or `proc` types, but that makes them something more than syntactic sugar, though not a lot more.