I have been thinking about unique closures, one of the last blocker items for the Rust 0.1 release. The original idea of a unique closure was that it should be a closure that can only access uniquely owned state, and which can therefore be sent to other tasks. However, I've come to have my doubts about this idea. In particular, a unique closure, like any closure, is a function, and can therefore be invoked many times: but this imposes some limits on what such a closure can safely do.

As an example, a closure that brings in a unique pointer cannot move it anywhere else (it could safely*swap* the pointer): if it did do the move, what would happen the next time the closure was called? You can't move the same variable twice.

Procedures

Therefore I've been thining about introducing the notion of *procedures*. A procedure is basically a one-shot closure. The user would create a coroutine using the keyword `proc`:

```
let p = ~proc(f: uint) {
    log_err f;
};
```

The result of this `proc` expression would be `proc::t<uint>`. As you can see, procedures do not have function type and cannot be called like other functions. They can accept arguments whose value will be provided when the function is called: if there are multiple arguments, they are packaged up into a tuple type. If no arguments are specified, the `()` may be omitted altogether (e.g., just `proc {...}`).

> *As an aside: this approach cannot accomodate modes. But I want to make those less vital.*

Invoking a procedure is done by the `proc::call()` function:

```
proc_call(p, 3u);
```

`proc::call()` takes the the procedure and the argument to the procedure and executes it. The definition of `proc::call()` is as follows:

```
fn proc_call<A>(-p: ~proc::t<A>, a: A)
```

Capture clauses in procedures

By default, procedures can only access upvars from the environment that are implicitly copyable. However, using a capture clause, procedures can copy or move other upvars into scope:

```
fn xfer(x: ~T) {
    let p = ~proc[move x]() {
        // x is accessible here.  It can be
        // moved, copied, whatever.
    }
    // x is inaccessible here!
}
```

Procedures as building blocks

One of the design goals for unique closures was to support task libraries. I believe procedures can satisfy this requirement. For example, suppose we wanted to have a task library that implicitly created a channel of type any for every task:

```
fn spawn_chan_task<A>(-p: ~proc::t<(chan<any>)>) {
    let c: chan::t<any> = /* create channel */;
    spawn proc[move p, c] {
        proc::call(p, c);
    };
}
```

That example is surely wrong in every particular as I am not that familiar with the chan API and don’t feel like looking it up. But hopefully you get the idea: you can create a channel and then invoke the procedure, passing it the channel. Using `spawn_chan_task()` would then be something like:

```
spawn_chan_task proc(c: chan<any>) {
    ...
}
```

Convenient syntax

The proc keyword should allow procedures to be passed as argument much like blocks, in other words without parentheses. There is a bit of confusion because `proc` doubles as a module. I’m OK with that but others may not be.

Generalizing procedures: coroutines

Procedures are actually a simplification of full-fledged *coroutines* (a.k.a., *iterators*, *delimited continuations*, and many other names). The main difference is that, whereas a procedure executes once, a coroutine can *yield* an arbitrary number of times back to its creator. I originally wanted to implement full-fledged coroutines. The design is similar: the `proc` keyword is replaced by `coro`. Many other aspects remain the same. I am still more-or-less in favor of this idea, but there are certain complications: primarily, when can yields occur and how do you type them?

The simplest approach is probably to have the yield occur only in the coroutine body. So you can then write `put x` (to use the traditional Rust keyword). If you want to delegate to another coroutine, you can write `coro::put_all()` which expects a coroutine as argument.

The cool thing about coroutines is that they separate out *context switching* from *parallelism*. They would provide us with a path to implement iterators, a feature we used to have, and also simplify the inversion of control problem that infects things like lexers and parsers.
The downside is that they are a bit more verbose than procedures, because they need a type like `coro::t<A,Y>` where `A` is the argument type and `Y` is the yield type. They are also more complex to implement, and they may not be fast enough to *really* use for iteration anyhow (though they ought to be; switching stacks is pretty cheap).

The ugly stuff

There are some challenges. To be safely sent to another task, procedures need to only close over sendable state. But they are kind of useful in their own right—particularly if we do full-fledged coroutines. Also, one thing that might be useful to close over are functions if not closures. Not to mention that Rust has too many function types as is. So I want to think about this a bit and see what can be done to simplify things.