One of the last remaining tasks for Rust 0.1 is to find a way to address issues #1128 and #1038. The key problem is that, right now, we can only spawn a task with a *bare function*, which is to say a function that carries no closure or environment. Due to the way that Rust is implemented, this even excludes *generic functions*. I have been wanting to lift this restriction but have been stymied by trying to make it accessible.

My current thought is to divide the set of callable things in Rust into four categories; but only three that I think users will really interact with. These three primary categories mirror Rust's type system, which divides data amongst the *exchange heap*, *task-local heap* (confusingly called *shared*), and the *stack*. The three function types are:

- `fn[send](A*) -> R`, pronounced *sendable closure*
- `fn(A*) -> R` pronounced *task-local closure*
- `block(A*) -> R` pronounced *blocks* (or *stack closure*)

The final function type is `fn[bare](A*)->R`, pronounced *bare function*. A bare function is a top-level function that closes over no state; it can be silently converted to any other function type. This is why I say users won't typically have to interact with them, except perhaps for some specialized purposes. It's possible they can go away entirely, actually.

Sendable closure

A sendable function `fn[send](A*)->R` is a function that can be sent to another task. As a consequence, it can only close over sendable data. Like all things that are to be sent, a sendable function is best manipulated by a unique pointer: `~fn[send](A*)->R`, which allows it to be sent with a `move` operation. Otherwise the closure must be copied; in the initial version, in fact, I think that functions will not be copyable, and hence the only way to send a function will be by unique pointer.

A sendable function can be defined inline using the keyword `fn` as part of an expression:

```
fn foo() {
    let c = ~fn[send](a: uint) {
        ...
    }
}
```

One can also specify capture clauses to move state into the closure. Eventually, if the no implicit copiesproposal is accepted, capture clauses will also be used

to copy non-trivial values into the closure (whereas today that will just happen implicitly):

```
fn foo(-x: ~T) {
    let c = ~fn[send; move x](a: uint) {
        ...
    }
}
```

The type of `c` remains the same but now `x` can be used within the body of `c` (but not within the remainder of `foo()`.

Task-local functions

Task-local functions are created the same way, but without the `[]` (or at least without the word `send`). One thing that will become possible is to specify a capture clause on a normal functon as well:

```
fn bar(-x: T) {
    let c = fn[move x](a: uint) {
        ...
    }
}
```

Here the type of `c` is `fn(uint)`. The body of `c` can access `x` without any copies having occurred.

Marijn's example

The primary goal of this proposal is to make it possible to write nice wrappers around the (rather bare bones) task system that exists today. Here is Marijn's example from bug #1128, written for this proposal:

```
use std;
import std::{task, comm};

type connected_fn<To, From> = fn[send](comm::port<To>, comm::chan<From>);
type ha<To, From> = (comm::chan<From>,
                     comm::chan<comm::chan<To>>,
                     ~connected_fn<To, From>);

fn connect_helper<send To, send From>(d: ha<To,From>) {
    let (out, send_in, f) = d;
    let in = comm::port::<To>();
    comm::send(send_in, comm::chan(in));
    f(in, out);
}

fn spawn_connected<send To, send From>(f: connected_fn<To, From>)
    -> (task::task, comm::port<From>, comm::chan<To>) {
    let in = comm::port::<From>();
    let get_out = comm::port::<comm::chan<To>>();
    let task = task::spawn(
        (comm::chan(in), comm::chan(get_out), f),
        bind[send] connect_helper::<To,From>(_));
    let out_chan = comm::recv(get_out);
```

```
    ret (task, in, out_chan);
}

tag msg { msg(int); quit; }

fn worker(in: comm::port<msg>, out: comm::chan<int>) {
    while true {
        alt comm::recv(in) {
          msg(i) { comm::send(out, i * 10); }
          quit. { break; }
        }
    }
}

fn main() {
    let (task, in, out) = spawn_connected(worker);
    comm::send(out, msg(20));
    log_err comm::recv(in);
    comm::send(out, quit);
}
```

Bare functions

It is unclear to me whether there is *ever* a need for bare function types. The main thing is that a bare function can be adapted silently to the other types. So it may be that we need a way to express bare function types just so that we can use it in the type checker. That will become clearer as I begin to implement.