One more thing we need: in pure linear logic the only thing we pass is channels; we encode data by having processes that represent data. But for practical programming, we'd like to be able to pass data: arbitrary data in some sense. So what is a logical construct that corresponds to being able to pass arbitrary data?

The right way to do this is quantification.

a contextual monad

$$\tau ::= \tau_1 \to \tau_2 \mid \cdots \mid \{A_1 \cdots A_n \vdash A\}$$

$$A ::= A_1 \otimes A_2 \mid 1 \mid A_1 \multimap B \mid \&\{l_i : A_i\} \mid \oplus\{l_i : A_i\}$$

generalized to finitely many w/ labels

recursive types

$$\mid \alpha \mid \mu\alpha.A \mid \forall n{:}\tau.A \mid \exists n{:}\tau.A \mid \tau \supset A \mid \tau \wedge A$$

$$M ::= \lambda x.M \mid M_1\, M_2 \mid \cdots \mid \{c \leftarrow P \leftarrow c_1, \ldots c_n\}$$

non-dependent versions

syntax for forming monad expressions

As usual, we'll explicate the behavior of quantifiers using their left and right rules.

$$\frac{\overbrace{\Psi, n{:}\tau}^{\text{persistent}} ; \overset{\text{ephemeral}}{\Delta} \vdash P_n^x :: x{:}A}{\Psi ; \Delta \vdash (n \leftarrow \mathsf{recv}\ x;\ P_n^x) :: x{:}\forall n{:}\tau.A}\ \forall R^n$$

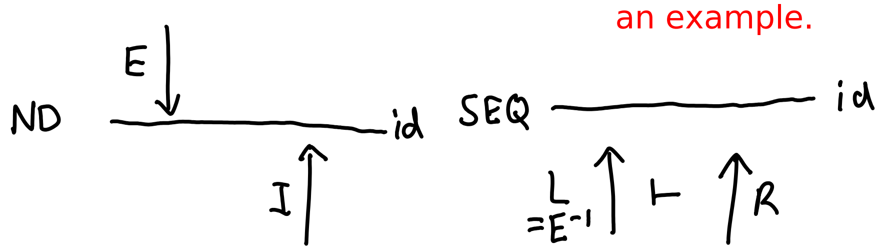
n shows up in P

$$\frac{\overbrace{\Psi \vdash N{:}\tau}^{\text{ordinary type theory}} \qquad \Psi ; \Delta, x{::}[N/n]A \vdash Q_x^z :: z{:}C}{\Psi ; \Delta, x{::}\forall n{:}\tau.A \vdash (\mathsf{send}\ x\ N;\ Q_x^z) :: z{:}C}\ \forall L^n$$

Intuitively, forall "inputs a value of type t", and exists "outputs a value of type t."

There is a general pattern to left and right rules in sequent calculus, where both rules move towards the identity proof. In natural deduction, the elimination rule moves in the other direction; thus elimination and left rule are "inverses" in some sense.

This would be clearer with an example.



At this point, we can right processes which compute functional values, but we cannot write functions with compute processes. So the idea is to use a monad to let us embed processes into the functional language. In fact, we will need a contextual monad which indicates which processes it uses. Let us state the language for processes, recast in monadic language:

$$
\begin{aligned}
P ::= \; & c \leftarrow a & \text{(id)} \\
\mid \; & x \leftarrow M \leftarrow c_1 \cdots c_n \, ; \, Q_x & \text{(comp)} \quad \text{also known as bind} \\
\mid \; & \cdots
\end{aligned}
$$

This is classical monads, except that the variables inside the monad have some extra binding structure.

Now for some examples:

$$
\begin{aligned}
\text{natstream} &= \exists n : \text{nat} .\ \text{natstream} \\
&= \text{nat} \wedge \text{natstream} \\
&= \mu\alpha .\ \text{nat} \wedge \alpha
\end{aligned}
$$

$\text{nats} : \text{nat} \rightarrow \{ \vdash \text{natstream} \}$

$c \leftarrow \text{nats}\ n =$

$\{ \text{send}\ c\ n ;$ — send the current number over the stream

$\quad c' \leftarrow \text{nats}(n+1) ;$ — make a recursive call

$\quad c \leftarrow c' \}$ — forward c' on c

these two lines are common, so they can be abbreviated to:
$c \leftarrow \text{nats}(n+1)$

Our goal is to generate a stream of prime numbers, using the Sieve of Eratosthenes.

$$\leftarrow (\times 5) \leftarrow (\times 3) \leftarrow (\times 2) \leftarrow \text{nats}$$

We have a process per prime number we've outputted so far, and it checks numbers streaming to see if they are divisible (dropping them if they are). If a number makes it through, it's prime, so we generate a new process for it.

First, we need to build a filter transducer:

$$\text{filter} : (\text{nat} \to \text{bool}) \to \{\text{natstream} \vdash \text{natstream}\}$$

```
c ← filter p ← d =
{ n ← recv d;
  if (p n)
  then send c n;
       c ← filter p ← d        abbreviation!
  else c ← filter p ← d }
```

Q: Is the send and recursive call concurrent?
A: No, it's a synchronous language.
Q: Can I pull out the filter p recursive call?
A: No, there is a dependence. Interesting!
Q: Where does the if come from?
A: It is in the term language. This is a standard monadic programming trick.

Note that if you try to actually implement this in Coq, you won't be able to do this. The problem is that filter is not productive. Now, in the case of a true prime sieve, it will be productive, since there are infinitely many primes, but that's not obvious! There's a paper on how to do this.

Now let's do the sieve. The sieve takes a stream, knowing that the first element is a prime number (by some invariant), and then starts building processes.

$$\text{sieve} : \{\text{natstream} \vdash \text{natstream}\}$$

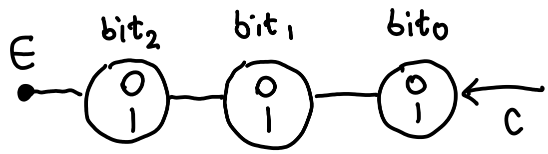
```
c ← sieve ← d =
{ k ← recv d;                       we know k is prime
  send c k;
  x ← filter (λn. k ∤ n) ← d;
  c ← sieve ← x                     not divisible by
}
```

Finally, we generate the primes stream:

```
primes : {⊢ natstream}
c ← primes =
{ x ← nats 2 ;
  c ← sieve ← x
}
```

Q: Could you do sieve using cut instead of monads?
A: Well, cut doesn't let you get the result of a function. We've just gotten rid of cut with monadic composition, since there is not really any reason to have it any more.

Our next example is a binary number counter.



operations: inc
val
dealloc

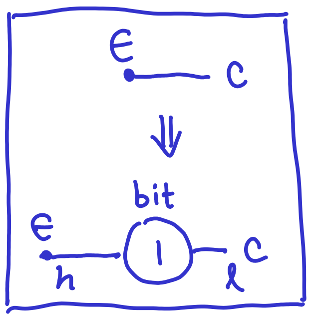
Q: Why isn't it tensor?
A: I'm outputting a value, not a new channel, and there is no dependence (so we didn't use an existential).

```
bin = &{ inc : bin
         val : nat ∧ bin
         dealloc : 1
       }
```

$$\text{val}: (\text{nat} \wedge 1) \otimes \text{bin}$$

This version sends a channel, and doesn't block! Which is potentially interesting.

Q: Why can't we just write nat here?
A: Well, nat is a tau, so it's not the right type.



flipped

```
ε : {⊢ bin}
c ← ε =
 { case recv c of
     | inc ⇒ x ← ε;
             c ← bit 1 ← x
     | val ⇒ send 0 c;
             c ← ε
     | dealloc ⇒ close c
 }
```

this is a bit confusing, because it's in reverse

```
                   higher  lower
bit : bool → { bin ⊢ bin }
lower        higher
c ← bit b ← d =
  { case recv of
      | inc ⇒ if b
              then send d inc;
                   c ← bit 0 ← d
              else c ← bit 1 ← d
      | val ⇒ send d val;
              n ← recv d;
              send c (b+2*n);
              c ← bit b ← d
      | dealloc ⇒ send d dealloc;
                  wait d;
                  close c
  }
```

Note that this is concurrent; you can send multiple increments, and they will all be in flight simultaneously!

if we omit the wait it's not linearly typed!

Homework: implement stacks.

$$\mathit{stack} = \&\{\mathit{push} : \mathit{nat} \supset \mathit{stack},$$
$$\mathit{pop} : \oplus\{\mathit{empty} : \mathit{stack},$$
$$\mathit{some} : \mathit{nat} \wedge \mathit{stack}\},$$
$$\mathit{dealloc} : 1\ \}$$

Remark: when you say 1, it really does mean that EVERYTHING is deallocated, because there must not be anything in the context. So if we made a mistake in implementing dealloc, type error!

$$\mathit{new} : \{\vdash \mathit{stack}\}$$

One last note: what is the meaning of persistent assumptions and persistent channels in this context? (I.e. bang-A) Interestingly, it doesn't seem to come up very often in this kind of program.

We have an implementation of this, but it's not quite stable enough. Note that Concurrent ML is almost expressive enough to write all these programs, but it is completely undisciplined. Session typing with these systems guarantees absence of deadlock, etc; and with restrictions on recursion, also guarantee termination.

$$\underbrace{u_1 : B_1, \ldots u_n : B_n}_{\text{shared}} ; \underbrace{x_1 : A_1, \ldots x_n : A_n}_{\text{linear}} \vdash P :: x : A$$

can show up many times

Recall that for ordinary channels, their type changes as the program evaluates. But for a shared channel, you can't allow the type to change because other references may get mixed up and type preservation would not hold. Linearity prevents this from happening, but for things that can be used multiple times, you need some restrictions. So instead, you spawn a copy of the service in question, and use that!

$$\frac{\Gamma, u : A ; \Delta, x : A \vdash Q_{u,x} :: z : C}{\Gamma, u : A ; \Delta \vdash x \leftarrow \mathit{copy}\ u ; Q_{u,x} :: z : C}\ \mathit{copy}$$

Q: So, is the reason this doesn't show up frequently because we can simulate it using the monadic syntax?
A: Well, sometimes you want the use of the shared channel to show up in the type. Consider a file storage system:

$u : \text{file} \supset (\text{file} \wedge 1)$

$x \leftarrow \text{copy } u;$
$\text{send } x\ F;$
$G \leftarrow \text{recv } x$

$u : \text{file} \supset !(\text{file} \wedge 1)$

$x \leftarrow \text{copy } u;$
$\text{send } x\ F;$
$w \leftarrow \text{recv } x;$
$\left.\begin{array}{l} y \leftarrow \text{copy } w \\ g \leftarrow \text{recv } y \\ y' \leftarrow \text{copy } w \\ g' \leftarrow \text{recv } y' \end{array}\right\}$ as many as you want

If it's correctly implemented, g and g' should be equal. With dependent types, you can actually express that these should be equal; this was a paper last year.

$u : \forall x{:}\text{file}.\ \exists y{:}\text{file}.\ \exists p{:}x \cong y.\ 1$

e.g. the index is similar

This is a dependent specification; notice that without the "input a file, output a file" there are a lot of implementations, but with the equality now we know what it should be.

$u : \forall x{:}\text{file}.\ !\exists y{:}\text{file}.\ \exists p{:}x = y.\ 1$

Note that in a distributed setting, you will need to do some runtime typechecking to ensure that all proof obligations are satisfied, if you don't trust the nodes that you are interacting with. So the proofs will have some runtime content! (On a single machine, nothing can go wrong).