When I originally taught this class, I used the pi-calculus to talk about the operational semantics of linear logic. Unfortunately, many of the people who were taking the class were unfamiliar with the pi-calculus so the ended up having to learn both at the same time. This did not work so well. So this year, I'm doing a modified format to avoid having to do this.

omitting persistent statements for simplicity

Sequent : $x_1 : A_1, \cdots x_n : A_n \vdash P :: (x : A)$

channels to use — channels we provide

recall in type theory $\Gamma \vdash M : A$ — computes value of type A

proof of A

term of type A

So we no longer call our rhs "terms", instead, we call them "processes". Furthermore, they don't reduce to "values", instead, they communicate over a channel.

We communicate in two ways: by providing a service (channel), and by consuming a service.

So we are going to talk about what our logical rules mean now explicitly talking about channels and processes.

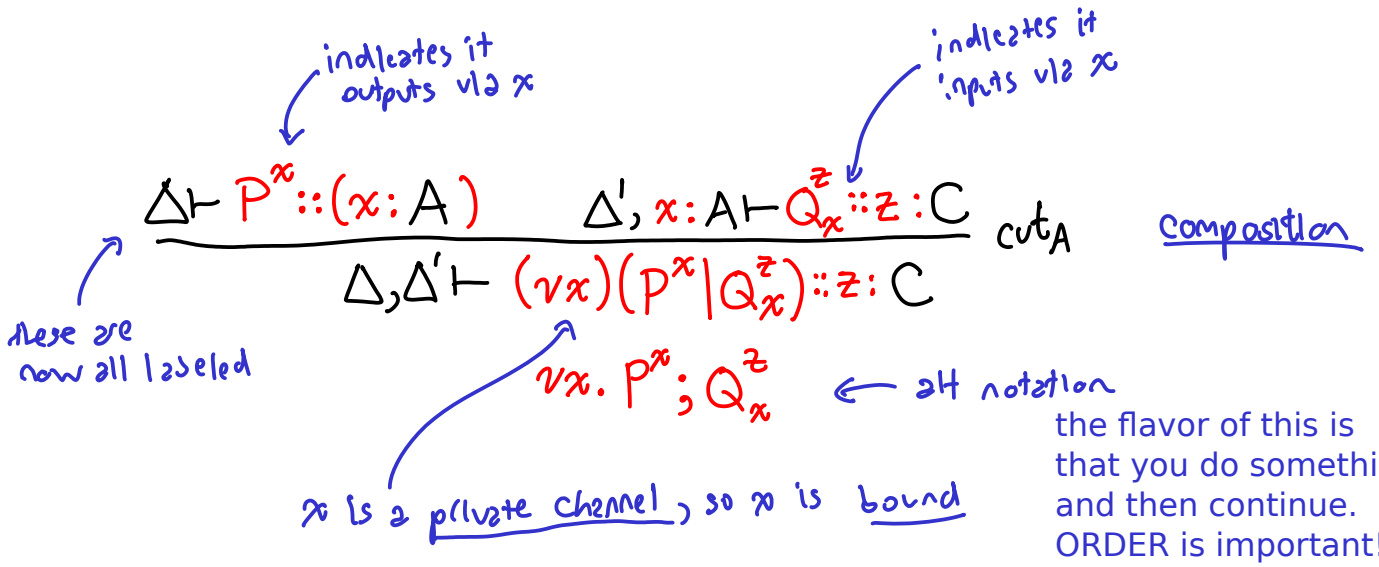
$$\frac{}{y : A \vdash [x \leftarrow y] :: (x : A)}\ id_A$$

forwarding

Importantly: the communication can be BOTH WAYS. It is like an wire that connects x and y.

Q: Can you say something about variable scoping?
A: The scope is the sequent itself. In the simply typed version, propositions don't contain the variables; so there is no notion of dependence.

Dependent linear logic is an open research question.



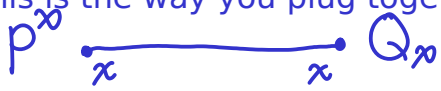
composition

the flavor of this is that you do something, and then continue. ORDER is important! (in pi calculus, this is not true.)

Interestingly enough, we identify processes by their output processes. However, both the bottom and the top produce z. This has to do with the asymmetry to linear logic, where you can use multiple resources but can only output one resource.

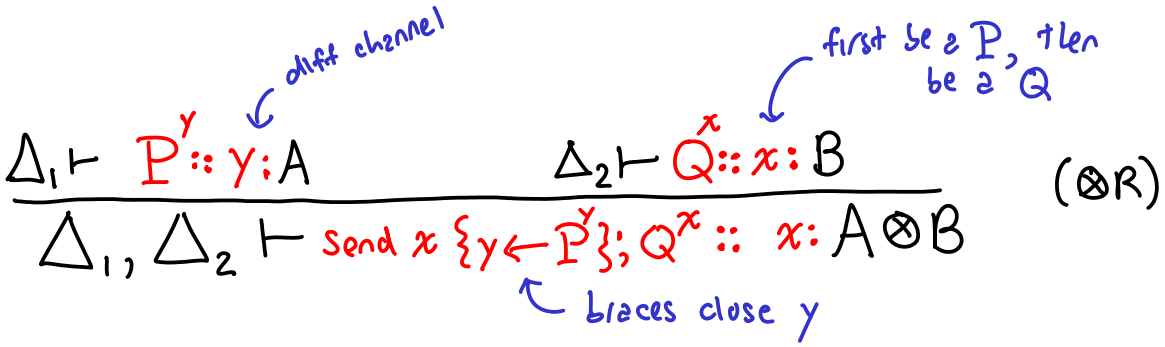
Q: If Q happens to not run x, they run in parallel?
A: This is impossible. It's a linear variable so it must be used, and must be used once. So it is in fact very difficult to represent a process calculus where things run independently without interacting.

Q: If you identify the process on the right with the variable, why do you need it?
A: When you plug them together, they communicate along channel x. If you don't have variables, you cannot actually have the communication between the two processes.
Q: Why do you need the variable for P?
A: Well, P provides the service along x.
Q: Are the superscripts necessary?
A: No. I'm just writing them so you remember what the rules are. But the channel is absolutely necessary.
Q: What if you changed the identity rule so that it didn't need it on the right side, it seems cut would still be valid?
A: Well, the cut would be valid, but you wouldn't know what P would do which has to do with actions along the channel x. If you erase x, the program makes no sense; even though x is linear, it has TWO occurrences, one in P and one in Q. This is the way you plug together two processes.



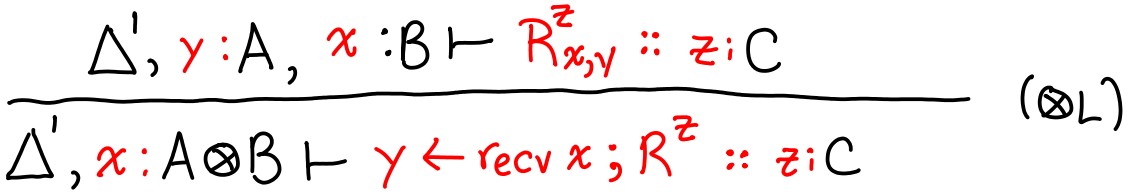
Q: Why is this better for distributed systems?
A: You can identify this as the location the process runs. BTW, this presentation is called a "synchronous calculus" in the literature, where they have to match exactly. In an asynchronous calculus, you can send and keep working. We have a paper on that, but it is not in this presentation. This is an interesting fact: the LOGIC does not determine the COMPUTATION.

Warning: the programs in this lecture are not that interesting; we'll need inductive types and coinductive types so that we can send data around. So take faith!



Idea: output A (not the process itself, but new channel y which provides A) along x, then behave as B along x. (We can send processes on channels, but usually we just send channels on channels.)

We also break symmetry here, due to sequentiality. We could also do B first, and then A.



Q: Does A tensor B imply B tensor A?
A: Yes, we can do this. I am going to do this after the formal semantics.

$y : A \otimes B \vdash \quad ? \quad ? \quad ? \quad :: x : B \otimes A$

Recall we had some checks for the logical rules last lecture. This is very similar when we do reduction on proofs, as a result of manipulations on the proof we're working with. So now we need to figure out what proof reduction corresponds to computationally.

Cut reduction is COMMUNICATION.

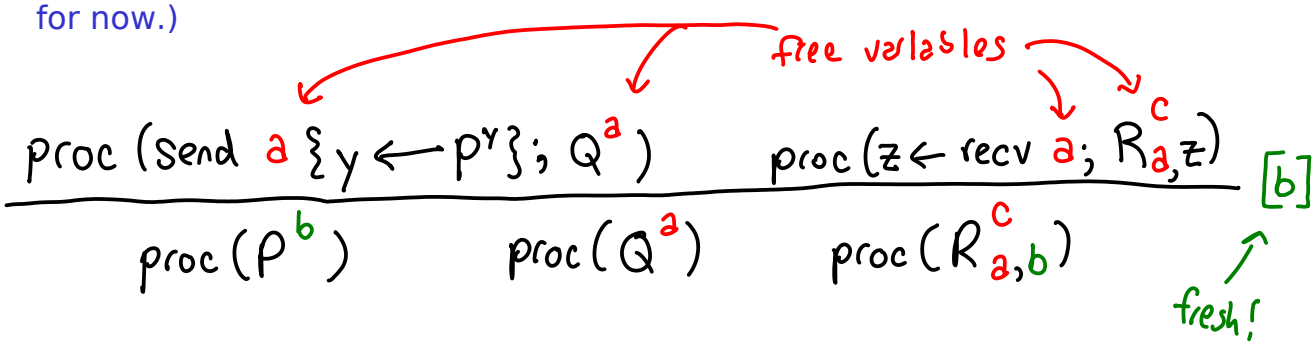
> In natural deduction, proof reduction = substitution
> In sequent calculus, proof reduction = communication
>
> It is a much smaller unit of computation than substitution: we perform a very small step of communication (one send is matched up with one receive, and then we continue). No global substitution. This is true even when you have sequent calculus formulations of intuitionistic logic.

Recall that we know how to represent computation as forward inference.
So we will use this to describe the semantics (rather than leaning on pi-calculus, which would maket his very easy.)

$$\underset{\text{state}}{proc(P_1), proc(P_2) \cdots proc(P_n)} \rightsquigarrow \Delta'$$

(transition)

We know how to do this from the original lecture! (Ignore the connectives for now.)



In process calculus, this is called a bound output.

Q: In the premise, we have ys in both processes. Can you change a y to something else?
A: Yep. [updated in notes]. Scope of y is inside the braces, scope of z is inside the process.

Time to show the previous problem. Recall proofs are processes! So build it up using the proof.

$$\dfrac{\dfrac{}{y:B \vdash [y' \leftarrow y] :: y' : B}\,Id_B \quad \dfrac{}{z:A \vdash [x \leftarrow z] :: x : A}\,Id_A}{z:A,\ y:B \vdash send\ x\{y' \leftarrow [y' \leftarrow y]\};\ [x \leftarrow z] :: x : B \otimes A}\,\otimes r$$

$$\dfrac{z:A,\ y:B \vdash send\ x\{y' \leftarrow [y' \leftarrow y]\};\ [x \leftarrow z] :: x : B \otimes A}{y:A \otimes B \vdash z \leftarrow recv\ y;\ send\ x\{y' \leftarrow [y' \leftarrow y]\};\ [x \leftarrow z] :: x : B \otimes A}\,\otimes l$$

$z : A \leftarrow recv\ y;$
$send\ x\ \{y' \leftarrow [y' \leftarrow y]\};$
$[x \leftarrow z]$

this happens frequently, so often say [send x y]

$y : A \otimes B \quad x : B \otimes A$
$y : B$
$x : A$

Channels CHANGE TYPE as communication proceeds!

Typically you don't have to write these programs, but these are what are at the bottom of it.

One thing that I don't want to discuss is linear implication.

$x : A \multimap B$

Input on A and behave as B. We've already got enough process expressions to express this; right rule is a receive, left rule is a send. But there are no new computational properties so I will skip it.

The fact that they are adjoint causes linear implication on the right to have the same computational meaning as tensor product on the left, and vice versa.

$$\frac{proc(\nu x.\, P^{x};\, Q_x^{c})}{proc(P^{a}) \qquad proc(Q_a^{c})}\,[a]$$

$$\frac{proc([a \leftarrow b]) \qquad proc(Q_a)}{proc(Q_b)}$$

It turns out there is another way to do this which keeps the process around, and forward values. But this is wasteful.

Note that in the pi calculus, there is a concept of "scope extrusion", because private channels can be passed around by the processes that have access to it. All a private allocation means is that it is not referred to by anyone at the time it is allocated.

$$\frac{}{\cdot \vdash close(x) :: x : 1}\,1L$$

$$\frac{\Delta \vdash R^{z} :: z : C}{\Delta, x : 1 \vdash wait(x);\, R^{z} :: z : C}\,1R$$

$$\frac{proc(close(a)) \qquad proc(wait(a);\, R)}{proc(R)}$$

So 1 (tensor unit) simply terminates a process.

$$\frac{\Delta \vdash P^x :: x : A \qquad \Delta \vdash Q^x :: x : B}{\text{case recv of } \pi_1 \Rightarrow P^x \mid \pi_2 \Rightarrow Q^x :: x : A \& B}\ \&R$$

Intuitively, you have to input along x whether or not you want A or B.

$$\frac{\Delta, x : A \vdash R_x^z :: z : C}{\Delta, x : A \& B \vdash \text{send } x\ \pi_1;\ R_x^z :: z : C}\ \&L_1$$

$$\frac{\text{process}(\text{case recv } a \text{ of } \pi_1 \Rightarrow P^a \mid \pi_2 \Rightarrow Q^a) \qquad \text{process}(\text{send } a\ \pi_1;\ R_a^c)}{\text{process}(P^a) \qquad\qquad \text{process}(R_a^c)}$$

Offer choice between A and B along x $\quad x : A \& B$

Provide either A opr B along x $\quad x : A \oplus B$

Preview for next time:

$x : \$file \multimap \$file \otimes 1$

↖ way of embedding data

$natstream = nat \otimes natstream$ (coinductive)

$P :: x : natstream$

Q: Because the first nat is waiting for a receive, this program terminates. Is there a type that doesn't terminate?
A: The obvious program doesn't terminate.

You could write a filter predicate which will not necessarily be productive. But this fails the coinductive definition test.

$stack = \&(nat \multimap stack)$ push

$\quad \& ((nat \oplus 1) \otimes stack)$ pop

$\quad \& 1$ deallocate

We'll write two programs: one that is a single program with a stack init, and another where each element is its own process and they are linked together.

If cut elimination holds, then these are strongly normalizing for pure programs. With recursion, it may not terminate unless you have some conditions.

Q: I've been working on the A tensor B and B tensor A proof. I can do it mechanically, but I'm looking for intuition.
A: We'll get lots of programs; next lecture is just writing programs. Hopefully by then, we'll have an idea what these things mean. But I think that you're not alone: the reason why this process interpretation has been missed is because it's intuitionistic logic (and Girard insisted on classical logic), and two is because tensor is asymmetric. So it's not a coincidence that this is not intuitive.

Q: Is there a symmetric version?
A: There is some flexibility. I wasn't going to say very much about it.
Q: Was Girard insisting on classical because there is something nice?
A: Girard was interested in the notion of a proof net, which is much more straightforward in the classical case. His claim is that the classical operation is already constructive, so we don't need an intuitionistic version. But if you try to build a process interpretation, the intuitionistic version is more intuitive; in the classical case, there is no correspondence between a process and proposition.
Q: What's the difference between it?
A: In the intuitionistic version,

But in classical logic, there can be multiple things on the right hand side:

Linear implication changes meaning in this setting.

Q: So how do you encode recursive types in linear logic? If linear logic has this cut elimination property, there must be a way to do it in the base logic.
A: Well, one way to do it is to just add it, because the theory is open ended. Alternatively, you can use the polymorphic version of linear logic, to use standard polymorphic encodings of data types. I will not do that, so we will add it as new type constructors in the logic which is consistent with what we already have.
Q: Cut elimination would not hold anymore?
A: It depends on what restrictions you place. If you do it will hold, if you don't it won't.

Q: In ordinary intuitionistic type theory, you get different computation types when you do fixpoints/cofixpoints from inductive types.
A: I don't know the answer. We have not investigated the metatheory of this with arbitrary inductive/coinductive types. We have just submitted a paper which is a first step for this. We have also written down a dependent version, but recall in dependent types equality plays a central role, but what kind of equality do you want to impose between processes? Strong bisimulation? Weak bisimulation? We don't know.

Q: Is it common to do cut elimination in this way, where you use the [indistinct] rule, instead of just permuting in the logic itself?
A: The difference between cut elimination and this thing, is it's a deep thing; normalizing the term even under lambdas, whereas when we hit a lambda binder, we don't normalize anymore. Here, when we hit a send/receive, we stop. That is the difference between computation and full cut elimination is where you start. This is the same as in functional programming languages.
Q: But you can still an analog of call by value, where instead of these rules, when you take two processes, you just reduce the [indistinct] of the two processes.
A: We did that in the original paper. The difficulty was our proofs were in the pi calculus, and it's a little tricky for where you can apply structural equivalence. The details are tricky. But you can do that too.