Session-Typed Concurrent Programming
Communication

Frank Pfenning

June 18, 2019

Introduction

We continue by reiterating the definition of Cut, and the definition of Identity.

$$\frac{A \vdash B \qquad B \vdash C}{A \vdash C}\ \mathrm{Cut_B}$$

$$\frac{}{A \vdash A}\ \mathrm{ID_A}$$

There are two things that can go wrong with a logic, and to capture this formally, we introduce dual notions of soundness and completeness. If the elimination rules are too strong, if we can extract information we didn't put in, we call it unsound. Conversely, if we cannot extract all the information we put in, we say that our our system lacks completeness. We say our logic is in Harmony if it is both sound and complete. To characterize completeness and soundness we provide the following two informal definitions:

- **(local) Completeness** We say that the logic is complete if we can, given an arbitrary proof of a proposition, apply elimination rules to reintroduce the proposition. For instance, we may prove identities on connectives, such as $A\&C \vdash A\&C$ by using the rules of those connectives and the identity rule on the variables.
- **(local) Soundndess** We say that the logic is sound if an introduction followed by an elimination can be reduced to a simpler proof. The logic is unsound if we can prove entailments which don't follow from the premises. Cut elimination shows that this cannot happen. For any proof of $A \vdash C$ we also have a cut-free proof, and there are no rules other than cut that apply to $A \vdash C$, so such a proof does not exist.

An example of incompleteness

The rules for the & connective are as follows.

$$\frac{A \vdash B \qquad A \vdash C}{A \vdash B\&C}\ \&\mathrm{R}$$

$$\frac{B \vdash D}{B\&C \vdash D}\ \&\mathrm{L_1}$$

$$\frac{C \vdash D}{B \& C \vdash D}\ \&\mathrm{L}_2$$

The logic becomes incomplete if we remove the $\&L_1$ rule. If we remove this rule, we can no longer prove $B\&C \vdash C\&B$. We show this with the following example:

$$\frac{\dfrac{B \vdash C}{B\&C \vdash C}\ \&\mathrm{L}_1 \qquad \dfrac{\overline{B \vdash B}^{ID_B}}{B\&C \vdash B}\ \&\mathrm{L}_1}{B\&C \vdash C\&B}\ \&\mathrm{R}$$

We cannot prove $B\&C \vdash C\&B$ without the $\&L_1$ rule. The proof gets stuck because $B \vdash C$ cannot be proven.

Without the $\&L_1$ rule we also can no longer prove the identity on connectives by using identity on variables:

$$\frac{}{A\&B \vdash A\&B}\ \mathrm{ID}$$

In the full logic we can prove this holds as follows.

$$\frac{\dfrac{\overline{A \vdash A}^{ID_A}}{A\&B \vdash A}\ \&\mathrm{L}_1 \qquad \dfrac{\overline{B \vdash B}^{ID_B}}{A\&B \vdash B}\ \&\mathrm{L}_2}{A\&B \vdash A\&B}\ \&\mathrm{R}$$

This proof uses both the $\&L_1$ and $\&L_2$ rules. Without the $\&L_1$ rule the proof no longer works. Using cut elimination we can show that there is no proof of $A\&B \vdash A\&B$.

Proofs as programs

An integral relationship between logic and computation is summarized in the Curry Howard Correspondence, which details the mutual interpretations between proofs and programs. We describe several examples in the following table.

Curry Howard Correspondence	
Proposition	Type
Proof	Program
Proof Reduction	Computation

We now cover this correspondence for our logic, and show that the logic corresponds to a language with concurrency, or the ability to have multiple computations occurring at the same time. To do so, we define $A \vdash P : B$ where A is a type prescribing communication on the left, P is a concurrent program and B is a type prescribing communication on the right.

The cut rule in this system does parallel composition.

$$\frac{A \vdash P : B \qquad B \vdash Q : C}{A \vdash (P|Q) : C}\ \text{Cut}$$

Given processes P and Q, the cut rule produces a process $P|Q$, which is the parallel composition of P and Q. The process P communicates with Q via the type B.

We can also show that identity allows for forwarding, according to this model.

$$\frac{}{A \vdash (\iff) : A}\ \text{ID}$$

This rule produces a process that simply forwards messages of type A.
We now show how our proof of cut elimination corresponds to reduction on these processes.

$$\frac{A \vdash A \qquad A \vdash C}{A \vdash C}\ \text{Cut} \Rightarrow A \vdash C$$

This rule can compose with P as follows.

$$\frac{A \vdash A \qquad A \vdash P : C}{A \vdash (\iff |P) : C}\ \text{Cut} \Rightarrow A \vdash P : C$$

Our system reduces $(\iff |P) \Rightarrow P$ and $(P| \iff) \Rightarrow P$. These are of left type A and of right type C, which does not change: $({}_A \iff |{}_A P)_C \Rightarrow {}_A P_C$ and $({}_A P|_B \iff)_B \Rightarrow {}_A P_B$. This is therefore defining a reduction relation, and reduction does not change the outside interface.

The $\oplus$-connective corresponds to the left side communicating one of two possibilities, and the right side doing case analysis on that message. We show how the reduction rule follows from the cut elimination proof:

$$\frac{\dfrac{A \vdash P : B}{A \vdash R.\pi_1; P : B \oplus C}\ \oplus R_1 \qquad \dfrac{B \vdash Q : D \quad C \vdash R : D}{B \oplus C \vdash D}\ \oplus L}{A \vdash (R.\pi_1; P)|CaseL(\pi_1 \Rightarrow Q, \pi_2 \Rightarrow R)}\ \text{Cut}$$

By cut elimination this reduces to:

$$\frac{A \vdash P : B \quad B \vdash Q : D}{A \vdash P|Q : D}\ \text{Cut}$$

We therefore have these reductions:

$$(R.\pi_1; P)|CaseL(\pi_1 \Rightarrow Q, \pi_2 \Rightarrow R) \implies P|Q$$
$$(R.\pi_2; P)|CaseL(\pi_1 \Rightarrow Q, \pi_2 \Rightarrow R) \implies P|R$$

Note that the R in $R.\pi_1$ and $R.\pi_2$ is not the variable R, but part of the syntax to indicate that the π_1 and π_2 are being sent to the right. We now show the types remain correct when we add processes to our original cut elimination proof.

The original logical rules for & without processes:

$$\frac{A \vdash B \qquad A \vdash C}{A \vdash B \& C}\ \&\text{R} \qquad \frac{B \vdash D}{B \& C \vdash D}\ \&\text{L}_1 \qquad \frac{C \vdash D}{B \& C \vdash D}\ \&\text{L}_2$$

With processes:

$$\frac{A \vdash P_1 : B \qquad A \vdash P_2 : C}{A \vdash caseR(\pi_1 \Rightarrow P_1 | \pi_2 \Rightarrow P_2) : B \& C} \ \&\mathrm{R}$$

$$\frac{B \vdash Q : D}{B \& C \vdash L.\pi_1; Q : D} \ \&\mathrm{L}_1$$

$$\frac{C \vdash Q : D}{B \& C \vdash L.\pi_2; Q : D} \ \&\mathrm{L}_2$$

Our cut elimination proof gives the following reduction rules:

$$(caseR(\pi_1 \Rightarrow P_1 | \pi_2 \Rightarrow P_2) | L.\pi_1; Q) \Rightarrow P_1 | Q$$

$$(caseR(\pi_1 \Rightarrow P_1 | \pi_2 \Rightarrow P_2) | L.\pi_2; Q) \Rightarrow P_2 | Q$$

Programming with processes

We now write a few programs in this language. To do so we need to add recursive types. The first program takes a bitstream as input and produces the same bitstream with all bits flipped as output. The type of bitstreams is given by the recursive equation $bits = bits \oplus bits$.

We define a process $bits \vdash flip : bits$ that flips the bits in an infinite bit stream. Flip needs to be defined with a $Case_{\mathrm{L}}$. If we receive a 1 then we emit a 0 and if we receive a 0 then we emit a 1. In both cases we continue with flip.

$$flip = Case_{\mathrm{L}}(\pi_1 \Rightarrow R.\pi_2; flip | \pi_2 \Rightarrow R.\pi_1; flip)$$

Dually, we can define $lits = lits \& lits$, and we define a lits-flipping process $lits \vdash flop : lits$. This process sends on the left and receives on the right.

We define the process $bits \vdash comp_1 : bits$ that compresses runs of 1's in the bit stream to a single 1.

$$comp_1 = Case_{\mathrm{L}}(\pi_1 \Rightarrow R.\pi_1; ignore_1 | \pi_2 \Rightarrow R.\pi_2; comp_1)$$

$$ignore_1 = Case_{\mathrm{L}}(\pi_1 \Rightarrow ignore_1 | \pi_2 \Rightarrow R.\pi_2; comp_1)$$

The language has forwarding, case left, case right, and parallel composition. These rules are limited; we have no memory in our language. We're limited to writing Finite State Automata (FSA) / Finite State Transducers (FST). These is an automata/transducers that allows parallel computation, but only computations that can be done on automata/transducers.

We can compose $flip$ and $comp_1$ to write a program that compresses 0 bits:

$$comp_0 = flip | comp_1 | flip$$

This language allows us to compose programs in parallel, but only in a linear chain. Communication could be asynchronous for even more concurrency, but in our current language we can actually implement asynchronous sending. We do this by creating a queue using the forwarding process.

$$comp_1 = Case_{\mathrm{L}}(\pi_1 \Rightarrow ignore_1 | (R.\pi_1; \Longleftrightarrow) | \pi_2 \Rightarrow comp_1 | (R.\pi_2; \Longleftrightarrow))$$

To create a general finite state automata that accepts any finite bitstream, we generalize to from binary to n-ary $\oplus$.

$$fits_\alpha = \oplus\{\pi_1 : fits_\alpha, \pi_2 : fits_\alpha, e : \alpha\}$$

This has three cases, pi_1 and pi_2 for a bit in the bit stream, and e for indicating that the stream has ended.

$$fits \vdash even \oplus \{accept : \alpha, reject : \alpha\}$$

$$even = Case_{\mathrm{L}}(\pi_1 \Rightarrow odd|\pi_2 \Rightarrow R.reject; consume|e \Rightarrow R.accept; \iff)$$

$$odd = Case_{\mathrm{L}}(\pi_1 \Rightarrow R.reject; consume|\pi_2 \Rightarrow even; |e \Rightarrow R.accept; \iff)$$

In order for the above program to work, we must define a consume function.

$$consume = Case_{\mathrm{L}}(\pi_1 \Rightarrow consume|\pi_2 \Rightarrow consume|e \Rightarrow \iff)$$

Conclusion

In summary what we introduced was a singleton logic with few connectors, and we interpreted propositions as types and proofs as programs. Once we have this we can write interesting things like FSAs and FSTs.

Turing Machine cells are lined up nicely, in a row, just like our chain of processes. We are almost at the computational power of Turing Machines, as we will see next time.