Lecture Notes on Computation in Structural Logic

15-816: Substructural Logics
Frank Pfenning

Lecture 16
October 25, 2016

In this lecture we attempt to extend our computational interpretation of ordered and linear logic to include structural logic (all of these being intuitionistic, of course). This completes what we started in Lecture 15, Section 2 where we provided an operational semantics only for the shift operators.

Our analysis will turn out to be different from the usual, celebrated *Curry-Howard isomorphism* between intuitionstic logic and the simply-typed λ-calculus [How69]. There are two reasons for this divergence. For one, and perhaps most importantly, we are working here with a sequent calculus instead of natural deduction. This means that the engine of computation is cut reduction, instead of the usual substitution. Cut reduction is a local transformation on a proof and proceeds in very small steps. You may remember the slogan of *cut reduction as communication*. In natural deduction, *substitution* is the engine of computation which is a much more global, "big step" operation. These two have been related in the past, most notably perhaps is Herbelin's analysis [Her94]. One new ingredient here is the explicit presence of concurrency, and that integration of ordered, linear, and structural computations.

Disclaimer: This lecture very much represents my very recent understanding of the state of affairs, based mostly on intuition without any carefully formulated, much less checked proofs.

1 Structural Intuitionistic Logic

We are aiming at the following combined system.

$$\begin{array}{rrcl} \text{Structural} & A_{\mathsf{U}} & ::= & p_{\mathsf{U}} \mid A_{\mathsf{U}} \to B_{\mathsf{U}} \mid A_{\mathsf{U}} \mathbin{\&} B_{\mathsf{U}} \mid A_{\mathsf{U}} \times B_{\mathsf{U}} \mid \mathbf{1} \mid A_{\mathsf{U}} + B_{\mathsf{U}} \mid \uparrow_{\mathsf{L}}^{\mathsf{U}} A_{\mathsf{L}} \\ \text{Linear} & A_{\mathsf{L}} & ::= & p_{\mathsf{L}} \mid A_{\mathsf{L}} \multimap B_{\mathsf{L}} \mid A_{\mathsf{L}} \mathbin{\&} B_{\mathsf{L}} \mid A_{\mathsf{L}} \otimes B_{\mathsf{L}} \mid \mathbf{1} \mid A_{\mathsf{L}} \oplus B_{\mathsf{L}} \mid \downarrow_{\mathsf{L}}^{\mathsf{U}} A_{\mathsf{U}} \end{array}$$

We already treated the linear mode and the shift modalites, although a slight update will be necessary. So today we focus on the structural layer only. Because of that, we will omit the subscript U and just write A, B, etc.

We begin with $A \times B$. It seems odd that our structural logic contains both $A \times B$ and $A \mathbin{\&} B$. Indeed, it turns out that they are logically equivalent in the sense that $A \times B \vdash A \mathbin{\&} B$ and $A \mathbin{\&} B \vdash A \times B$. But they behave very differently operationally, so it may be worthwhile to support both.

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \times B}\ \times R \qquad \frac{\Gamma, A \times B, A, B \vdash C}{\Gamma, A \times B \vdash C}\ \times L$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \mathbin{\&} B}\ \mathbin{\&} R \qquad \frac{\Gamma, A \mathbin{\&} B, A \vdash C}{\Gamma, A \mathbin{\&} B \vdash C}\ \mathbin{\&} L_1 \qquad \frac{\Gamma, A \mathbin{\&} B, B \vdash C}{\Gamma, A \mathbin{\&} B \vdash C}\ \mathbin{\&} L_2$$

Since antecedents persist, the two right rules for $A \times B$ and $A \mathbin{\&} B$ coincide! This should raise a red flag, but the presence of weakening and contraction allows us to verify harmony for each of the connectives despite the differing left rules. This exemplifies a lesson I learned after many years of working in this field: the presence of structural rules makes observations "fuzzier", systems less crisp, and allows one to get away with some things that are not as elegant as one would like. This is one reason that I am teaching this course now and start with the weakest logic I could easily make sense of (subsingleton logic), working my way up to the present point (combined structural and substructural logics).

The next point will be to derive a one-premise right rule for $A \times B$. As a reminder: we do this here (and also for $A \to B$ in Section 3) so that the spawning of new processes is limited to the cut rule, which is a significant simplification of the operational semantics. We can go back and forth between the one-premise and two-premise rules, using cut in one direction

and identity in the other.

$$\frac{\Gamma, A \vdash B}{\Gamma, A \vdash A \times B}\ \times R^*$$

$$\frac{\Gamma \vdash A \quad \dfrac{\dfrac{\Gamma \vdash B}{\Gamma, A \vdash B}\ \mathsf{weaken}}{\Gamma, A \vdash A \times B}\ \times R^*}{\Gamma \vdash A \times B}\ \mathsf{cut}_A \qquad \frac{\dfrac{}{\Gamma, A \vdash A}\ \mathsf{id}_A \quad \Gamma, A \vdash B}{\Gamma, A \vdash A \times B}\ \times R$$

2 Assigning Process Terms to Proofs of Positive Propositions

Next, we have to design a process assignment. Experience dictates that we should try the sending rule first. By analogy with the linear logic, where $\otimes R$ sends, here, $\times R$ should send.

$$\frac{\Gamma, w{:}A \vdash P :: (x : B)}{\Gamma, w{:}A \vdash \mathsf{send}\ x\ w\ ;\ P :: (x : A \times B)}\ \times R^*$$

Persistence of antecedents implies that even though we send w along x, we also retain w. The left rule is more complicated: in the premise, we have to figure out how to label the new antecedent A and B. Previously, we would have written $x{:}B$, but now we need to retain $x{:}A \times B$ since it may continue to occur in Q, so we rename it to x'.

$$\frac{\Gamma, x{:}A \times B, y{:}A, x'{:}B \vdash Q :: (z : C)}{\Gamma, x{:}A \times B \vdash (y, x') \leftarrow \mathsf{recv}\ x\ ;\ Q :: (z : C)}\ \times L$$

This means we actually receive two channels: $y{:}A$, corresponding to w that is being sent, and a continuation channel $x'{:}B$. In the end, though, perhaps it is not too surprising that $x{:}A \times B$ will send both a $y{:}A$ and an $x'{:}B$. Both of these, together with x and also z can occur in Q. We have chosen not to display this dependence explicitly but rely on the judgment in the premise to express this information.

Now, however, we should be concerned with a mismatch: $\times R^*$ appears to send only one channel (namely w) while $\times L$ expects two. But during asynchronous communication we also have to create a new channel x' to represent the continuation of the process, so that

$$\mathsf{send}\ x\ w\ ;\ P \simeq x' \leftarrow [x'/x]P\ ;\ (\mathsf{send}\ x\ w\ ;\ x \leftarrow x')$$

Using this as a guidance, we get the following rule which introduces a new continuation channel c and substitutes this for x in P.

$$\frac{\mathsf{proc}(x, \mathsf{send}\ x\ w\ ;\ P)}{\mathsf{proc}(c, [c/x]P) \quad \mathsf{msg}(x, \mathsf{send}\ x\ w\ ;\ x \leftarrow c)}\ \times C\_\mathsf{send}^c$$

The message reads: *send w along x and continue as c.* So we do indeed have two channels for the recipient, even if one is not explicit in the syntax of the sender.

$$\frac{\mathsf{msg}(x, \mathsf{send}\ x\ w\ ;\ x \leftarrow c) \quad \mathsf{proc}(z, (y, x') \leftarrow \mathsf{recv}\ x\ ;\ Q)}{\mathsf{proc}(z, [c/x][w/y]Q)}\ \times C\_\mathsf{recv}$$

Ah, we have ignored one important aspect here: these rules are written as if all communications are linear. But they are not! The original channel $x : A \times B$ along which we offer may have multiple clients. This seems okay when the message is sent (since now $\mathsf{msg}(x, \ldots)$ provides along x), but it becomes problematic when the message is received. In order to avoid that other clients of x to be left dangling, we make this message *persistent*.

$$\frac{\mathsf{proc}(x, \mathsf{send}\ x\ w\ ;\ P)}{\mathsf{proc}(c, [c/x]P) \quad \underline{\mathsf{msg}}(x, \mathsf{send}\ x\ w\ ;\ x \leftarrow c)}\ \times C\_\mathsf{send}^c$$

$$\frac{\underline{\mathsf{msg}}(x, \mathsf{send}\ x\ w\ ;\ x \leftarrow c) \quad \mathsf{proc}(z, (y, x') \leftarrow \mathsf{recv}\ x\ ;\ Q)}{\mathsf{proc}(z, [c/x][w/y]Q)}\ \times C\_\mathsf{recv}$$

The processes themselves proceed with their continuations P and Q, after some channel substitution, after the persistent message has been sent or received, respectively.

We next consider disjunction as another positive proposition, $A + B$ (usually written as $A \vee B$) or, as a more convenient type, $+\{l_i : A_i\}_{i \in I}$. First, logically:

$$\frac{\Gamma \vdash A_k}{\Gamma \vdash +\{l_i : A_i\}_{i \in I}}\ +R_k \qquad \frac{\Gamma, +\{l_i : A_i\}_{i \in I}, A_i \vdash C \quad (\text{for all } i \in I)}{\Gamma, +\{l_i : A_i\}_{i \in I} \vdash C}\ +L$$

Again, judging merely from the perspective of provability, the antecedent $+\{l_i : A_i\}_{i \in I}$ is redundant, but we adhere to the principle of persistence of antecedents in structural logic. The process terms look quite similar to the

rules for linear $\oplus$, but we have to account for the continuation which we call x' in the rule.

$$\frac{\Gamma \vdash P :: (x : A_k)}{\Gamma \vdash x.l_k \ ; P :: (x : +\{l_i : A_i\}_{i \in I})} \ +R_k$$

$$\frac{\Gamma, x{:}+\{l_i : A_i\}_{i \in I}, x'{:}A_i \vdash Q_i :: (z : C) \quad (\text{for all } i \in I)}{\Gamma, x{:}+\{l_i : A_i\}_{i \in I} \vdash \mathsf{case}\ x\ (l_i(x') \Rightarrow Q_i)_{i \in I} :: (z : C)} \ +L$$

The computation rule follows the previous pattern: since the offer is along a persistent channel, we create a persistent message and a fresh continuation channel c.

$$\frac{\mathsf{proc}(x, x.l_k \ ; P)}{\mathsf{proc}(c, [c/x]P) \quad \underline{\mathsf{msg}}(x, x.l_k \ ; x \leftarrow c)} \ +C\_\mathsf{send}^c$$

Receiving the message will select the correct branch and also substitute the continuation channel c for x'.

$$\frac{\underline{\mathsf{msg}}(x, x.l_k \ ; x \leftarrow c) \quad \mathsf{proc}(z, \mathsf{case}\ x\ (l_i(x') \Rightarrow Q_i)_{i \in I})}{\mathsf{proc}(z, [c/x']Q_k)} \ +C\_\mathsf{recv}$$

We do not need to mention the message in the conclusion since it is persistent. Persistence is again critical since there may be many clients of x and we cannot leave them dangling without a provider. One of these clients could be Q_k itself since it may depend on x. As was noted in lecture, this dependence on Q_k in x is not strictly necessary, but it is sometimes convenient.

As the last positive proposition we have $\mathbf{1}$. The only novelty here is that we have no continuation.

$$\frac{}{\Gamma \vdash \mathbf{1}} \ \mathbf{1}R \qquad \frac{\Gamma, \mathbf{1} \vdash C}{\Gamma, \mathbf{1} \vdash C} \ \mathbf{1}L$$

The left rule looks like a typo, but it is not. The principal formula of the inference persists, but no other antecedents are generated. Operationally, it may make a little more sense.

$$\frac{}{\Gamma \vdash \mathsf{close}\ x :: (x : \mathbf{1})} \ \mathbf{1}R \qquad \frac{\Gamma, x{:}\mathbf{1} \vdash Q :: (z : C)}{\Gamma, x{:}\mathbf{1} \vdash \mathsf{wait}\ x \ ; Q :: (z : C)} \ \mathbf{1}L$$

$$\frac{\mathsf{proc}(x, \mathsf{close}\ x)}{\underline{\mathsf{msg}}(x, \mathsf{close}\ x)} \ \mathbf{1}C\_\mathsf{send} \qquad \frac{\underline{\mathsf{msg}}(x, \mathsf{close}\ x) \quad \mathsf{proc}(z, \mathsf{wait}\ x \ ; Q)}{\mathsf{proc}(z, Q)} \ \mathbf{1}C\_\mathsf{recv}$$

This means potentially many clients can check that a persistent provider has terminated by closing its persistent channel. There does not seem to be much point to allow this in Q, but we should be careful and find a good proof-theoretic justification for omitting $x{:}\mathbf{1}$ in the premise before we change our rules.

Now we can think about the meaning of purely positive types, such as

$$\mathsf{list}_A = +\{\mathsf{cons} : A \times \mathsf{list}_A, \mathsf{nil} : \mathbf{1}\}$$

We see a parallel with functional programming here: any datatype declaration with eager constructors and no embedded functions are represented here as a purely positive recursive type. The labels of the sum represent the constructors.

Image we have a process $P :: (x : \mathsf{list}_A)$. If it runs to completion, if will asynchronously send a number of persistent messages. For example, if P sends messages corresponding to the list $[a, b]$, they would look like:

$\underline{\mathsf{msg}}(x, x.\mathsf{cons}\ ;\ x \leftarrow x_1)$
$\underline{\mathsf{msg}}(x_1, \mathsf{send}\ x_1\ a\ ;\ x_1 \leftarrow x_2)$
$\underline{\mathsf{msg}}(x_2, x_2.\mathsf{cons}\ ;\ x_2 \leftarrow x_3)$
$\underline{\mathsf{msg}}(x_3, \mathsf{send}\ x_3\ b\ ;\ x_3 \leftarrow x_4)$
$\underline{\mathsf{msg}}(x_4, x_4.\mathsf{nil}\ ;\ x_4 \leftarrow x_5)$
$\underline{\mathsf{msg}}(x_5, \mathsf{close}\ x_5)$

We can see that this is an explicit linked list representation of the list, where channels act like pointers. These messages are persistent, which means multiple clients can access this data structure, simultaneously in multiple places. It is important however that is is *immutable*, where receiving a message is synonymous with reading the associated data.

In some sense this is a somewhat wasteful representation. We could for example, construct longer messages which would be outside of our currently envisioned grammer of what messages are. For example, to represent all of these in one big message, we could have

$\underline{\mathsf{msg}}(x, x.\mathsf{cons}\ ;\ \mathsf{send}\ x\ a\ ;\ x.\mathsf{cons}\ ;\ \mathsf{send}\ x\ b\ ;\ x.\mathsf{nil}\ ;\ \mathsf{close}\ x)$

However, contrary to what I said in lecture, such compact representations are actually more difficult in structural logic since they preclude direct access to the middle of these blocks, or they require new messages to be created when one is received. In the linear case, this would be less problematic than here.

Nevertheless, there is a logical technique called *focusing* that may justify big blocks of messages. In fact, focusing will be the subject of the next few lectures in this course.

3 Assigning Processes to Proofs for Negative Propositions

The negative propositions in the structural fragments are $A^+_{\mathsf{U}} \to B^-_{\mathsf{U}}$, $\uparrow^{\mathsf{U}}_{\mathsf{L}} A^+_L$, and $A^-_{\mathsf{U}} \mathbin{\&} B^-_{\mathsf{U}}$. The pattern for $\uparrow^{\mathsf{U}}_{\mathsf{L}} A_L$ in the last lecture was different from what we saw above for positive connectives. Essentially, a persistent process of type $\uparrow A$ just waited to receive a shift to a fresh linear channel c and then spawned a fresh copy of itself which offered along c. When we had only one proposition in the structural layer, this was sufficient. Here, we have to ask how we obtain a persistent process in the first place. Not every process can be persistent, since processes, whether they offer along a persistent channel or not, must be able to make progress in their computation. As far as I can see, this problem is best solved by having another type of semantic object in addition to proc and msg which is a persistent service srvc. We may view msg and srvc as duals, where msg sends while srvc receives.

But, first, the one-premise version of the usual rules.

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ {\to}R \qquad \frac{\Gamma, A, A \to B, B \vdash C}{\Gamma, A, A \to B \vdash C}\ {\to}L^*$$

When assigning process terms, sending is somewhat tricky. We have to send a $w{:}A$ along a channel $x{:}A \to B$ and also a continuation channel $x'{:}B$.

$$\frac{\Gamma, w{:}A, x{:}A \to B, x'{:}B \vdash Q :: (z : C)}{\Gamma, w{:}A, x{:}A \to B \vdash x' \leftarrow \mathsf{send}\ x\ w\ ;\ Q :: (z : C)}\ {\to}L^*$$

The syntax here might suggest that we pass w to x and receive an x', but sending is actually asynchronous and we send a continuation channel for x' as well. We can then later communicate along that new channel to communicate further with the recipient process. The right rule is simpler by comparison.

$$\frac{\Gamma, y{:}A \vdash P :: (x : B)}{\Gamma \vdash y \leftarrow \mathsf{recv}\ x\ ;\ P :: (x : A \to B)}\ {\to}R$$

Computationally, receiving along a persistent channel with potentially many clients means that we create a persistent service.

$$\frac{\mathsf{proc}(x, y \leftarrow \mathsf{recv}\ x\ ;\ P)}{\underline{\mathsf{srvc}}(x, y \leftarrow \mathsf{recv}\ x\ ;\ P)}\ {\to}C\_\mathsf{srvc}$$

Now send and recv use a linear message.

$$\frac{\mathsf{proc}(z, x' \leftarrow \mathsf{send}\ x\ w\ ;\ Q)}{\mathsf{msg}(c, x' \leftarrow \mathsf{send}\ x\ w\ ;\ c \leftarrow x') \quad \mathsf{proc}(z, [c/x']Q)}\ \rightarrow C\_\mathsf{send}^c$$

$$\frac{\underline{\mathsf{srvc}}(x, y \leftarrow \mathsf{recv}\ x\ ;\ P) \quad \mathsf{msg}(c, x' \leftarrow \mathsf{send}\ x\ w\ ;\ c \leftarrow x')}{\mathsf{proc}(c, [c/x][w/y]P)}\ \rightarrow C\_\mathsf{recv}$$

We leave $\&\{l_i : A_i\}_{i \in I}$ to Exercise 1.

4 Examples: Map and Finite Differences

We use two simple examples to illustrate programming. The first is to map a process of type $A \rightarrow B$ over a list. The second computes a list of differences between elements of a given list.

$\mathsf{list}_A = +\{\mathsf{cons} : A \times \mathsf{list}_A, \mathsf{nil} : \mathbf{1}\}$

$f{:}A \rightarrow B, k{:}\mathsf{list}_A \vdash \mathit{map} :: (l : \mathsf{list}_B)$

$l \leftarrow \mathit{map} \leftarrow f\ k =$
  $\mathsf{case}\ k\ \ (\mathsf{nil}(k_1) \Rightarrow \mathsf{wait}\ k_1\ ;\ l.\mathsf{nil}\ ;\ \mathsf{close}\ l$
  $\quad\quad | \ \mathsf{cons}(k_1) \Rightarrow (x, k_2) \leftarrow \mathsf{recv}\ k_1$
  $\quad\quad\quad\quad\quad\quad\quad\quad y \leftarrow \mathsf{send}\ f\ x$
  $\quad\quad\quad\quad\quad\quad\quad\quad l.\mathsf{cons}\ ;\ \mathsf{send}\ l\ y$
  $\quad\quad\quad\quad\quad\quad\quad\quad l \leftarrow \mathit{map} \leftarrow f\ k)$

For the second example we use the type of integer lists and an existential quantifiers $\exists x{:}\mathsf{int}.\ \mathsf{intlist}$, abbreviated as $\mathsf{int} \wedge \mathsf{intlist}$. Note that it satisfies the same rules as $A \times B$, except it sends an integer instead of a channel of type A.

Our example takes a list of integers and computes the list of differences between successive integers, which will be one element shorter unless the given list is already empty. We avoid further syntactic sugar, which should not be too difficult to imagine.

$\mathsf{intlist} = +\{\mathsf{cons} : \mathsf{int} \wedge \mathsf{intlist}, \mathsf{nil} : \mathbf{1}\}$

$k{:}\mathsf{intlist} \vdash \mathit{diffs} :: (l : \mathsf{intlist})$

$l \leftarrow \mathit{diffs} \leftarrow k =$
  $\mathsf{case}\ k\ \ (\mathsf{nil}(k_1) \Rightarrow \mathsf{wait}\ k_1\ ;\ l.\mathsf{nil}\ ;\ \mathsf{close}\ l$

$$
\begin{array}{l}
\mid \mathsf{cons}(k_1) \Rightarrow (y, k_2) \leftarrow \mathsf{recv}\ k_1\ ; \\
\qquad\qquad \mathsf{case}\ k_2\ \ (\mathsf{nil}(k_3) \Rightarrow \mathsf{wait}\ k_3\ ;\ l.\mathsf{nil}\ ;\ \mathsf{close}\ l \\
\qquad\qquad\qquad\qquad \mid \mathsf{cons}(k_3) \Rightarrow (z, k_4) \leftarrow \mathsf{recv}\ k_3\ ; \\
\qquad\qquad\qquad\qquad\qquad\qquad l.\mathsf{cons}\ ;\ \mathsf{send}\ l\ (z - y)\ ; \\
\qquad\qquad\qquad\qquad\qquad\qquad l \leftarrow \mathit{diffs} \leftarrow k_2))
\end{array}
$$

A key aspect of this example, which makes it non-linear, is that the recursive call to *diffs* is passed k_2, which is the tail of k, rather than k_4, which is the tail of the tail (and which is is ignored). This structure arises because we look ahead one element in the list to compute the difference.

5 Upshift, Revisited

In the more general setting of this lecture, we revise the computation rules for the up modality slightly, taking advantage of the $\underline{\mathsf{srvc}}$ predicate.

$$\frac{\mathsf{proc}(z_{\mathsf{L}}, y_{\mathsf{L}} \leftarrow (\mathsf{L})x_{\mathsf{U}}\ ;\ Q_y)}{\mathsf{msg}(w_{\mathsf{L}}, y_{\mathsf{L}} \leftarrow (\mathsf{L})x_{\mathsf{U}}\ ;\ w_{\mathsf{L}} \leftarrow y_{\mathsf{L}}) \quad \mathsf{proc}(z_{\mathsf{L}}, Q_w)}\ \uparrow C^w\_\mathsf{send}$$

$$\frac{\mathsf{proc}(x_{\mathsf{U}}, y_{\mathsf{L}} \leftarrow (\mathsf{L})x_{\mathsf{U}}\ ;\ P_y)}{\underline{\mathsf{srvc}}(x_{\mathsf{U}}, y_{\mathsf{L}} \leftarrow (\mathsf{L})x_{\mathsf{U}}\ ;\ P_y)}\ \uparrow C\_\mathsf{srvc}$$

$$\frac{\underline{\mathsf{srvc}}(x_{\mathsf{U}}, y_{\mathsf{L}} \leftarrow (\mathsf{L})x_{\mathsf{U}}\ ;\ P_y) \quad \mathsf{msg}(w_{\mathsf{L}}, y_{\mathsf{L}} \leftarrow (\mathsf{L})x_{\mathsf{U}}\ ;\ w_{\mathsf{L}} \leftarrow y_{\mathsf{L}})}{\mathsf{proc}(w_{\mathsf{L}}, P_w)}\ \uparrow C^w\_\mathsf{recv}$$

Exercises

Exercise 1 Give the process term assignment and computation rules for $\&\{l_i : A_i\}_{i \in I}$ in structural logic.

Exercise 2 Prove the logical equivalence between $A \& B$ and $A \times B$ in structural logic. The write out the processes

$p{:}A \,\&\, B \vdash \mathit{back} :: (q : A \times B)$
$q{:}A \times B \vdash \mathit{forth} :: (p : A \,\&\, B)$

where $A \,\&\, B = \&\{\mathsf{inl} : A, \mathsf{inr} : B\}$ and the proof term assignment and computation rules come from Exercise 1.

Can you say succinctly what these two processes do?

References

[Her94] Hugo Herbelin. A lambda-calculus structure isomorphic to Gentzen-style sequent calculus structure. In L. Pacholski and J. Tiuryn, editors, *8th International Workshop on Computer Science Logic*, pages 61–75, Kazimierz, Poland, September 1994. Springer LNCS 933.

[How69] W. A. Howard. The formulae-as-types notion of construction. Unpublished note. An annotated version appeared in: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, 479–490, Academic Press (1980), 1969.