Lecture Notes on Ordered Forward Chaining

15-816: Linear Logic
Frank Pfenning

Lecture 16
March 21, 2012

In the last lecture we saw ordered logic, which is in some sense even more primitive than linear logic. We also saw a focusing system for it, which is the basis for logic programming. We did not prove the completeness of focusing with respect to the sequent calculus—it follows the same pattern as for linear logic.

In this lecture we will develop some examples of forward chaining ordered logic programs. One example encodes Turing machines, another extends the ideas behind substructural operational semantics to the ordered case, and a third concerns binary arithmetic.

1 Ordered Forward Chaining

Forward chaining interprets proofs search as a form of committed-choice operational semantics. This can be justified by deriving complete sets of rules from certain kinds of unrestricted hypotheses, collected in a *program* of the form Γ_P. This is just like we developed in some detail in Lecture 13 except that now order must be preserved amont the ordered resources. A state transition now has the form

$$(\Psi \,;\, \Gamma \,;\, \Delta \,;\, \Omega) \longrightarrow (\Psi' \,;\, \Gamma' \,;\, \Delta' \,;\, \Omega')$$

where $\Psi \subseteq \Psi'$ contains parameters, $\Gamma \subseteq \Gamma'$ unrestricted resources, Δ and Δ' linear resources, and Ω and Ω' ordered resources.

To permit such an operational interpretation, the program clauses and goals should be drawn from the following grammar. This can be obtained

by analyzing the focusing system in the style of Andreoli (and therefore including the so-called atom optimization of Lecture 15).[1]

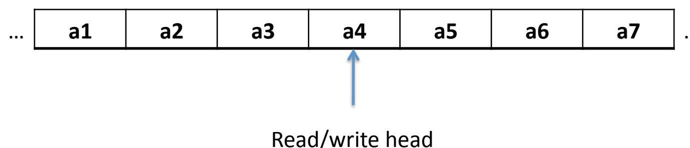
$$\begin{array}{llll} \text{Clauses} & D & ::= & \forall x{:}\tau.D \mid D_1 \mathbin{\&} D_2 \mid \top \mid G \twoheadrightarrow D \mid G \rightarrowtail D \mid H \\ \text{Heads} & H & ::= & P^+ \mid H_1 \bullet H_2 \mid H_1 \circ H_2 \mid \mathbf{1} \mid \exists x{:}\tau.H \mid H_1 \oplus H_2 \mid \mathbf{0} \mid ¡D \mid !D \\ \text{Goals} & G & ::= & P^+ \mid G_1 \bullet G_2 \mid G_1 \circ G_2 \mid \mathbf{1} \mid \exists x{:}\tau.G \mid G_1 \oplus G_2 \mid \mathbf{0} \; [\mid ¡P^+ \mid !P^+] \end{array}$$

The last two are not strictly conformant, but they are helpful for expressing whether a positive atomic proposition P^+ is supposed to be in the ordered, linear, or unrestricted context. They could just be written as P^+, or we could introduce additional focusing rules that allow them.

2 Turing Machines

In honor of the 100th birthday of Alan Turing[2] this year, we start with an encoding of Turing machines in ordered logic. We will engineer this representation so that forward chaining corresponds to the computation of a Turing machine.

Be begin by considering the tape, which we assume is two-way infinite. Because the tape has a definite notion of order and adjacency, it makes sense to represent a tape with symbols



as an ordered context

$$\Omega = (a_1, a_2, a_3, a_4, a_5, a_6, a_7)$$

where each symbol is represented as an atomic proposition. To encode a potentially infinite tape in a finite ordered context, we add a special propositions end ("end") to mark the left and right end of the explored tape.

$$\Omega = (\mathsf{end}, a_1, a_2, a_3, a_4, a_5, a_6, a_7, \mathsf{end})$$

[1] The Ollibot implementation implements a slightly different fragment.

[2] June 23, 1912 – June 7, 1954

How do we represent the tape head? It would make sense to use a new atomic proposition h to represent the position of the head, except that it would have to be either to the left or right of the symbols that it reads. Rather than fixing one or the other, we allow *both*, and have two different special proposition representing the tape head: left for the left-looking head, and right for the right-looking head. Therefore the tape above, while reading a_4, could be represented either as

$$\Omega = (\mathsf{end}, a_1, a_2, a_3, a_4, \mathsf{left}, a_5, a_6, a_7, \mathsf{end})$$

or

$$\Omega = (\mathsf{end}, a_1, a_2, a_3, \mathsf{right}, a_4, a_5, a_6, a_7, \mathsf{end})$$

A Turing machine has a special blank symbol, which we write as b. If we are looking right and see the end of the tape, we create a new square containing the blank symbol, without moving the tape head.

$$\mathsf{right} \bullet \mathsf{end} \twoheadrightarrow \mathsf{right} \bullet \mathsf{b} \bullet \mathsf{end}$$

We have a symmetric rule if we are the left end, looking left.

$$\mathsf{end} \bullet \mathsf{left} \twoheadrightarrow \mathsf{end} \bullet \mathsf{b} \bullet \mathsf{left}$$

Before we move on to encode the transitions themselves, let's think about why we wrote $A \twoheadrightarrow B$ instead of $A \rightarrowtail B$ or $A \multimap B$ (where the latter could be written as $¡A \twoheadrightarrow B$ or $¡A \rightarrowtail B$). Note that the rules above are *unrestricted* or *persistent* resources, so they are used via the ordered copy rule:

$$\frac{\Gamma \,;\, \Delta \,;\, \Omega_L, [A], \Omega_R \rightarrow C \quad A \in \Gamma}{\Gamma \,;\, \Delta \,;\, \Omega_L, \Omega_R \rightarrow C}\ \mathsf{copy}$$

This rule can place A somewhere in the ordered context. So if we have a persistent clause $a_1 \bullet a_2 \twoheadrightarrow b_1 \bullet b_2$ somewhere in Γ_p, we must place it to the *left* of the atoms a_1 and a_2:

$$\dfrac{\dfrac{\Gamma_p \,;\, \cdot \,;\, a_1, a_2 \rightarrow [a_1 \bullet a_2] \quad \Gamma_p \,;\, \Delta \,;\, \Omega_L, [b_1 \bullet b_2], \Omega_R \rightarrow C}{\Gamma_p \,;\, \Delta \,;\, \Omega_L, [a_1 \bullet a_2 \twoheadrightarrow b_1 \bullet b_2], a_1, a_2, \Omega_R \rightarrow C}\ \mathsf{rimp}L}{\Gamma_p \,;\, \Delta \,;\, \Omega_L, a_1, a_2, \Omega_R \rightarrow C}\ \mathsf{copy}$$

But if we formulate the clause with a *left implication* we can just place it to the *right* of the propositions in question, resulting in the same premises.

$$\dfrac{\dfrac{\Gamma_p \,;\, \cdot \,;\, a_1, a_2 \rightarrow [a_1 \bullet a_2] \quad \Gamma_p \,;\, \Delta \,;\, \Omega_L, [b_1 \bullet b_2], \Omega_R \rightarrow C}{\Gamma_p \,;\, \Delta \,;\, \Omega_L, a_1, a_2, [a_1 \bullet a_2 \rightarrowtail b_1 \bullet b_2], \Omega_R \rightarrow C}\ \mathsf{limp}L}{\Gamma_p \,;\, \Delta \,;\, \Omega_L, a_1, a_2, \Omega_R \rightarrow C}\ \mathsf{copy}$$

This exemplifies the observation that a for a persistent clause with a top-level ordered implication, it does not matter if it is written as a left implication or a right implication. The same is true for linear clauses, since they also can be placed at an arbitrary position in the ordered context.

We prefer to use right implications, because we are used to write new antecedents of sequents at the right end on the context.

A Turing machine program has a finite number of states $q_0, \ldots, q_{n-1}$. In each state, for each symbol on the tape at the position of the tape head, a particular action is described. The action has three parts: writing a symbol back to the tape, moving the tape head left or right, and going into a next state.

For each state q_i of a machine we have an atomic proposition q_i. In order not to interfere with the contents of the tape, we keep the state as the only proposition in the *linear* context.

For example, in state q_1 we might read the symbol 0, write 1, move to the right, and enter state q_2. If we face to the right, this would be represented as

$$\text{¡}\mathsf{q}_1 \bullet \mathsf{right} \bullet \mathsf{0} \twoheadrightarrow \text{¡}\mathsf{q}_2 \bullet \mathsf{1} \bullet \mathsf{right}$$

If we face to the left the move is represented simply by a reversal of the direction.

$$\text{¡}\mathsf{q}_1 \bullet \mathsf{0} \bullet \mathsf{left} \twoheadrightarrow \text{¡}\mathsf{q}_2 \bullet \mathsf{1} \bullet \mathsf{right}$$

In both cases, q_1 in the premise is guaranteed to be in the linear context since the state always has the form

$$\Gamma_p \; ; \; \mathsf{q}_i \; ; \; \mathsf{end}, a_{-k}, \ldots, a_0, \mathsf{left}, \ldots, a_j, \mathsf{end}$$

or

$$\Gamma_p \; ; \; \mathsf{q}_i \; ; \; \mathsf{end}, a_{-k}, \ldots, \mathsf{right}, a_0, \ldots, a_j, \mathsf{end}$$

Γ_p are the clauses representing the Turing machine program, q_i is the current state, the tape has symbols $a_{-k}, \ldots, a_j$ (and blanks everywhere else) and the head reads a_0.

If the transition function is represented in this manner, with each possible transition being represented by two clauses, computation steps of the Turing machine correspond almost exactly to forward chaining steps. The only difference is due to the steps that extend the tape on either end, which happen silently in Turing's definition. We could obtain an exact correspondence of computation steps by adding more clauses (see Exercise 1).

Assume we have an input $a_0, \ldots a_{n-1}$. We start the machine in the *starting state* q_0 and the following configuration

$$\Gamma_p \; ; \; \mathsf{q}_0 \; ; \; \mathsf{end}, \mathsf{right}, a_0, \ldots a_{n-1}, \mathsf{end}$$

where Γ_p represents the transition function as sekteched above. A Turing machine also has a set of *final* or *accepting* states, say, $f_0, \ldots, f_{k-1}$. We want to be able to prove halt if we have reached a final state. This is encoded by the clause

$$¡f_0 \oplus \cdots \oplus ¡f_{k-1} \twoheadrightarrow ¡\mathsf{halt}$$

Then, overall, the Turing machine succeeds if and only if

$$\Gamma_p \; ; \; \mathsf{q}_0 \; ; \; \mathsf{end}, a_0, \ldots, a_{n-1}, \mathsf{end} \rightarrow ¡\mathsf{halt} \bullet \top$$

where we use $\top$ to ensure that the non-empty tape contents does not prevent the halt from being proven.

Since the halting problem for Turing machines is undecidable, this demonstrates that the theorem proving problem for propositional ordered logic is also undecidable. This is in contrast to traditional classical or even intuitionistic propositional logics, which are decidable. Linearity by itself is already enough to prove undecidability [LMSS92].

3 Ordered Substructural Operational Semantics

Can we exploit order in substructural operational semantics? The answer is "*yes*": many specifications become more concise. As an example, we revisit the operational semantics of our earlier linear λ-calculus. In many parts of this specification, we can replace uses of destinations simply by exploiting order [PS09].

We start with a version of evaluation where we use explicit substitutions for variables, instead of binding variables to values. This is for a call-by-value language.

$$\begin{array}{l}
\mathsf{eval}(\lambda x.M_x, d) \multimap \mathsf{retn}(\lambda x.M_x, d) \\
\mathsf{eval}(M\,N, d) \multimap \exists d_1.\mathsf{eval}(M, d_1) \otimes \mathsf{cont}(d_1, _\,N, d) \\
\mathsf{retn}(V_1, d_1) \otimes \mathsf{cont}(d_1, _\,N, d) \multimap \exists d_2.\mathsf{eval}(N, d_2) \otimes \mathsf{cont}(d_2, V_1\,_, d) \\
\mathsf{retn}(V_2, d_2) \otimes \mathsf{cont}(d_2, (\lambda x.M_x)\,_, d) \multimap \mathsf{eval}([V_2/x]M_x, d)
\end{array}$$

We assume an initial state $\mathsf{eval}(M, d_0)$ and expect a final state $\mathsf{retn}(V, d_0)$. Then the linear state always has one of the following two forms:

$$\begin{array}{l}
\mathsf{eval}(M, d_n), \mathsf{cont}(d_n, f_n, d_{n-1}), \mathsf{cont}(d_{n-1}, f_{n-1}, d_{n-2}), \ldots, \mathsf{cont}(d_1, f_1, d_0) \\
\mathsf{retn}(V, d_n), \mathsf{cont}(d_n, f_n, d_{n-1}), \mathsf{cont}(d_{n-1}, f_{n-1}, d_{n-2}), \ldots, \mathsf{cont}(d_1, f_1, d_0)
\end{array}$$

This state invariant is easy to check by going through each of the rules. The first rule transitions from a state of the first kind to the second. The second rule creates a new continuation, but otherwise stays within the first of the possible kinds of state. The third rule transitions from a state of the second kind to the first, and so does the last rule.

Notice in particular that the threading of destinations through the continuations is precisely as described: continuations form a stack, whose structure is maintained by a threaded sequence of continuations. Using order, we can eliminate these destinations and use the relative position of the propositions instead. Then the *ordered* state will be one of

$$\begin{array}{l} \mathsf{eval}(M), \mathsf{cont}(f_n), \mathsf{cont}(f_{n-1}), \ldots, \mathsf{cont}(f_1) \\ \mathsf{retn}(V), \mathsf{cont}(f_n), \mathsf{cont}(f_{n-1}), \ldots, \mathsf{cont}(f_1) \end{array}$$

and the linear clauses above become the following ordered clauses:

$$\begin{array}{l} \mathsf{eval}(\lambda x.M_x) \twoheadrightarrow \mathsf{retn}(\lambda x.M_x) \\ \mathsf{eval}(M\,N) \twoheadrightarrow \mathsf{eval}(M) \bullet \mathsf{cont}(_\,N) \\ \mathsf{retn}(V_1) \bullet \mathsf{cont}(_\,N) \twoheadrightarrow \mathsf{eval}(N) \bullet \mathsf{cont}(V_1\,_) \\ \mathsf{retn}(V_2) \bullet \mathsf{cont}((\lambda x.M_x)\,_) \twoheadrightarrow \mathsf{eval}([V_2/x]M_x) \end{array}$$

This is generally referred to as a *stack machine*, implemented here in an extremely concise manner in the forward-chaining fragment of ordered logic.

In order to avoid the explicit appeal to substitution, we can combine this with the use of destinations (also called channels or variables) by using mobility. For this, we need a mobile version of return with a second argument, which we write here as retn_2. This change replaces the last rule above by the last two rules below.

$$\begin{array}{l} \mathsf{eval}(\lambda x.M_x) \twoheadrightarrow \mathsf{retn}(\lambda x.M_x) \\ \mathsf{eval}(M\,N) \twoheadrightarrow \mathsf{eval}(M) \bullet \mathsf{cont}(_\,N) \\ \mathsf{retn}(V_1) \bullet \mathsf{cont}(_\,N) \twoheadrightarrow \mathsf{eval}(N) \bullet \mathsf{cont}(V_1\,_) \\[4pt] \mathsf{retn}(V_2) \bullet \mathsf{cont}((\lambda x.M_x)\,_) \twoheadrightarrow \exists x.\, ¡\mathsf{retn}_2(V, x) \bullet \mathsf{eval}(M_x) \\ \mathsf{eval}(x) \bullet ¡\mathsf{retn}_2(V, x) \twoheadrightarrow \mathsf{retn}(V) \end{array}$$

If variables are not necessarily linear, we can use persistent returns instead, replacing the last two rules by:

$$\begin{array}{l} \mathsf{retn}(V_2) \bullet \mathsf{cont}((\lambda x.M_x)\,_) \twoheadrightarrow \exists x.\, !\mathsf{retn}_2(V, x) \bullet \mathsf{eval}(M_x) \\ \mathsf{eval}(x) \bullet !\mathsf{retn}_2(V, x) \twoheadrightarrow \mathsf{retn}(V) \end{array}$$

At this point, it should be relatively straightforward to complete refinement of the earlier semantics to exploit order so as to avoid a proliferation of

destinations. Some care is nevertheless necessary. For example, it may be tempting to recover some parallelism in evaluation with rules like the following:

$$\begin{array}{l} \mathsf{eval}(\lambda x.M_x) \twoheadrightarrow \mathsf{retn}(\lambda x.M_x) \\ \mathsf{eval}(M\,N) \twoheadrightarrow \mathsf{eval}(M) \bullet \mathsf{eval}(N) \\ \mathsf{retn}(\lambda x.M_x) \bullet \mathsf{retn}(V) \twoheadrightarrow \mathsf{eval}([V/x]M_x) \end{array}$$

The problem here is that the ordered conjunction is still associative, so if we had $\mathsf{eval}((M_1\,M_2)\,N)$, it could turn into $\mathsf{eval}(M_1), \mathsf{eval}(M_2), \mathsf{eval}(N)$ in two steps. Now a value returned by N might interact with a value returned by M_2, which is incorrect. It may be possible to use this kind of representation if we reject associativity as well, in which case the context will have a tree-shaped form, but it does not seem particularly straightforward to define the right modalities to recover order, linearity, or persistence.

4 Subcomputations

When programming with functions, computations naturally decompose into subcomputations. For example, under call-by-value, an expression such as $f(g(x))$ will first compute $g(x)$ to a value v and then compute $f(v)$. The arguments passed (here only x) and the value returned (here v) constitute the interface to a well-defined subcomputation.

In a forward-chaining operational semantics, whether it is persistent, linear, or ordered, a decomposition of an overall computation into subcomputations is much less obvious. We have a global state, and the semantics dictates that any any point during the computation, any rule that applies can fire, possibly consuming part of the state and replacing it with something new. This models concurrency and parallelism quite well, but it complicates reasoning about programs modularly or clearly identifying subcomputations.

As an example, imagine we would like to extend our little functional language with natural numbers in binary representation and some operations such as addition. We use the syntax

$$\text{Nats} \quad n \quad ::= \quad \epsilon \mid n\mathsf{0} \mid n\mathsf{1}$$

where we interpret bitstrings n as numbers $\llcorner n \lrcorner$ as follows:

$$\begin{array}{rcl} \llcorner \epsilon \lrcorner & = & 0 \\ \llcorner n\mathsf{0} \lrcorner & = & 2 * \llcorner n \lrcorner \\ \llcorner n\mathsf{1} \lrcorner & = & 2 * \llcorner n \lrcorner + 1 \end{array}$$

In other words, we represent a natural number as a bitstring with the least significant bit written last. Bitstring are not unique, because we can always add leading zeroes without changing its value. In an explicit term representation we might write $n0$ as $\mathsf{zero}(n)$ or as $\mathsf{bit}(n, 0)$.

In order to evaluate addition, we have to evaluate the two summands to number first:

$$\begin{array}{l}\mathsf{eval}(M + N) \twoheadrightarrow \mathsf{eval}(M) \bullet \mathsf{cont}(_ + N)\\ \mathsf{retn}(m) \bullet \mathsf{cont}(_ + N) \twoheadrightarrow \mathsf{eval}(N) \bullet \mathsf{cont}(m + _)\\ \mathsf{retn}(n) \bullet \mathsf{cont}(m + _) \twoheadrightarrow \mathsf{plus}(m, n)\end{array}$$

Now we need some way to compute $\mathsf{plus}(m, n)$ for bit strings m and n. We can define this based on the structure of m and n.

$$\begin{array}{l}\mathsf{plus}(\epsilon, n) \twoheadrightarrow \mathsf{retn}(n)\\ \mathsf{plus}(n, \epsilon) \twoheadrightarrow \mathsf{retn}(n)\\ \mathsf{plus}(n0, m0) \twoheadrightarrow \mathsf{plus}(n, m) \bullet \mathsf{cont}(_0)\\ \mathsf{plus}(n0, m1) \twoheadrightarrow \mathsf{plus}(n, m) \bullet \mathsf{cont}(_1)\\ \mathsf{plus}(n1, m0) \twoheadrightarrow \mathsf{plus}(n, m) \bullet \mathsf{cont}(_1)\\ \mathsf{plus}(n1, m1) \twoheadrightarrow \mathsf{plus}(n, m) \bullet \mathsf{cont}(\mathsf{inc}\ _) \bullet \mathsf{cont}(_0)\end{array}$$

The first two clauses overlap when $n = \epsilon$, but since we are working in a committed choice forward-chaining language, this might be acceptable.

In the last line, the continuation $\mathsf{cont}(\mathsf{inc}\ _)$ instructs the computation to increment the result of adding n and m in order to account for the carry that arises from adding the two 1 bits. The continuations work as follows:

$$\begin{array}{l}\mathsf{retn}(n) \bullet \mathsf{cont}(_0) \twoheadrightarrow \mathsf{retn}(n0)\\ \mathsf{retn}(n) \bullet \mathsf{cont}(_1) \twoheadrightarrow \mathsf{retn}(n1)\\ \\ \mathsf{retn}(n) \bullet \mathsf{cont}(\mathsf{inc}\ _) \twoheadrightarrow \mathsf{inc}(n)\\ \\ \mathsf{inc}(\epsilon) \twoheadrightarrow \mathsf{retn}(\epsilon 1)\\ \mathsf{inc}(n0) \twoheadrightarrow \mathsf{retn}(n1)\\ \mathsf{inc}(n1) \twoheadrightarrow \mathsf{inc}(n) \bullet \mathsf{cont}(_0)\end{array}$$

Instead of an explicit increment operation, we could also add another argument bit to plus indicating the presence or absence of a carry (see Exercise 4).

The state of computation is now a little harder to characterize, because we either are performing ordinary evaluation steps, or we are in the middle of an addition or an increment. Also, the steps of the addition or increment operations can be interleaved with other operations, in case there is any

parallelism in the language. If possible, we would like to specify addition as an *atomic step*. That is, we would like to replace

$$\mathsf{retn}(n) \bullet \mathsf{cont}(m + _) \twoheadrightarrow \mathsf{plus}(m, n)$$

by something like

$$\mathsf{retn}(n) \bullet \mathsf{cont}(m + _) \bullet \mathsf{plus}(m, n, r) \twoheadrightarrow \mathsf{retn}(r)$$

where $\mathsf{plus}(m, n, r)$ is now a three-place relation that holds iff $m + n = r$. Under the pure forward-chaining semantics we have worked with so far, this would be problematic because we would have to have potentially infinitely many propositions $\mathsf{plus}(m, n, r)$ in the context.

Fortunately, we can exploit the flexibility behind polarity and focusing and make $\mathsf{plus}(m, n, r)$ a *negative* atom. If we play through the focusing rules, we see that focusing on the above clause leads to the following derived rule:

$$\frac{\cdot \rightarrow \mathsf{plus}^-(m, n, r) \quad \Omega_L, \mathsf{retn}^+(r), \Omega_R \rightarrow C}{\Omega_L, \mathsf{retn}^+(n), \mathsf{cont}^+(m + _), \Omega_R \rightarrow C}$$

We have elided there Γ and Δ and also assumed that the proof of $\mathsf{plus}(m, n, r)$ requires no ordered or linear resources.

In the next lecture we will see how to write the $\mathsf{plus}^-(m, n, r)$ predicate so that it properly computes r from m and n under a *backward-chaining* operational semantics.

Exercises

Exercise 1 Modify the encoding of Turing machine transitions so that we can eliminate the two generic rules that extend the tape head. Make sure your encoding still only uses the propositional fragment, that is, no variables or quantifiers.

Exercise 2 Modify the undecidability proof for propositional ordered logic to show that propositional linear logic is also undecidable.

Exercise 3 Develop *non-associative logic* in which we also reject the law of associativity. Non-associative antecedents would form a kind of tree structure. Develop enough of the proof theory to see what kinds of implications, conjunctions, and modalities might be available. Use your logic to specify parallel evaluation, as attempted at the end of Section 3.

Exercise 4 Rewrite the forward-chaining code for addition to that it takes the carry bit as another argument and avoid explicit uses of increment.

Exercise 5 Rewrite the code for addition assuming a unary representation of numbers as we have introduced it in earlier lectures.

Exercise 6 Rewrite the code for addition assuming two's complement arbitrary precision arithmetic. In this representation, the most significant bit indicates the sign of the number. We can always replicate the most significant bit without changing the value of a number.

References

[LMSS92] Patrick Lincoln, John Mitchell, Andre Scedrov, and Natarajan Shankar. Decision problems for propositional linear logic. *Annals of Pure and Applied Logic*, 56:239–311, April 1992.

[PS09] Frank Pfenning and Robert J. Simmons. Substructural operational semantics as ordered logic programming. In *Proceedings of the 24th Annual Symposium on Logic in Computer Science (LICS 2009)*, pages 101–110, Los Angeles, California, August 2009. IEEE Computer Society Press.