Lecture Notes on Substructural Operational Semantics

15-317: Constructive Logic
André Platzer

Lecture 23
April 23, 2020

1 Introduction

This lecture is an excursion on using linear logic (or other substructural logics) for specifying the operational semantics of programming languages, which is called substructural operational semantics (SSOS) [1]. The lecture could hardly claim to be an exhaustive overview of the interesting field of substructural operational semantics. But it constitutes an interesting outlook of what applications linear logic can be good for and simultaneously motivates the study of other substructural logics that are more general than linear logic. By necessity, today's lecture will not end up spelling out all the details needed to comprehensively cover real languages. Instead, we will be exploring the use of linear logic to illustrate the main idea of substructural operational semantics, while also indicating places where generalizations of linear logic are needed. This is where substructural logics come in, the ones that fulfill only a subset of the structural rules of weakening, contraction, and cut. Linear logic satisfies none of them (except in its modified form of a linear cut) although it still accepts permutation of assumptions, while other substructural logics satisfy some but not all of them.

2 The Idea of Substructural Operational Semantics

Operational semantics captures the behavior of programming languages as transition systems describing how the state changes over time as the program executes one step at a time. The starting point of substructural

operational semantics is the observation that if we can capture a state of a program as a logical formula, then all we need is a way of describing how that logical formula changes as a function of the steps that the program does. Linear logic is good at that, because it can describe what becomes true even if it wasn't true before. And linear logic can even describe what stops being true even if it used to be true before. That way we can represent the transitions of a program in linear logic.

That brings up the question how to best represent the particular transitions of a program in linear logic. At this point of the course should it be immediately obvious how we could start developing new proof rules that capture the transitions of truth that a particular given program performs. Just think back of the idea of linear logic programming or, in fact, even just intuitionistic logic programming either in forward or backward-chaining style.

Yet, that would merely represent the semantics of a particular program in linear logic. We're after something bigger: representing the semantics of an entire programming language in linear logic, so once and for all the meaning of running any arbitrary program of that programming language. That means that we should leave the proof rules of linear logic untouched (or at least only add a fixed set of rules) and then represent any arbitrary program as input formulas. Linear logical implications with the lolli operator $\multimap$ are useful to describe state transitions, because using the linear implication $A \multimap B$ describes the process of consuming resource A (so it is not true anymore) and producing resource B (so it is true from now on until further notice).

Consequently, the basic idea of substructural operational semantics is to describe the operational semantics of a programming language as a transition system that is internalized in a logic of change (such as linear logic).

3 Toward Semantics for Imperative Programs

Linear logic is a logic for state change, but how does it fare for representing genuine imperative programming languages? Let us consider a simple imperative programming language:

$$\alpha \ ::= \ X := N \mid X := Y \mid \text{if } (X = 0)\ \alpha \text{ else } \beta \mid \alpha; \beta$$

In this simple programming language, all program variables such as X, Y, Z begin with capital letters and N is reserved to stand for natural number literals such as 1, 5, or 317. The only control flow statement is an if-then-else

zero-test for a variable X. Without loops or recursion or interesting computations, this programming language is not even Turing-complete but that does not prevent us from using it to explore the use of substructural logics.

The idea we will pursue is that we will give a meaning to running a program α by defining linear logic propositions involving $\mathsf{step}(\alpha)$ that describe the program's effect in logic. For example, a proposition involving $\mathsf{step}(X := N)$ will describe how the state changes when assigning the number literal N to the variable X. The effect of running a number assignment $X := N$ is that N will be stored as the value of variable X in memory. Let us use $\mathsf{mem}(X, V)$ to represent that the value of variable X in memory is V. Even if, $\mathsf{mem}(X, N)$ was not true previously, running $X := N$ will make $\mathsf{mem}(X, N)$ true, as represented by the following linear logic proposition:

$$\mathsf{step}(X := N) \multimap \mathsf{mem}(X, N)$$

Fortunately, linear logic can make things true such as $\mathsf{mem}(X, N)$ that were not true before. Running a variable copying operation $X := Y$ will involve $\mathsf{step}(X := Y)$ but we also need to look up the present value V of variable Y in the memory to determine what value X will have:

$$\mathsf{mem}(Y, V) \otimes \mathsf{step}(X := Y) \multimap \mathsf{mem}(X, V)$$

That proposition says that X will from now on have value V. But since a use of this linear implication *consumed* the resource $\mathsf{mem}(Y, V)$ along the way, we would lose the value of the unaffected variable Y in the process, which would be a mistake. Correcting for that, we put its value back:

$$\mathsf{mem}(Y, V) \otimes \mathsf{step}(X := Y) \multimap \mathsf{mem}(X, V) \otimes \mathsf{mem}(Y, V)$$

Now thinking about what was true before, there is not only a concern to retain facts that continue to be true after a step, but also to manage what is no longer true afterwards. After assigning to X, the value of X in memory is no longer whatever it was before. If we were to adopt any of the above propositions, then X would end up having both its old and new value, which would brake the entire principle of program states. Fortunately, linear logic can not only represent what becomes true but also what stops being true. So, instead, we use linear propositions that consume the fact that X used to have some value W and, running either $X := N$ or $X := Y$, turns it into the fact that X now has its new value N, no longer the old one:

$$\mathsf{mem}(X, W) \otimes \mathsf{step}(X := N) \multimap \mathsf{mem}(X, N)$$
$$\mathsf{mem}(X, W) \otimes \mathsf{mem}(Y, V) \otimes \mathsf{step}(X := Y) \multimap \mathsf{mem}(X, V) \otimes \mathsf{mem}(Y, V)$$

This looks like a pretty promising model for describing the effect of assignments. But there is still a fundamental mistake. Can you spot it?

Think about it while we move on to the next operation: if-then-else. The effect of if-then-else clearly depends on whether its condition is true or not. In this simple language, the only conditions that can be used are direct evaluations of variables, which means that we can directly look up the resulting variable in memory:

$$\mathsf{mem}(X, 0) \otimes \mathsf{step}(\text{if } (X = 0)\ \alpha \text{ else } \beta) \multimap \mathsf{mem}(X, 0) \otimes \mathsf{step}(\alpha)$$
$$\mathsf{mem}(X, V) \otimes V \neq 0 \otimes \mathsf{step}(\text{if } (X = 0)\ \alpha \text{ else } \beta) \multimap \mathsf{mem}(X, V) \otimes \mathsf{step}(\beta)$$

Running if $(X = 0)\ \alpha$ else β will keep the value of the variable X unaffected and runs α if the value of X in memory is 0 and run β if it is a different value $V \neq 0$. For this purpose, we imagine we have access a proof rule that can just prove any comparison such as $V \neq 0$, so that we do not have to put back this knowledge after the transition. You are invited to check back with the lecture on the treatment of equality in forward logic programming to find out how to implement such comparisons.

Now that we are in the mode of thinking about two different possible effects of running the same program, you have probably spotted the problem with the assignment semantics already. For $X := X$, the linear implication erroneously produces a duplicate fact $\mathsf{mem}(X, V) \otimes \mathsf{mem}(X, V)$. Consequently, we should modify it to explicitly demand that X and Y are different variables (written $X \neq Y$):

$$X \neq Y \otimes \mathsf{mem}(X, W) \otimes \mathsf{mem}(Y, V) \otimes \mathsf{step}(X := Y) \multimap \mathsf{mem}(X, V) \otimes \mathsf{mem}(Y, V)$$

That now leaves the need to describe the semantics of a self-assignment $X := X$, which is easily done by simply discarding it without producing any resources:

$$\mathsf{step}(X := X) \multimap \mathbf{1}$$

Admittedly, incorrectly abusing the assignment $X := Y$ for self-assignment $X := X$ purposes would also require two redundant resources $\mathsf{mem}(X, W) \otimes \mathsf{mem}(X, V)$ as input, but it is still good practice to be explicit about the fact that X had better not be Y. Overall, it is crucial to retain the fact that every variable has exactly one value at any step during the substructural operational semantics transitions. Otherwise we would break all programs apart incorrectly.

All that is missing now is a way of running $\alpha; \beta$, which will first run α and then β. Summarizing the rules yields:

$$\begin{aligned}
\mathsf{mem}(X, W) \otimes \mathsf{step}(X := N) &\multimap \mathsf{mem}(X, N) \\
X \neq Y \otimes \mathsf{mem}(X, W) \otimes \mathsf{mem}(Y, V) \otimes \mathsf{step}(X := Y) &\multimap \mathsf{mem}(X, V) \otimes \mathsf{mem}(Y, V) \\
\mathsf{step}(X := X) &\multimap \mathbf{1} \\
\mathsf{mem}(X, 0) \otimes \mathsf{step}(\text{if } (X = 0)\ \alpha \text{ else } \beta) &\multimap \mathsf{mem}(X, 0) \otimes \mathsf{step}(\alpha) \\
\mathsf{mem}(X, V) \otimes V \neq 0 \otimes \mathsf{step}(\text{if } (X = 0)\ \alpha \text{ else } \beta) &\multimap \mathsf{mem}(X, V) \otimes \mathsf{step}(\beta) \\
\mathsf{step}(\alpha; \beta) &\multimap \mathsf{step}(\alpha) \otimes \mathsf{step}(\beta)
\end{aligned}$$

Of course, we may have to use these rules any number of times to understand the meaning of any particular imperative program. So the structural operational semantics of the simple imperative programming language could be represented as the conjunction of the exponentials of each of these linear implications.

This is almost right but not quite, and for very important reasons, too. Stepping $\alpha; \beta$ will correctly run α as well as β but not necessarily in the correct order. Of course, $\alpha; \beta$ will first run α and then, when done, run β.

This deficiency can be correct in multiple ways. The most convenient way is to switch from linear logic to ordered logic, which is a substructural logic in which the order of propositions is crucial. While linear logic tracks the exact multiplicity of its resources, only ordered logic also tracks their order, making it possible to express the following proposition with the ordered conjunction $\mathsf{step}(\alpha) \rtimes \mathsf{step}(\beta)$ that requires $\mathsf{step}(\alpha)$ to be provided *before* $\mathsf{step}(\beta)$ is considered:

$$\mathsf{step}(\alpha; \beta) \multimap \mathsf{step}(\alpha) \rtimes \mathsf{step}(\beta)$$

While this motivates ordered logic well, a less radical solution to correct the deficiency is to augment the semantics with markers that track when stepping through a program's semantics is completed. For simplicity, imagine no subprogram occurs twice and augment all of the above linear implications with an additional resource ... $\otimes \mathsf{done}(\alpha)$ to indicate that the processing of α is completed. Then the semantics of $\alpha; \beta$ can also be expressed purely in linear logic with a continuation:

$$\mathsf{step}(\alpha; \beta) \multimap \mathsf{step}(\alpha) \otimes (\mathsf{done}(\alpha) \multimap \mathsf{step}(\beta)) \otimes (\mathsf{done}(\beta) \multimap \mathsf{done}(\alpha; \beta))$$

This linear implication expresses that running $\alpha; \beta$ amounts to first running α and then, when α is done, running β and then, when β is also done, declaring that $\alpha; \beta$ is done. The simplifying assumption that no subprogram occurs twice can be lifted when working with program counters or

other extra machinery such as existential quantifiers or state-passing dependency parameters.

Finally note how nicely focusing on $\multimap$ implications has the effect that rewriting does not just proceed halfway through stepping but actually proceeds all the way.

4 Substructural Operational Semantics for Calculation

Terms in the simple imperative programming language considered above were hopelessly impoverished: only number literals or variables. As a replacement for that, let's consider a simplistic term language. For the sake of illustration consider a term language with addition and function abstraction and application.

$$T \ ::= \ N \mid X \mid T + S \mid \lambda y.T \mid TS$$

Again, we will use N for natural number literals and X, Y, Z for program variables. We already understood from previous lectures how we can track and distinguish types for the λ-abstraction $\lambda y.T$ to avoid the issues of untyped λ-calculus. Since this has been covered previously, we will shamelessly leave out types from our brief excursion into substructural operational semantics.

The primary purpose of terms in the context of the simplistic imperative programming language would be to serve as a generalization for the right-hand side of assignments $X := T$. Unlike with the impoverished variables and number literals from Section 3, we cannot directly look up the final value of a term in memory anymore but will have to evaluate it.

For this purpose, let us use a predicate $\mathsf{eval}(T, u)$ to express that we are in the process of evaluating term T and will put the resulting value into location u. You can think of u as the variable name for the result of the evaluation of T. At some point, we will have to be notified that the evaluation of the value for u completes with a resulting value V. We will use the predicate $\mathsf{retn}(V, u)$ for that. Then, assigning the value of term T to variable X in assignment $X := T$ has the effect of starting an evaluation of term T with result waiting to be put into u along with a linear implication that, when the evaluation for u returns, puts the resulting value V into the memory for X:[1]

$$\mathsf{step}(X := T) \multimap \mathsf{eval}(T, u) \otimes (\mathsf{retn}(V, u) \multimap \mathsf{mem}(X, V))$$

[1] A cleaner way of modeling this would be to use a universal quantifier for V around the second conjunct to be explicit about the fact that any resulting value V can be accepted. We

In order to also make the old memory value of X disappear, we could, by analogy to previous cases, try this:

$$\mathsf{mem}(X, W) \otimes \mathsf{step}(X := T) \multimap \mathsf{eval}(T, u) \otimes (\mathsf{retn}(V, u) \multimap \mathsf{mem}(X, V))$$

But that does not actually work correctly, because the memory value of X is then consumed when the assignment $X := T$ starts evaluating. It might still be needed during the evaluation of T. So instead, we delay the consumption of the old value of X till return time:

$$\mathsf{step}(X := T) \multimap \mathsf{eval}(T, u) \otimes (\mathsf{retn}(V, u) \otimes \mathsf{mem}(X, W) \multimap \mathsf{mem}(X, V))$$

The important takeaway for term evaluation going forward is merely its interface. We indicate the start of an evaluation of term T with result put into u via $\mathsf{eval}(T, u)$. And we indicate completion of the valuation with resulting value V for u via $\mathsf{retn}(V, u)$.

The easiest case to begin with is the evaluation of number literals, which immediately returns with said value:

$$\mathsf{eval}(N, u) \multimap \mathsf{retn}(N, u)$$

Left-to-right evaluation of a sum $T+S$ is best done by starting an evaluation of T and then leaving around an indication of the computation that needs to be continued once the former is completed. That is called a continuation, whose presence we will mark by a predicate $\mathsf{cont}(x, F, u)$ to say that we are waiting for a result on x, then carry out the remaining computation F, whose result is to be made available as u. By convention, we mark the place in F where the result of x enters the computation to be continued by the name $\_$, but F ultimately only uses helpful mnemonic notation for us.

Consequently, evaluating $T + S$ consists of three steps, 1) starting the evaluation of T with a continuation for $\_ + S$, then 2) reacting to a resulting value V of that by evaluating S with a continuation for the remaining $V + \_$, and finally 3) reacting to a resulting value W for the latter by returning the sum of V and W. The only slight subtlety is that a fresh name for the intermediate target of the evaluations is needed, for which existential quantifiers $\exists x$ and $\exists y$ are used, respectively:

$$\begin{aligned}
\mathsf{eval}(T + S, u) &\multimap \exists x.\mathsf{eval}(T, x) \otimes \mathsf{cont}(x, _ + S, u) \\
\mathsf{retn}(V, x) \otimes \mathsf{cont}(x, _ + S, u) &\multimap \exists y.\mathsf{eval}(S, y) \otimes \mathsf{cont}(y, V + _, u) \\
\mathsf{retn}(W, y) \otimes \mathsf{cont}(y, V + _, u) &\multimap \mathsf{retn}(V \text{ plus } W, u)
\end{aligned}$$

will refrain from doing so, because we have not studied quantifiers in linear logic yet. To avoid u we could use $\mathsf{step}(X := T) \multimap \mathsf{eval}(T, X) \otimes \forall V.(\mathsf{retn}(V, X) \multimap \mathsf{mem}(X, V))$.

While this provides a compelling motivation to study quantification in linear logic, for the time being we can work with it intuitively as giving rise to some arbitrary fresh x, y. We also write V plus W for the result of adding the concrete numbers V and W together that has already been defined on numerous occasions in this course (although often in predicate form $\mathsf{plus}(V, W, R)$ with result R).

Adding the case of variables X to the above evaluation is exceedingly simple despite the fact that the value V of variables needs to be looked up from memory via $\mathsf{mem}(X, V)$:

$$\mathsf{mem}(X, V) \otimes \mathsf{eval}(X, u) \multimap \mathsf{mem}(X, V) \otimes \mathsf{retn}(V, u)$$

If we are supposed to evaluate variable X whose present value is V then, while the value of X still is V, we return that V. This case is essentially as simple as the direct variable copy $X := Y$ from the simple imperative programming language.

Why is this remarkable? Because when we would suddenly augment a programming language semantics given in other styles with variables whose values needs to be looked up from the memory, then we need to augment the entire semantic definitions with explicit memory state or environments that needs to be passed through the evaluation function. That is despite the fact that the meaning of addition, say, itself couldn't care less about the role of state, because it is a stateless operation. Thanks to the substructural operational semantics modeled in linear logic can we simply add extraneous state information into the resources tracked and preserved by linear logic whose presence does not bother the linear implications for addition, say. But these extra memory predicates can then be consumed and replaced in the evaluation of variables and its assignments on a need-to-know basis.

Just to illustrate that we have not been overstating the potential of substructural operational semantics and how it benefits from ideas of linear logic, we next consider the case of function application and λ-abstraction. Left-to-right evaluation of function application TS starts an evaluation of T with some intermediate x as a target for the result and leaves around a continuation $\mathsf{cont}(x, \_S, u)$ for evaluating the argument S once T has evaluated to a function:

$$\mathsf{eval}(TS, u) \multimap \exists x.\mathsf{eval}(T, x) \otimes \mathsf{cont}(x, \_S, u)$$

Evaluating a λ-abstraction just leaves the λ-abstraction untouched and returns it since functions are values:

$$\mathsf{eval}(\lambda y.T, u) \multimap \mathsf{retn}(\lambda y.T, u)$$

Finally, when a λ-abstraction is returned into x waiting for a function application continuation $\mathsf{cont}(x, \_S, u)$ then this result in an evaluation of the argument S into some intermediate target z that will be used instead of parameter y in the function body T as indicated by the (extraneously defined) result $T(z/y)$ of the (capture-avoiding) substitution of z for y in T, whose result will be evaluated to be put into u:

$$\mathsf{retn}(\lambda y.T, x) \otimes \mathsf{cont}(x, \_S, u) \multimap \exists z.\mathsf{eval}(S, z) \otimes \mathsf{eval}(T(z/y), u)$$

While more can and should be said about the specification of programming languages with substructural operational semantics, this short glimpse should suffice to give an idea of its power and the role that linear logic (or generalized substructural logics) can play in it. Observe how the overall substructural operational semantics of term evaluation is fairly succinct:

$$\begin{aligned}
\mathsf{eval}(N, u) &\multimap \mathsf{retn}(N, u) \\
\mathsf{eval}(T + S, u) &\multimap \exists x.\mathsf{eval}(T, x) \otimes \mathsf{cont}(x, _ + S, u) \\
\mathsf{retn}(V, x) \otimes \mathsf{cont}(x, _ + S, u) &\multimap \exists y.\mathsf{eval}(S, y) \otimes \mathsf{cont}(y, V + _, u) \\
\mathsf{retn}(W, y) \otimes \mathsf{cont}(y, V + _, u) &\multimap \mathsf{retn}(V \text{ plus } W, u) \\
\mathsf{mem}(X, V) \otimes \mathsf{eval}(X, u) &\multimap \mathsf{mem}(X, V) \otimes \mathsf{retn}(V, u) \\
\mathsf{eval}(TS, u) &\multimap \exists x.\mathsf{eval}(T, x) \otimes \mathsf{cont}(x, _S, u) \\
\mathsf{eval}(\lambda y.T, u) &\multimap \mathsf{retn}(\lambda y.T, u) \\
\mathsf{retn}(\lambda y.T, x) \otimes \mathsf{cont}(x, _S, u) &\multimap \exists z.\mathsf{eval}(S, z) \otimes \mathsf{eval}(T(z/y), u)
\end{aligned}$$

Again, the structural operational semantics of the simple term language would be represented as the conjunction of the exponentials of each of these linear implications.

References

[1] Frank Pfenning and Robert J. Simmons. Substructural operational semantics as ordered logic programming. In *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA*, pages 101–110. IEEE Computer Society, 2009.