15-819K: Logic Programming

Lecture 25

Substructural Operational Semantics

Frank Pfenning

November 30, 2006

In this lecture we combine ideas from the previous two lectures, linear monadic logic programming and higher-order abstract syntax, to present a specification technique for programming languages we call *substructural operational semantics*. The main aim of this style of presentation is semantic modularity: we can add new language features without having to rewrite prior definitions for smaller language fragments. We determine that this is mostly the case, although structural properties of the specification such as weakening or contraction might change.

25.1 A Big-Step Natural Semantics

As a warm-up exercise, and also to understand the lack of modularity in traditional specifications, we present the semantics for functional abstraction and application in a call-by-value language. This is called *natural semantics* because of an analogy to *natural deduction*. The representation of terms employs higher-order abstract syntax, as sketched in the last lecture.

$$\text{Expressions} \quad e \quad ::= \quad x \mid \mathsf{lam}(x.\, e) \mid \mathsf{app}(e_1, e_2)$$

In the expression $\mathsf{lam}(x.\, e)$ the variable x is bound with scope e.

The main judgment is $e \hookrightarrow v$, where e and v are expressions. This is a big-step semantics, so the judgment directly relates e to its final value v.

$$\frac{}{\mathsf{lam}(x.\, e) \hookrightarrow \mathsf{lam}(x.\, e)} \qquad \frac{e_1 \hookrightarrow \mathsf{lam}(x.\, e_1') \quad e_2 \hookrightarrow v_2 \quad e_1'(v_2/x) \hookrightarrow v}{\mathsf{app}(e_1, e_2) \hookrightarrow v}$$

We represent $e \hookrightarrow v$ as $\mathsf{neval}(e, v)$. In the translation into a logic programming notation using higher order abstract syntax we have to be careful about variable binding. It would be incorrect to write the first rule as

$$\mathsf{neval}(\mathsf{lam}(x.\, E), \mathsf{lam}(x.\, E))$$

because E is (implicitly) quantified on the outside, so we could not instantiate with a term that contains x. Instead we must make any dependency explicit with the technique of raising from last lecture.

$$\mathsf{neval}(\mathsf{lam}(x.\, E(x)), \mathsf{lam}(x.\, E(x))).$$

Here, $E(x)$ is a term whose head is a variable. For the second rule we see how substitution is represented as application in the meta-language. E_1' will be bound to an abstraction $x.\, e_1'$, and $E_1'(V_2)$ will carry out the substitution of $e_1'(V_2/x)$.

$$\begin{array}{l}\mathsf{neval}(\mathsf{app}(E_1, E_2), V) \leftarrow \\ \quad \mathsf{neval}(E_1, \mathsf{lam}(x.\, E_1'(x))), \\ \quad \mathsf{neval}(E_1, V_2), \\ \quad \mathsf{neval}(E_1'(V_2), V).\end{array}$$

25.2 Substructural Operational Semantics

In a judgment $e \hookrightarrow v$ the whole state of execution must be present in the components of the judgment. This means, for example, when we add mutable store, we have to rewrite the judgment as $\langle s, e\rangle \hookrightarrow \langle s', v\rangle$, where s is the store before evaluation and s' after. Now all rules (including those for functions which should not be concerned with the store) have to be updated to account for the store. Similar considerations hold for continuations, exceptions, and other enrichments of the language.

Substructural operational semantics has an explicit goal to achieve a more modular presentation, where earlier rules may not have to be revisited. We achieve this through a combination of various ideas. One is the that logical rules that permit contexts are parametric in those contexts. For example, a left rule for conjunction

$$\frac{\Delta, A_1 \Vdash C\ \mathit{true}}{\Delta, A_1 \mathbin{\&} A_2 \Vdash C\ \mathit{true}}$$

remains valid even if new connectives or new judgments are added. The second idea is to evaluate expressions with explicit destinations whose nature remains abstract. Destinations are implemented as parameters.

Substructural operational semantics employs linear and unrestricted assumptions, although I have also considered ordered and affine assumptions. The conclusion on the right-hand is generally only relevant when we tie the semantics into a larger framework, so we omit it in an abstract presentation of the rules.

There are three basic propositions:

$\mathsf{eval}(e, d)$	Evaluate e with destination d
$\mathsf{comp}(f, d)$	Compute frame f with destination d
$\mathsf{value}(d, v)$	Value of destination d is v

Evaluation takes place asynchronously, following the structure of e. Frames are suspended computations waiting for a value to arrive at some destination before they are reawakened. Values of destinations, once computed, are like messages send to suspended frames.

In this first, pure call-by-value language, evaluations, computations, and values are all linear. All the rules are left rules, although we do not specify the right-hand side. A complete evaluation has the form

$$\begin{array}{c} \Delta, \mathsf{value}(d, v) \Vdash \\ \vdots \\ \Delta, \mathsf{eval}(e, d) \Vdash \end{array}$$

where v is the value of e. We begin with expressions $\mathsf{lam}(x.\, E(x))$ which are values and returned immediately to the destination D.

$$\frac{\Delta, \mathsf{value}(D, \mathsf{lam}(x.\, E(x))) \Vdash}{\Delta, \mathsf{eval}(\mathsf{lam}(x.\, E(x)), D) \Vdash}$$

In applications we evaluate the function part, creating a frame that remembers to evaluate the argument, once the function has been computed. For this, we need to create a new destination d_1. The derivation of the premiss must be parametric in d_1. We indicate this by labeling the rule itself with $[d_1]$.

$$\frac{\Delta, \mathsf{comp}(\mathsf{app}_1(d_1, E_2), D), \mathsf{eval}(E_1, d_1) \Vdash}{\Delta, \mathsf{eval}(\mathsf{app}(E_1, E_2), D) \Vdash}\ [d_1]$$

When the expression E_1 has been evaluated, we have to switch to evaluating the argument E_2 with a new destination d_2., keeping in mind that eventually we have to perform the function call.

$$\frac{\Delta, \mathsf{comp}(\mathsf{app}_2(V_1, d_2), D), \mathsf{eval}(E_2, d_2) \Vdash}{\Delta, \mathsf{comp}(\mathsf{app}(D_1, E_2), D), \mathsf{value}(D_1, V_1) \Vdash}\ [d_2]$$

It may be possible to reuse the destination D_1, but we are not interested here in this kind of optimization. It might also interfere negatively with extensibility later on if destinations are reused in this manner.

Finally, the β-reduction when both function and argument are known.

$$\frac{\Delta, \mathsf{eval}(E_1'(V_2), D) \Vdash}{\Delta, \mathsf{comp}(\mathsf{app}_2(\mathsf{lam}(x.\, E_1'(x)), D_2), D), \mathsf{value}(D_2, V_2) \Vdash}$$

25.3 Substructural Operational Semantics in LolliMon

It is easy to take the four rules of our substructural specification and implement them in LolliMon. We need here linear forward chaining and existential quantification to introduce new destinations. LolliMon's term language permits abstraction, so we can use this to implement higher-order abstract syntax.

$$\mathsf{eval}(\mathsf{lam}(x.\, E(x)), D) \multimap \{\mathsf{value}(D, \mathsf{lam}(x.\, E(x)))\}.$$

$$\mathsf{eval}(\mathsf{app}(E_1, E_2), D) \multimap \{\exists d_1.\, \mathsf{eval}(E_1, d_1) \otimes \mathsf{comp}(\mathsf{app}_1(d_1, E_2), D)\}.$$

$$\mathsf{value}(D_1, V_1) \otimes \mathsf{comp}(\mathsf{app}_1(D_1, E_2), D) \multimap \{\exists d_2.\, \mathsf{eval}(E_2, d_2) \otimes \mathsf{comp}(\mathsf{app}_2(V_1, d_2), D)\}.$$

$$\mathsf{value}(D_2, V_2) \otimes \mathsf{comp}(\mathsf{app}_2(\mathsf{lam}(x.\, E_1'(x)), D_2), D) \multimap \{\mathsf{eval}(E_1'(V_2), D)\}.$$

The only change we have made to the earlier specification is to exchange the order of eval, comp, and value propositions for a more natural threading of destinations.

LolliMon combines forward and backward chaining, so we can also write the top-level judgment to obtain the final value.

$$\mathsf{evaluate}(E, V) \circ\!\!- (\forall d_0.\, \mathsf{eval}(E, d_0) \multimap \{\mathsf{value}(d_0, V)\}).$$

25.4 Adding Mutable Store

We would now like to add mutable store to the operational semantics. We have three new kinds of expressions to create, read, and assign to a cell of mutable storage.

$$\text{Expressions} \quad e \quad ::= \quad \ldots \mid \mathsf{ref}(e) \mid \mathsf{deref}(e) \mid \mathsf{assign}(e_1, e_2) \mid \mathsf{cell}(c)$$

There is also a new kind of value $\mathsf{cell}(c)$ where c is a destination which serves as a name for a cell of storage. Note that $\mathsf{cell}(c)$ cannot appear in the

source. In order to enforce this syntactically we would distinguish a type of values from the type of expressions, something we avoid here for the sake of brevity.

First, the rules for creating a new cell. I suggest reading these rules from last to first, in the way they will be used in a computation. We write the destinations that model cells at the left-hand side of the context. This is only a visual aid and has no logical significance.

$$\frac{\mathsf{value}(c_1, V_1), \Delta, \mathsf{value}(D, \mathsf{cell}(c_1)) \Vdash}{\Delta, \mathsf{comp}(\mathsf{ref}_1(D_1), D), \mathsf{value}(D_1, V_1) \Vdash}\ [c_1]$$

$$\frac{\Delta, \mathsf{comp}(\mathsf{ref}_1(d_1), D), \mathsf{eval}(E_1, d_1) \Vdash}{\Delta, \mathsf{eval}(\mathsf{ref}(E_1), D) \Vdash}\ [d_1]$$

We keep track of the value of in a storage cell with an assumption $\mathsf{value}(c, v)$ where c is a destination and v is a value. While destinations to be used as cells are modeled here as linear, they are in reality *affine*, that is, they may be used at most once. The store, which is represented by the set of assumptions $\mathsf{value}(c_i, v_i)$ will remain until the end of the computation.

Next, reading the value of a cell. Again, read the rules from the bottom up, and the last rule first.

$$\frac{\mathsf{value}(C_1, V_1), \Delta, \mathsf{value}(D, V_1) \Vdash}{\mathsf{value}(C_1, V_1), \Delta, \mathsf{comp}(\mathsf{deref}_1(D_1), D), \mathsf{value}(D_1, \mathsf{cell}(C_1)) \Vdash}$$

$$\frac{\Delta, \mathsf{comp}(\mathsf{deref}_1(d_1), D), \mathsf{eval}(E_1, d_1) \Vdash}{\Delta, \mathsf{eval}(\mathsf{deref}(E_1), D) \Vdash}\ [d_1]$$

Next, assigning a value to a cell. The assignment $\mathsf{assign}(e_1, e_2)$ returns the value of e_2.

$$\frac{\mathsf{value}(C_1, V_2), \mathsf{value}(D, V_2) \Vdash}{\mathsf{value}(C_1, V_1), \Delta, \mathsf{comp}(\mathsf{assign}_2(\mathsf{cell}(C_1), D_2), D), \mathsf{value}(D_2, V_2) \Vdash}$$

$$\frac{\Delta, \mathsf{comp}(\mathsf{assign}_2(V_1, d_2), D), \mathsf{eval}(E_2, d_2) \Vdash}{\Delta, \mathsf{comp}(\mathsf{assign}_1(D_1, E_2), D), \mathsf{value}(D_1, V_1) \Vdash}\ [d_2]$$

$$\frac{\Delta, \mathsf{comp}(\mathsf{assign}_1(d_1, E_2), D), \mathsf{eval}(E_1, d_1) \Vdash}{\Delta, \mathsf{eval}(\mathsf{assign}(E_1, E_2), D) \Vdash}\ [d_1]$$

Because values are included in expressions, we need one more rule for cells (which are treated as values). Even if they do not occur in expressions initially, the arise from substitutions of values into expressions.

$$\frac{\Delta, \mathsf{value}(D, \mathsf{cell}(C)) \Vdash}{\Delta, \mathsf{eval}(\mathsf{cell}(C), D) \Vdash}$$

All of these rules are just added to the previous rules for functions. We have achieved semantic modularity, at least for functions and store.

Again, it is easy to turn these rules into a LolliMon program.

$$\begin{array}{l}
\mathsf{eval}(\mathsf{ref}(E_1), D) \\
\quad \multimap \{\exists d_1.\, \mathsf{eval}(E_1, d_1) \otimes \mathsf{comp}(\mathsf{ref}_1(d_1), D)\}. \\
\mathsf{value}(D_1, V_1) \otimes \mathsf{comp}(\mathsf{ref}_1(D_1), D) \\
\quad \multimap \{\exists c_1.\, \mathsf{value}(c_1, V_1) \otimes \mathsf{value}(D, \mathsf{cell}(c_1))\}. \\
\\
\mathsf{eval}(\mathsf{deref}(E_1), D) \\
\quad \multimap \{\exists d_1.\, \mathsf{eval}(E_1, d_1) \otimes \mathsf{comp}(\mathsf{deref}_1(d_1), D)\}. \\
\mathsf{value}(D_1, \mathsf{cell}(C_1)) \otimes \mathsf{value}(C_1, V_1) \otimes \mathsf{comp}(\mathsf{deref}_1(D_1), D) \\
\quad \multimap \{\mathsf{value}(C_1, V_1) \otimes \mathsf{value}(D, V_1)\}. \\
\\
\mathsf{eval}(\mathsf{assign}(E_1, E_2), D) \\
\quad \multimap \{\exists d_1.\, \mathsf{eval}(E_1, d_1) \otimes \mathsf{comp}(\mathsf{assign}_1(d_1, E_2), D)\}. \\
\mathsf{value}(D_1, V_1) \otimes \mathsf{comp}(\mathsf{assign}_1(D_1, E_2), D) \\
\quad \multimap \{\exists d_2.\, \mathsf{eval}(E_2, d_2) \otimes \mathsf{comp}(\mathsf{assign}_2(V_1, d_2), D)\}. \\
\mathsf{value}(D_2, V_2) \otimes \mathsf{comp}(\mathsf{assign}_2(\mathsf{cell}(C_1), D_2), D) \otimes \mathsf{value}(C_1, V_1) \\
\quad \multimap \{\mathsf{value}(C_1, V_2) \otimes \mathsf{value}(D, V_2)\}. \\
\\
\mathsf{eval}(\mathsf{cell}(C), D) \\
\quad \multimap \{\mathsf{value}(D, \mathsf{cell}(C))\}.
\end{array}$$

Written in SSOS form, the program evaluates e with some initial destination d_0 as

$$\begin{array}{c}
\Delta, \mathsf{value}(d_0, v) \Vdash \\
\vdots \\
\mathsf{eval}(e, d_0) \Vdash
\end{array}$$

where v is the value of e and

$$\Delta = \mathsf{value}(c_1, v_1), \ldots, \mathsf{value}(c_n, v_n)$$

for cells $c_1, \ldots, c_n$. The matter is complicated further by the fact that c_i, parameters introduced during the deduction, may appear in v. So we would have to traverse v to eliminate references to c_i, or we could just print it, or we could create some form of closure over the store Δ. In either case, we need to be sure to comsume Δ to retain linearity overall. If we just want to check termination, the top-level program would be

$$\mathsf{terminates}(E) \circ\!\!- \forall d_0.\, \mathsf{eval}(E, d_0) \multimap \{\exists V.\, \mathsf{value}(d_0, V) \otimes \top\}.$$

Here, the existential quantifier will be instantiated after forward chaining reaches quiescence, so it is allowed to depend on all the parameters introduced during forward chaining.

25.5 Adding Continuations

We now add callcc to capture the current continuation and throw to invoke a continuation as an example of an advanced control-flow construct.

$$\text{Expressions} \quad e \quad ::= \quad \ldots \mid \mathsf{callcc}(x.\, e) \mid \mathsf{throw}(e_1, e_2) \mid \mathsf{cont}(d)$$

The intuitive meaning is that $\mathsf{callcc}(x.\, e)$ captures the current continuation (represented as the value $\mathsf{cont}(d)$) and substitutes it for x in e, and that $\mathsf{throw}(e_1, e_2)$ evaluates e_1 to v_1, e_2 to a continuation k and then invokes k on v_1.

In linear destination-passing style, we use a destination d to stand for a continuation. We invoke a continuation d on a value v simply by setting $\mathsf{value}(d, v)$. Any frame waiting to receive a value can be activated in this manner.

But this creates several problems in the semantics. To illustrate them, we add z and $\mathsf{s}(e)$ for zero and successor, and a new frame $\mathsf{s}_1(d)$ which waits to increment the value returned to destination d. See Exercise 25.2 for the rules.

Then an expression

$$\mathsf{s}(\mathsf{callcc}(k.\, \mathsf{s}(\mathsf{throw}(\mathsf{z}, k))))$$

evaluates to $\mathsf{s}(\mathsf{z})$, never returning anything to the inner frame waiting to calculate a successor. This means frames are no longer linear—they may be ignored and therefore be left over at the end.

But the problems do not end there. Consider, for example,

$$\mathsf{app}(\mathsf{callcc}(k.\, \mathsf{lam}(x.\, \mathsf{throw}(\mathsf{lam}(y.\, y), k))), \mathsf{z}).$$

Because any λ-expression is a value, the callcc returns immediately and applies the function $\mathsf{lam}(x.\ \ldots)$ to z. This causes the embedded throw to take place, this time applying $\mathsf{lam}(y.\ y)$ to z, yielding z as the final value. In this computation, the continuation in place when the first argument to app is evaluated is invoked twice: first, because we return to it, and then again when we throw to it. This means frames may not only be ignored, but also duplicated.

The solution is to make all frames $\mathsf{comp}(f, d)$ *unrestricted* throughout. At the same time the other predicates must remain linear: $\mathsf{eval}(e, d)$ so that there is only one thread of computation, and $\mathsf{value}(d, v)$ so that at any given time there is at most one value v at any given destination d.

We present the semantics for callcc and related constructs directly in LolliMon, which is more compact than the inference rule presentation.

$$\begin{array}{l}\mathsf{eval}(\mathsf{callcc}(k.\, E(k)), D) \\ \quad \multimap \{\mathsf{eval}(E(\mathsf{cont}(D)), D)\}.\end{array}$$

$$\begin{array}{l}\mathsf{eval}(\mathsf{throw}(E_1, E_2), D) \\ \quad \multimap \{\exists d_1.\, \mathsf{eval}(E_1, d_1) \otimes !\mathsf{comp}(\mathsf{throw}_1(d_1, E_2), D)\}.\end{array}$$

$$\begin{array}{l}\mathsf{value}(D_1, V_1) \otimes !\mathsf{comp}(\mathsf{throw}_1(D_1, E_2), D) \\ \quad \multimap \{\exists d_2.\, \mathsf{eval}(E_2, d_2) \otimes !\mathsf{comp}(\mathsf{throw}_2(V_1, d_2), D)\}.\end{array}$$

$$\begin{array}{l}\mathsf{value}(D_2, \mathsf{cont}(D_2')) \otimes !\mathsf{comp}(\mathsf{throw}_2(V_1, D_2), D) \\ \quad \multimap \{\mathsf{value}(D_2', V_1)\}.\end{array}$$

$$\begin{array}{l}\mathsf{eval}(\mathsf{cont}(D'), D) \\ \quad \multimap \{\mathsf{value}(D, \mathsf{cont}(D'))\}.\end{array}$$

Of course, all other rules so far must be modified to make the suspended computations (which we now recognize as continuations) unrestricted by prefixing each occurrence of $\mathsf{comp}(f, d)$ with '!'. The semantics is not quite modular in this sense.

With this change we have functions, mutable store, and continuations in the same semantics, specified in the form of a substructural operational semantics.

25.6 Substructural Properties Revisited

The only aspect of our specification we had to revise was the substructural property of suspended computations. If we step back we see that we can use substructural properties of various language features for a kind of taxonomy.

Our first observation is that for functions alone it would have been sufficient to keep the suspended computations next to each other and in order. In such an ordered specification we would not even have needed the destinations, because adjacency guarantees that values arrive at proper locations. We will leave this observation informal, rather than introducing a version of LolliMon with an ordered context, although this would clearly be possible.

If we add mutable store, then propositions $\mathsf{value}(d, v)$ remain linear, while propositions $\mathsf{value}(c, v)$ for destinations d that act as cells are affine. Continuations $\mathsf{comp}(f, d)$ are still linear, as are propositions $\mathsf{eval}(e, d)$.

If we further add a means to capture the continuation, then suspended computations $\mathsf{comp}(f, d)$ must become unrestricted because we may either ignore a continuation or return to it more than once. Values $\mathsf{value}(d, v)$ must remain linear, as must evaluations $\mathsf{eval}(e, d)$. Storage cells remain affine.

With sufficient foresight we could have made suspended computations $\mathsf{comp}(f, d)$ unrestricted to begin with. Nothing in the early semantics relies on their linearity. On other hand, it is more interesting to see what structural properties would and would not work for various languages, and also more natural to assume only the properties that are necessary.

25.7 Historical Notes

The presentation of a big-step operational semantics relating an expression to its value by inference rules is due to Kahn [2] under the name *natural semantics*. Earlier, Plotkin [6] developed a presentation of operational semantics using rewrite rules following the structure of expressions under the name *structural operational semantics* (SOS). I view substructural operational semantics as a further development and extension of SOS. Another generalization to achieve modularity is Mosses' modular structural operational semantics [4] which uses top-down logic programming and a form of row polymorphism for extensibility.

To my knowledge, the first presentation of an operational semantics in linear destination-passing style appeared as an example for the use of the Concurrent Logical Framework (CLF) [1]. The formulation there was intrinsically of higher order, which made reasoning about the rules more difficult. The approach was formulated as an independent technique for modular language specification in an invited talk [5], but only an abstract was published. Further examples of substructural operational semantics were given in the paper that introduced LolliMon [3].

Using ordered assumptions in logical frameworks and logic programming was proposed by Polakow and myself [8, 7], although this work did not anticipate monadic forward chaining as a computational mechanism.

25.8 Exercises

Exercise 25.1 *Prove that the natural semantics and substructural semantics for the functional fragment coincide in a suitable sense.*

Exercise 25.2 *Add natural number constants* z *and* $\mathsf{s}(e)$ *to our language specification to permit more examples for* callcc*. Give formulations both in substructural operational semantics and in LolliMon.*

Exercise 25.3 *Extend the functional language with unit, pairs, sums, void, and recursive types as well as recursion at the level of expressions. Give a substructural operational semantics directly in LolliMon.*

Exercise 25.4 *Think of other interesting control constructs, for example, for parallel or concurrent computation, and represent them in substructural operational semantics.*

Exercise 25.5 *Give a specification of a call-by-value language where we do not substitute a complete value for a term, but only the name for the destination which holds the (immutable) value. Which is the proper substructural property for such destinations?*

Further extend this idea to capture a call-by-need semantics where arguments are evaluated the first time they are needed and then memoized. This is the semantics underlying lazy functional languages such as Haskell.

25.9 References

[1] Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.

[2] Gilles Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer-Verlag LNCS 247, 1987.

[3] Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In A.Felty, editor, *Proceedings of the 7th International Symposium on Principles and Practice of Declarative Programming (PPDP'05)*, pages 35–46, Lisbon, Portugal, July 2005. ACM Press.

[4] Peter D. Mosses. Foundations of modular SOS. In *Proceedings of the 24th International Symposium on Mathematical Foundations of Computer Science (MFCS'99)*, pages 70–80, Szklarska Poreba, Poland, September 1999. Springer-Verlag LNCS 1672. Extended version available as BRICS Research Series RS-99-54, University of Aarhus.

[5] Frank Pfenning. Substructural operational semantics and linear destination-passing style. In Wei-Ngan Chin, editor, *Proceedings of the 2nd Asian Symposium on Programming Languages and Systems (APLAS'04)*, page 196, Taipei, Taiwan, November 2004. Springer-Verlag LNCS 3302.

[6] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[7] Jeff Polakow. *Ordered Linear Logic and Applications*. PhD thesis, Department of Computer Science, Carnegie Mellon University, August 2001.

[8] Jeff Polakow and Frank Pfenning. Relating natural deduction and sequent calculus for intuitionistic non-commutative linear logic. In Andre Scedrov and Achim Jung, editors, *Proceedings of the 15th Conference on Mathematical Foundations of Programming Semantics*, New Orleans, Louisiana, April 1999. Electronic Notes in Theoretical Computer Science, Volume 20.