Substructural Operational Semantics as Ordered Logic Programming

Frank Pfenning Robert J. Simmons*
Carnegie Mellon University
{fp,rjsimmon}@cs.cmu.edu

Abstract

We describe a substructural logic with ordered, linear, and persistent propositions and then endow a fragment with a committed choice forward-chaining operational interpretation. Exploiting higher-order terms in this metalanguage, we specify the operational semantics of a number of object language features, such as call-by-value, call-by-name, call-by-need, mutable store, parallelism, communication, exceptions and continuations. The specifications exhibit a high degree of uniformity and modularity that allows us to analyze the structural properties required for each feature in isolation. Our substructural framework thereby provides a new methodology for language specification that synthesizes structural operational semantics, abstract machines, and logical approaches.

1 Introduction

Substructural logics, of which the most well known is linear logic, have proven a powerful framework for describing systems that incorporate a notion of state and state transition. Linear logic endowed with a *forward-chaining* (or "*bottom-up*"), *committed choice* operational semantics can be used to specify stateful computations from operational semantics [2, 15] to greedy algorithms [24]; in fact, both multiset rewriting and process calculi can be related to linear logic in this form [3].

In this paper, we describe a forward-chaining, committed choice operational semantics for a fragment of *ordered logic* [21], an extension of intuitionistic linear logic with ordered hypotheses in the style of the Lambek calculus [8]. Our motivation for doing this is that it allows us to use ordered logic to specify the *substructural operational semantics* of programming languages. This style of specification, previously considered only for linear logic, is, in ordered logic, a particularly elegant, compact, and modular representation of the semantics of programming languages. Our primary contributions are a sound and nondeterministically complete semantics for a forward-reasoning fragment of ordered logic and a new foundation for substructural operational semantics in ordered logic.

We briefly review ordered logic in Section 2. In Section 3 we present a fragment of this logic and describe our sound and non-deterministically complete operational semantics on this fragment. In Section 4 we develop substructural operational semantics in ordered logic and examine the specification of a variety of language features in terms of their substructural properties. We mention some additional related work in Section 5 and conclude with a summary and remarks on future work in Section 6.

While Sections 2 and 3 provide the formal background for Section 4, the development of substructural operational semantics can be informally understood independently of the introduction to ordered logic.

2 Ordered logic

We briefly review a sequent formulation of ordered logic [22]. The principal judgment has the form $\Gamma; \Delta; \Omega \vdash C$, where Γ is a context satisfying exchange, weakening, and contraction, Δ is a context satisfying only exchange, and Ω is a context satisfying none of the three structural principles. The rules for the fragment relevant to this paper are given in Fig. 1. We use A, B, C to stand for arbitrary propositions and p for atomic propositions, and write $\Delta_1 \bowtie \Delta_2$ for the nondeterministic merge of two linear contexts.

In ordered logic, the linear implication $A \multimap B$ splits into a left and a right implication, written as $A \rightarrowtail B$ and $A \twoheadrightarrow B$, respectively. Their right rules add the hypothesis A to the left and right of the ordered context, respectively. Because of the limited use we make of implication, we only use right implication, although we could have just as well used left implication.

*This material is based upon work supported under a National Science Foundation Graduate Research Fellowship.

Structural Rules

$$\frac{}{\Gamma;\cdot;p\vdash p}\ \textbf{init} \qquad \frac{\Gamma_L A\Gamma_R;\Delta;\Omega_L A\Omega_R\vdash B}{\Gamma_L A\Gamma_R;\Delta;\Omega_L\Omega_R\vdash B}\ \textbf{copy} \qquad \frac{\Gamma;\Delta_L\Delta_R;\Omega_L A\Omega_R\vdash B}{\Gamma;\Delta_L A\Delta_R;\Omega_L\Omega_R\vdash B}\ \textbf{place}$$

Ordered Implication

$$\frac{\Gamma;\Delta;\Omega A\vdash B}{\Gamma;\Delta;\Omega\vdash A\twoheadrightarrow B}\ {\twoheadrightarrow}_R \qquad \frac{\Gamma;\Delta_A;\Omega_A\vdash A \qquad \Gamma;\Delta;\Omega_L B\Omega_R\vdash C}{\Gamma;\Delta\bowtie\Delta_A;\Omega_L(A\twoheadrightarrow B)\Omega_A\Omega_R\vdash C}\ {\twoheadrightarrow}_L$$

Conjunction

$$\frac{\Gamma;\Delta_A;\Omega_L\vdash A \qquad \Gamma;\Delta_B;\Omega_R\vdash B}{\Gamma;\Delta_A\bowtie\Delta_B;\Omega_L\Omega_R\vdash A\bullet B}\ \bullet_R \qquad \frac{\Gamma;\Delta;\Omega_L AB\Omega_R\vdash C}{\Gamma;\Delta;\Omega_L(A\bullet B)\Omega_R\vdash C}\ \bullet_L \qquad \frac{}{\Gamma;\cdot;\cdot\vdash\mathbf{1}}\ \mathbf{1}_R \qquad \frac{\Gamma;\Delta;\Omega_L\Omega_R\vdash C}{\Gamma;\Delta;\Omega_L(\mathbf{1})\Omega_R\vdash C}\ \mathbf{1}_L$$

Modalities

$$\frac{\Gamma;\cdot;\cdot\vdash A}{\Gamma;\cdot;\cdot\vdash !A}\ !_R \qquad \frac{\Gamma A;\Delta;\Omega_L\Omega_R\vdash C}{\Gamma;\Delta;\Omega_L(!A)\Omega_R\vdash C}\ !_L \qquad \frac{\Gamma;\Delta;\cdot\vdash A}{\Gamma;\Delta;\cdot\vdash ¡A}\ ¡_R \qquad \frac{\Gamma;\Delta A;\Omega_L\Omega_R\vdash C}{\Gamma;\Delta;\Omega_L(¡A)\Omega_R\vdash C}\ ¡_L$$

Quantifiers

$$\frac{\Gamma;\Delta;\Omega\vdash A[a/x]}{\Gamma;\Delta;\Omega\vdash\forall x.A}\ \forall^a_R \qquad \frac{\Gamma;\Delta;\Omega_L(A[t/x])\Omega_R\vdash C}{\Gamma;\Delta;\Omega_L(\forall x.A)\Omega_R\vdash C}\ \forall_L \qquad \frac{\Gamma;\Delta;\Omega\vdash A[t/x]}{\Gamma;\Delta;\Omega\vdash\exists x.A}\ \exists_R \qquad \frac{\Gamma;\Delta;\Omega_L(A[a/x])\Omega_R\vdash C}{\Gamma;\Delta;\Omega_L(\exists x.A)\Omega_R\vdash C}\ \exists^a_L$$

Figure 1. Cut-free sequent calculus for a fragment of ordered logic

In addition to the exponential modality $!A$ of linear logic we have a modality of *mobility* $¡A$ that promotes A from the ordered to the linear context.

For the purposes of this paper, variables are typed and terms (which are embedded in atomic propositions) are drawn from the simply-typed λ-calculus. We generally omit types and type declarations in order to concentrate on logical structure. We use a to stand for parameters, which are freshly introduced in the $\forall_R$ and $\exists_L$ rules.

In this sequent calculus, both cut and identity (that is $\cdot;\cdot;A\vdash A$ for any A) are admissible [22].

3 Ordered linear logic programming

A fragment of ordered logic can be endowed with a backward-chaining (or "top-down") operational semantics (in the style of Prolog) that is sound and nondeterministically complete with respect to the logic [19, 20]. This so-called *uniform fragment* [13] is effectively freely generated by all connectives with invertible right rules, and conservatively extends prior work on linear logic programming by Hodas and Miller [7]. The complexity of resource management in implementations of ordered logic programming is considerable, which has so far limited the uniform fragment's range of applications. Examples so far have been drawn mainly from computational linguistics, programming languages, and imperative algorithms [23, 20, 5].

A different approach to logic programming in general is to give a forward-chaining operational semantics. While Horn clauses are simple enough to support both kinds of semantics, for richer languages appropriate fragments have been difficult to isolate. A key insight, due to Andreoli [1], is that we may freely consider atoms either as *negative* or *positive*, corresponding to propositions invertible on the right or left of a sequent, respectively. A minimalistic language including positive connectives (those with invertible left rules) and atoms admits an interesting and useful forward-chaining operational semantics for linear logic [24]. In this section, we extend this to ordered logic and prove it sound and nondeterministically complete.

3.1 Weak focusing

Focusing reduces nondeterminism from proof search in three ways: eagerly applying invertible rules, chaining non-invertible rules, and restricting the use of atoms in initial sequents. For technical reasons it is convenient to present a system of *weak focusing* which forces the chaining of non-invertible rules and restricts atoms, but does not require invertible rules to be applied eagerly. Laurent describes the analogous property in classical linear logic as *+-focalization* [9].

We consider here only the positive fragment of ordered logic which allows us to define rules with a premise S and a conclusion S' made up exclusively from positive propositions. These rules constitute the logic program and are reflected as logical propositions with a shallow layer of negative propositions of the form $\forall x_1\ldots\forall x_n.S\twoheadrightarrow S'$. These considerations yield the fol-

lowing syntax:

$$\begin{array}{llll} \textit{Neg. Props.} & A & ::= & \forall x.A \mid S_1 \twoheadrightarrow S_2 \\ \textit{Pos. Props.} & S & ::= & p \mid ¡p \mid !p \mid S_1 \bullet S_2 \mid \mathbf{1} \mid \exists x.S \end{array}$$

An atomic proposition p is said to be *ordered*, an atomic proposition $¡p$ is *linear* (and therefore mobile), and an atomic proposition $!p$ is persistent. As in previous work [24], we restrict the use of the ¡ and ! modalities to atomic propositions, and further require *separation*: every predicate must be used consistently as either ordered, linear, or persistent, that is, always occur as p, $¡p$ or $!p$.

The weak focusing system in Fig. 2 uses three forms of sequents

Unfocused sequent	$\Gamma; \Delta; \Omega \Rightarrow S'$
Left focused sequent	$\Gamma; \Delta; \Omega_L[A]\Omega_R \Rightarrow S'$
Right focused sequent	$\Gamma; \Delta; \Omega \Rightarrow [S]$

where contexts are restricted to

$$\begin{array}{llll} \textit{Persistent Hyps.} & \Gamma & ::= & \cdot \mid \Gamma A \mid \Gamma p \\ \textit{Linear Hyps.} & \Delta & ::= & \cdot \mid \Delta p \\ \textit{Ordered Pos. Hyps.} & \Omega & ::= & \cdot \mid \Omega S \end{array}$$

and atomic propositions p in Γ, Δ, and Ω must be persistent, linear, and ordered, respectively. Note that the rules in Fig. 2 prevent invertible rules from being applied during focus; the premise and conclusion of $\mathbf{1}_L$, for instance, are both specifically unfocused sequents.

The **init**¡ and **init**! rules combine aspects of the modality with the initial sequent. This is a slight extension beyond Andreoli's focusing system, even on just the linear fragment. In most focusing systems, it is critical that the modalities be allowed to interrupt the focusing phase. This is problematic from the point of view of our desired operational semantics, because we will want to require the *immediate* presence of an atomic proposition within the ordered, linear, or persistent context as a precondition to applying a rule, which is precisely what is captured by the **init**¡ and **init**! rules. However, because those rules do not allow the modalities to break focus, they would be incomplete if it were not for the assumption of *separation* mentioned above.

The $\twoheadrightarrow_L$ rule retains focus in the first premise as expected, but not in the second because of the implicit shift operator (in the polarized form $S_1 \twoheadrightarrow \uparrow S_2$, or in monadic form $S_1 \twoheadrightarrow \{S_2\}$ as in CLF [26]).

The soundness of the focusing system is trivial: if we ignore all the brackets, each rule is derivable in the ordered sequent calculus. The remainder of this subsection is devoted to the completeness theorem.

When we look at an ordinary sequent proof of $\Gamma; \Delta; \Omega \vdash S$ where Γ, Δ, and Ω are drawn from our fragment, we notice that there are two difficulties. One is that during the proof, the ordered context may contain not only positive propositions S but also arbitrarily many negative propositions A copied from Γ. The second difficulty is the usual one: rules must be applied to the proposition in focus to the exclusion of others.

We need some straightforward structural properties, where applicable, and the following admissibility lemma.

Lemma 1 (Unfocused admissibility of $\bullet_R$, $\exists_R$, $\twoheadrightarrow_L$, and $\forall_L$)**.**

1. *If* $\Gamma; \Delta_1; \Omega_1 \Rightarrow S_1$ *and* $\Gamma; \Delta_2; \Omega_2 \Rightarrow S_2$ *then* $\Gamma; \Delta_1 \bowtie \Delta_2; \Omega_1\Omega_2 \Rightarrow S_1 \bullet S_2$.
2. *If* $\Gamma; \Delta_1; \Omega_1 \Rightarrow S_1$ *and* $\Gamma; \Delta_2; \Omega_L S_2 \Omega_R \Rightarrow S'$ *for* $S_1 \twoheadrightarrow S_2 \in \Gamma$, *then* $\Gamma; \Delta_1 \bowtie \Delta_2; \Omega_L\Omega_1\Omega_R \Rightarrow S'$.
3. *If* $\Gamma; \Delta; \Omega \Rightarrow S[t/x]$ *then* $\Gamma; \Delta; \Omega \Rightarrow \exists x.\, S$
4. *If* $\Gamma; \Delta; \Omega_L(A[t/x])\Omega_R \Rightarrow S'$ *then* $\Gamma; \Delta; \Omega_L(\forall x.A)\Omega_R \Rightarrow S'$.

Proof. We generalize (1) to allow either premise to be a left-focused sequent and then proceed by mutual simultaneous induction on the structure of the two given derivations.

We generalize (2) to allow the first premise to be a left-focused sequent and then proceed by mutual induction on the structure of the first given derivation.

Parts (3) and (4) follow similarly. □

The property of separation influences how we relate an unfocused proof to a weakly focused proof in the completeness theorem. If a positive proposition in an unfocused sequent (written S^*) is a persistent or linear atom p, then the focused sequent calculus must add the prefix "!" or "¡" (this appropriately-prefixed version of S^* is written S^+). The same goes for an unfocused ordered context $\Omega^\pm$, which must be separated into a context of appropriately-prefixed positive propositions Ω^+ and a context of negative propositions Ω^-.

Theorem 2 (Completeness of focusing for the positive fragment)**.** *If* $\Gamma; \Delta; \Omega^\pm \vdash S^*$ *then* $\Gamma\Omega^-; \Delta; \Omega^+ \Rightarrow S^+$

Proof. By induction on the structure of the given derivation. We need various simple structural properties, plus the admissibility of unfocused $\bullet_R$, $\exists_R$, $\twoheadrightarrow_L$ and $\forall_L$ rules. In the case of a **copy** or **place** rule we exploit the invertiblity of $!_L$ and $¡_L$, respectively (Lemma 3). An appeal to invertibility could be avoided with a slightly more complicated translation that sorts persistent and linear atoms from $\Omega^\pm$ into Γ and Δ, respectively. □

Initial Rules

$$\frac{}{\Gamma;\cdot;p \Rightarrow [p]}\ \mathbf{init} \qquad \frac{}{\Gamma;p;\cdot \Rightarrow [¡p]}\ \mathbf{init}_¡ \qquad \frac{}{\Gamma_L p \Gamma_R;\cdot;\cdot \Rightarrow [!p]}\ \mathbf{init}_!$$

Focusing Rules

$$\frac{\Gamma_L A \Gamma_R;\Delta;\Omega_L[A]\Omega_R \Rightarrow S'}{\Gamma_L A \Gamma_R;\Delta;\Omega_L\Omega_R \Rightarrow S'}\ \mathbf{focus}_L \qquad \frac{\Gamma;\Delta;\Omega \Rightarrow [S']}{\Gamma;\Delta;\Omega \Rightarrow S'}\ \mathbf{focus}_R$$

Ordered Implication

$$\frac{\Gamma;\Delta_1;\Omega_1 \Rightarrow [S_1] \qquad \Gamma;\Delta_2;\Omega_L S_2 \Omega_R \Rightarrow S'}{\Gamma;\Delta_1 \bowtie \Delta_2;\Omega_L[S_1 \twoheadrightarrow S_2]\Omega_1\Omega_R \Rightarrow S'}\ \twoheadrightarrow_L$$

Conjunction

$$\frac{\Gamma;\Delta_1;\Omega_L \Rightarrow [S_1] \quad \Gamma;\Delta_2;\Omega_R \Rightarrow [S_2]}{\Gamma;\Delta_1 \bowtie \Delta_2;\Omega_L\Omega_R \Rightarrow [S_1 \bullet S_2]}\ \bullet_R \qquad \frac{\Gamma;\Delta;\Omega_L S_1 S_2 \Omega_R \Rightarrow S'}{\Gamma;\Delta;\Omega_L(S_1 \bullet S_2)\Omega_R \Rightarrow S'}\ \bullet_L \qquad \frac{}{\Gamma;\cdot;\cdot \Rightarrow [\mathbf{1}]}\ \mathbf{1}_R \qquad \frac{\Gamma;\Delta;\Omega_L\Omega_R \Rightarrow S'}{\Gamma;\Delta;\Omega_L(\mathbf{1})\Omega_R \Rightarrow S'}\ \mathbf{1}_L$$

Modalities

$$\frac{\Gamma p;\Delta;\Omega_L\Omega_R \Rightarrow S'}{\Gamma;\Delta;\Omega_L(!p)\Omega_R \Rightarrow S'}\ !_L \qquad \frac{\Gamma;\Delta p;\Omega_L\Omega_R \Rightarrow S'}{\Gamma;\Delta;\Omega_L(¡p)\Omega_R \Rightarrow S'}\ ¡_L$$

Quantifiers

$$\frac{\Gamma;\Delta;\Omega_L[A[t/x]]\Omega_R \Rightarrow_\Sigma S'}{\Gamma;\Delta;\Omega_L[\forall x.A]\Omega_R \Rightarrow_\Sigma S'}\ \forall_L \qquad \frac{\Gamma;\Delta;\Omega \Rightarrow_\Sigma [S[t/x]]}{\Gamma;\Delta;\Omega \Rightarrow_\Sigma [\exists x.S]}\ \exists_R \qquad \frac{\Gamma;\Delta;\Omega_L(S[a/x])\Omega_R \Rightarrow_{\Sigma,a} S'}{\Gamma;\Delta;\Omega_L(\exists x.S)\Omega_R \Rightarrow_\Sigma S'}\ \exists^a_L$$

Figure 2. Weakly focused sequent calculus; focus propositions shown in [brackets]

3.2 Logic programming semantics

For the operational semantics we need to resolve the choice of terms t in the $\forall_L$ and $\exists_R$ rules. It is sufficient for our purposes to utilize a decidable and unitary form of higher-order matching [16]. We must require all program rules $\forall x_1 \ldots \forall x_n.S \twoheadrightarrow S'$ to be *range restricted*: all variables occurring in S' also have at least one *strict occurrence* [16] in S. Moreover, each universally or existentially quantified variable should have a strict occurrence in its scope. Together, these requirements guarantee that if we start with a ground database of atomic propositions, any matching problem that arises during forward chaining is decidable, and any new atomic proposition generated will be ground.

We now rewrite the weakly focused rules in the form of a state transition system capturing the intended proof search semantics. A *stable sequent* is an unfocused sequent $\Gamma;\Delta;\Omega \Rightarrow S'$ where $\Omega = \Omega^p$ consists entirely of atomic propositions. Whenever we reach an unfocused sequent $\Gamma;\Delta;\Omega \Rightarrow S'$ (where Ω may contain arbitrary positive propositions) we decompose the positive propositions eagerly to reach a stable sequent. This is possible without losing completeness because all left rules for positive propositions are invertible.

Lemma 3 (Invertibility)**.** *The rules $\bullet_L$, $\mathbf{1}_L$, $!_L$, $¡_L$, and $\exists_L$ are all invertible.*

Proof. In each case we generalize the induction hypothesis to include focused sequents and then prove the theorem by mutual induction on the given derivation. □

It is also easy to see that the order in which the invertible rules are applied does not matter—the resulting stable sequent is always the same, possibly modulo renaming of the new parameters. Because scope is no longer obvious, we index a sequent by its parameters Σ. We write

$$(\Gamma;\Delta;\Omega \Rightarrow_\Sigma S') \xrightarrow{+} (\Gamma';\Delta';\Omega^p \Rightarrow_{\Sigma'} S')$$

if there is a proof

$$\begin{array}{c} \Gamma';\Delta';\Omega^p \Rightarrow_{\Sigma'} S' \\ \vdots \\ \Gamma;\Delta;\Omega^+ \Rightarrow_\Sigma S' \end{array}$$

using only invertible rules $\bullet_L$, $\mathbf{1}_L$, $!_L$, $¡_L$ and $\exists_L$, and Σ' extends Σ with new parameters introduced by applications of the $\exists_L$ rule.

Next we define another kind of transition derived from the focusing phase. We write

$$(\Gamma;\Delta;\Omega^p \Rightarrow_\Sigma S') \xrightarrow{-} (\Gamma;\Delta';\Omega \Rightarrow_\Sigma S')$$

for $\Omega^p = \Omega^p_L \Omega^p_R$ if there is a proof

$$\frac{\begin{array}{c}\Gamma; \Delta'; \Omega \Rightarrow S' \\ \vdots \\ \Gamma; \Delta; \Omega^p_L [A] \Omega^p_R \Rightarrow S'\end{array}}{\Gamma; \Delta; \Omega^p_L \Omega^p_R \Rightarrow S'}\ \mathbf{focus}_L$$

using only the rules acting on left or right focused sequents (**init**, **init**↓, **init**!, $\twoheadrightarrow_L$, $\bullet_R$, $\mathbf{1}_R$, $\forall_L$, $\exists_R$). Note that in all these rules, Γ and S' remain the same (in the only or the rightmost premise), and that no new parameters are introduced.

Finally, we can complete a proof only by successfully focusing on the right. We write

$$(\Gamma; \Delta; \Omega^p \Rightarrow_\Sigma S') \rightarrow|$$

if there is a (complete) proof

$$\frac{\begin{array}{c}\vdots \\ \Gamma; \Delta; \Omega^p \Rightarrow [S']\end{array}}{\Gamma; \Delta; \Omega^p \Rightarrow S'}\ \mathbf{focus}_R$$

We note that this proof stays entirely within right-focused sequents. In fact, it is decidable if $(\Gamma; \Delta; \Omega^p \Rightarrow_\Sigma S') \rightarrow|$ holds because the required higher-order matching is decidable.

The resulting system of transitions is sound and complete in the sense of the following theorem. Together with the soundness and completeness of the focusing system, this yields nondeterministic soundness and completeness with respect to ordered logic.

Theorem 4 (Nondeterministic completeness)**.** *If the contexts Δ_0 and Ω_0 contain only ordered propositions, then $\Gamma_0; \Delta_0; \Omega_0 \Rightarrow_{\Sigma_0} S'$ iff there exists a sequence of alternating transitions*

$$\begin{array}{l}(\Gamma_0; \Delta_0; \Omega_0 \Rightarrow_{\Sigma_0} S') \xrightarrow{-} (\Gamma_0; \Delta'_0; \Omega'_0 \Rightarrow_{\Sigma_0} S') \xrightarrow{+} \\ (\Gamma_1; \Delta_1; \Omega_1 \Rightarrow_{\Sigma_1} S') \xrightarrow{-} (\Gamma_1; \Delta'_1; \Omega'_1 \Rightarrow_{\Sigma_1} S') \xrightarrow{+} \\ \ldots \\ (\Gamma_n; \Delta_n; \Omega_n \Rightarrow_{\Sigma_n} S') \rightarrow|\end{array}$$

Proof. Soundness is shown by reading the transitions off a version of the focused proof where all invertible rules are applied eagerly, according to the invertibility lemma.

Completeness follows by reconstructing a proof according to the definitions of the transitions of the operational semantics. □

We want to use the transitions to model computations of object languages. In this setting, successful termination is usually not crucial — we are interested instead in the evolution of the state given by the contexts Γ, Δ, and Ω. From a given initial state $(\Gamma_0; \Delta_0; \Omega_0)$ and fixed program in Γ_0, we proceed with alternating transitions as above, ignoring the right-hand side (which would remain unchanged anyway). We commit to any transition rather than potentially backtracking in order to capture the possible behavior of an object language program rather than exploring all of its behaviors (which is an interesting but separate problem). We terminate if no further transitions are possible. Other refinements, such as terminating if all further transitions remain in the same state, that is, saturation, can also be considered, but are not important for the purpose of this paper.

4 Substructural operational semantics

Substructural operational semantics (SSOS) is a style of defining the operational semantics of programming languages using substructural logics. It has its origins in some examples illustrating the Concurrent Logical Framework (CLF) [2] and was proposed as a specification methodology in an invited talk [15]. Previous work has considered only frameworks with linear and affine propositions, and has applied aspects of substructural operational semantics to functional programming [10], to derive program approximations for functional and imperative languages [25], and to reason about concurrent computations [27].

In this paper we illustrate the remarkable expressiveness of SSOS when we enrich it further by considering ordered logic as the underlying framework. By giving the framework the forward-chaining, committed choice semantics presented in Section 3, the semantic specifications are not only elegant, compact, and modular, but also executable in a natural way.

The SSOS methodology distinguishes three categories of atomic propositions. *Active* propositions are always ephemeral (ordered or linear), and may be consumed eagerly and replaced with something else. *Latent* propositions represent suspended computations that await some information or event in order to start an active computation. Finally, *passive* propositions either hold values or represent events; they do not drive computation, but they may activate latent propositions.

4.1 Basic language features

Call-by-value. We start with a call-by-value operational semantics for the (untyped) λ-calculus. Expressions are represented using higher-order abstract syntax of the form x, $\mathsf{lam}\ (\lambda x.\, e\, x)$, and $\mathsf{app}\ e_1\ e_2$. Meta-level application is used to represent substitution, as

$$\mathbf{eval}(\mathsf{lam}\ (\lambda x.\, E\, x)) \twoheadrightarrow \mathbf{return}(\mathsf{lam}\ (\lambda x.\, E\, x))$$
$$\mathbf{eval}(\mathsf{app}\ E_1\ E_2) \twoheadrightarrow \mathbf{comp}(\mathsf{app}_1\ E_2) \bullet \mathbf{eval}(E_1)$$
$$\mathbf{comp}(\mathsf{app}_1\ E_2) \bullet \mathbf{return}(V_1) \twoheadrightarrow \mathbf{comp}(\mathsf{app}_2\ V_1) \bullet \mathbf{eval}(E_2)$$
$$\mathbf{comp}(\mathsf{app}_2\ (\mathsf{lam}\ (\lambda x.\, E_1'\, x))) \bullet \mathbf{return}(V_2) \twoheadrightarrow \mathbf{eval}(E_1'\ V_2)$$

Figure 3. Call-by-value functions

usual with this representation technique.

We treat the ordered context as a stack of latent propositions $\mathbf{comp}(f)$, where f is a *frame*, followed by either an active proposition $\mathbf{eval}(e)$, representing the goal to evaluate e, or a passive proposition $\mathbf{return}(r)$, representing a value being returned.

Frames are either $\mathsf{app}_1\ e_2$, which contains the argument part of an application and waits for the function part to be evaluated, or else $\mathsf{app}_2\ v_1$, which contains the evaluated function part of an application and waits for the argument part to be evaluated.

The rules are shown in Fig. 3. The free variables in each rule are implicitly universally quantified over the whole rule. Note that order is crucial here, because a frame receives a value from the right, performs some computation, and then passes the resulting value to the frame on its left.

Using informal pencil-and-paper representation for object-language terms ($\lambda x.x$ instead of $\mathsf{lam}(\lambda x.x)$) and abbreviating **comp**, **eval**, and **return** as **c**, **e**, and **r**, we show the evaluation of $(\lambda x.x)((\lambda y.y)(\lambda z.e))$ below. Since the persistent context contains only the logic program Γ, and the linear context remains empty, we show only the evolution of the ordered context, where $\Omega_1 \xrightarrow{-+} \Omega_2$ is short for $(\Gamma; \cdot; \Omega_1 \Rightarrow_{\cdot} S') \xrightarrow{-} (\Gamma; \cdot; \Omega_1' \Rightarrow_{\cdot} S') \xrightarrow{+} (\Gamma; \cdot; \Omega_2 \Rightarrow_{\cdot} S')$ for an arbitrary S'.

$$\begin{array}{rrrl}
 & & \mathbf{e}((\lambda x.x)((\lambda y.y)(\lambda z.e))) & \xrightarrow{-+} \\
 & \mathbf{c}(\mathsf{app}_1\ ((\lambda y.y)(\lambda z.e))) & \mathbf{e}(\lambda x.x) & \xrightarrow{-+} \\
 & \mathbf{c}(\mathsf{app}_1\ ((\lambda y.y)(\lambda z.e))) & \mathbf{r}(\lambda x.x) & \xrightarrow{-+} \\
 & \mathbf{c}(\mathsf{app}_2\ (\lambda x.x)) & \mathbf{e}((\lambda y.y)(\lambda z.e)) & \xrightarrow{-+} \\
\mathbf{c}(\mathsf{app}_2\ (\lambda x.x)) & \mathbf{c}(\mathsf{app}_1\ (\lambda z.e)) & \mathbf{e}(\lambda y.y) & \xrightarrow{-+} \\
 & & \cdots & \\
 & \mathbf{c}(\mathsf{app}_2\ (\lambda x.x)) & \mathbf{r}(\lambda z.e) & \xrightarrow{-+} \\
 & & \mathbf{e}(\lambda z.e) & \xrightarrow{-+} \\
 & & \mathbf{r}(\lambda z.e) &
\end{array}$$

Mutable storage. We now extend the call-by-value λ-calculus with mutable storage in a completely modular way, that is, without changing the rules for functions and applications. We have three new source language expressions, $\mathsf{ref}\ e$ to allocate a new cell initialized with the value of e, $\mathsf{get}\ e$ to read the value of the cell denoted by e, and $\mathsf{set}\ e_1\ e_2$ to assign the value of e_2 to the location denoted by e_1. Frames ref_1, get_1, $\mathsf{set}_1\ e_2$ and $\mathsf{set}_2\ v_1$ that correspond to partially evaluated terms are also added.

$$\mathbf{eval}(\mathsf{ref}\ E) \twoheadrightarrow \mathbf{comp}(\mathsf{ref}_1) \bullet \mathbf{eval}(E)$$
$$\mathbf{comp}(\mathsf{ref}_1) \bullet \mathbf{return}(V) \twoheadrightarrow \exists d.\, \mathbf{return}(\mathsf{loc}\ d) \bullet \text{¡}\mathbf{cell}\ d\ V$$

$$\mathbf{eval}(\mathsf{get}\ E) \twoheadrightarrow \mathbf{comp}(\mathsf{get}_1) \bullet \mathbf{eval}(E)$$
$$\mathbf{comp}(\mathsf{get}_1) \bullet \mathbf{return}(\mathsf{loc}\ D) \bullet \text{¡}\mathbf{cell}\ D\ V \twoheadrightarrow \mathbf{return}(V) \bullet \text{¡}\mathbf{cell}\ D\ V$$

$$\mathbf{eval}(\mathsf{set}\ E_1\ E_2) \twoheadrightarrow \mathbf{comp}(\mathsf{set}_1\ E_2) \bullet \mathbf{eval}(E_1)$$
$$\mathbf{comp}(\mathsf{set}_1\ E_2) \bullet \mathbf{return}(V_1) \twoheadrightarrow \mathbf{comp}(\mathsf{set}_2\ V_1) \bullet \mathbf{eval}(E_2)$$
$$\mathbf{comp}(\mathsf{set}_2\ (\mathsf{loc}\ D_1)) \bullet \mathbf{return}(V_2) \bullet \text{¡}\mathbf{cell}\ D_1\ V' \twoheadrightarrow \mathbf{return}(V_2) \bullet \text{¡}\mathbf{cell}\ D_1\ V_2$$

$$\mathbf{eval}(\mathsf{loc}\ D) \twoheadrightarrow \mathbf{return}(\mathsf{loc}\ D)$$

Figure 4. Mutable storage

In addition to the stack of frames in the ordered context followed by **eval** or **return**, we now also have a *linear* context of the form

$$\mathbf{cell}\ d_1\ v_1, \ldots, \mathbf{cell}\ d_n\ v_n$$

where $d_1, \ldots, d_n$ are pairwise distinct parameters representing abstract locations holding values $v_1, \ldots, v_n$, respectively. The **cell** propositions cannot be ordered because locations may be accessed from any frame. They cannot be persistent because assignment must be able to change the value associated with a location; this is not possible for persistent propositions, but is achieved with linear propositions by consuming the previous value.

We generate a new abstract location (see the rule for $\mathbf{comp}(\mathsf{ref}_1)$) by using existential quantification in the head of a clause, that is, on the right-hand side of an ordered implication. This will create a fresh parameter according to the operational semantics of the metalanguage. Reading a location (see the rule for $\mathbf{comp}(\mathsf{get}_1)$) consumes and then restores its content in one atomic step. Writing to a location (see the rule for $\mathbf{comp}(\mathsf{set}_2)$) consumes it and then restores it with its new value.

The abstract locations created by existential quantification are the first use we have made of *destinations*, a distinguishing feature of substructural operational semantics. Destinations are used to connect non-local information (in the form of passive or latent propositions) to the active part of the computation (represented by active propositions).

$$\mathbf{eval}(\mathsf{pair}\ E_1\ E_2) \\ \quad \twoheadrightarrow \mathbf{comp}(\mathsf{pair}_1) \bullet \mathbf{eval}(E_1) \bullet \mathbf{eval}(E_2)$$
$$\mathbf{comp}(\mathsf{pair}_1) \bullet \mathbf{return}(V_1) \bullet \mathbf{return}(V_2) \\ \quad \twoheadrightarrow \mathbf{return}(\mathsf{pair}\ V_1\ V_2)$$

$$\mathbf{eval}(\mathsf{split}\ E\ (\lambda x_1.\lambda x_2.\, E'\, x_1\, x_2)) \\ \quad \twoheadrightarrow \mathbf{comp}(\mathsf{split}_1\ (\lambda x_1.\lambda x_2.\, E'\, x_1\, x_2)) \bullet \mathbf{eval}(E)$$
$$\mathbf{comp}(\mathsf{split}_1\ (\lambda x_1.\lambda x_2.\, E'\, x_1\, x_2)) \bullet \mathbf{return}(\mathsf{pair}\ V_1\ V_2) \\ \quad \twoheadrightarrow \mathbf{eval}(E'\ V_1\ V_2)$$

Figure 5. Parallel evaluation for pairs

Parallel evaluation. As another modular extension, we add pairs whose components are evaluated in parallel and where both components must be evaluated to values before a pair of values is formed. The latent $\mathbf{comp}(\mathsf{pair}_1)$ is waiting here for two values before proceeding to form the pair and return it. We decompose such a pair using a construct $\mathsf{split}\ e\ \mathsf{as}\ (x_1, x_2)\ \mathsf{in}\ e'$, which is represented by $\mathsf{split}\ e\ (\lambda x_1.\lambda x_2.\, e'\, x_1\, x_2)$ in higher-order abstract syntax.

This example illustrates that multiple active propositions (**eval**) may be in the ordered context at a time. The committed choice semantics of the metalanguage will step these propositions in some nondeterministic order.

$$\mathbf{eval}(\mathsf{new}\ (\lambda x.\, E\, x)) \twoheadrightarrow \exists c.\, \mathbf{eval}(E\, c)$$

$$\mathbf{eval}(\mathsf{send}\ C\ E) \twoheadrightarrow \mathbf{comp}(\mathsf{send}_1\ C) \bullet \mathbf{eval}(E)$$
$$\mathbf{comp}(\mathsf{send}_1\ C) \bullet \mathbf{return}(V) \twoheadrightarrow ¡\mathbf{msg}\ C\ V \bullet \mathbf{return}(V)$$

$$\mathbf{eval}(\mathsf{rcv}\ C) \twoheadrightarrow \mathbf{comp}(\mathsf{await}\ C)$$
$$\mathbf{comp}(\mathsf{await}\ C) \bullet ¡\mathbf{msg}\ C\ V \twoheadrightarrow \mathbf{return}(V)$$

Figure 6. Asynchronous communication

Asynchronous communication. As a final modular extension in this line of examples, we add asynchronous communication. We create a new channel with $\mathsf{new}\ (\lambda x.\, e\, x)$ and bind x to the new channel that is created by existential quantification. We send the value of e along a channel x with $\mathsf{send}\ c\ e$ and receive it with $\mathsf{rcv}\ c$. Channels are not first class for the sake of simplicity, but this could easily be extended.

Here, we add to the *linear* context new passive propositions $\mathbf{msg}\ c\ v$. These need to be mobile (instead of ordered) because they may be sent and received from different frames.

4.2 Alternative evaluation strategies

Having looked at a number of language extensions that ordered logic programming can treat in a modular fashion, we will now look at two alternative evaluation strategies for functions and function application. We will start with a *call-by-name* specification and then extend it, first by using destinations to create an environment semantics, and then by using linear propositions to specify *call-by-need* execution.

$$\mathbf{eval}(\mathsf{lam}\ (\lambda x.\, E\, x)) \twoheadrightarrow \mathbf{return}(\mathsf{lam}\ (\lambda x.\, E\, x))$$
$$\mathbf{eval}(\mathsf{app}\ E_1\ E_2) \twoheadrightarrow \mathbf{comp}(\mathsf{app}_1\ E_2) \bullet \mathbf{eval}(E_1)$$
$$\mathbf{comp}(\mathsf{app}_1\ E_2) \bullet \mathbf{return}(\mathsf{lam}\ (\lambda x.\, E_1'\, x)) \\ \quad \twoheadrightarrow \mathbf{eval}(E_1'\ E_2)$$

Figure 7. Call-by-name functions

Call-by-name substitution semantics. The call-by-name specification in Fig. 7 is similar to the call-by-value specification in Fig. 3, but instead of creating an app_2 frame when the function is returned to the frame $\mathsf{app}_1\ e_2$ containing the waiting argument, we immediately substitute the expression e_2 into the function body.

$$\mathbf{eval}(\mathsf{lam}\ (\lambda x.\, E\, x)) \twoheadrightarrow \mathbf{return}(\mathsf{lam}\ (\lambda x.\, E\, x))$$
$$\mathbf{eval}(\mathsf{app}\ E_1\ E_2) \twoheadrightarrow \mathbf{comp}(\mathsf{app}_1\ E_2) \bullet \mathbf{eval}(E_1)$$
$$\mathbf{comp}(\mathsf{app}_1\ E_2) \bullet \mathbf{return}(\mathsf{lam}\ (\lambda x.\, E_1'\, x)) \\ \quad \twoheadrightarrow \exists d_2.\mathbf{eval}(E_1'\, d_2) \bullet !\mathbf{bind}\ d_2\ E_2$$
$$\mathbf{eval}(D) \bullet !\mathbf{bind}\ D\ E \twoheadrightarrow \mathbf{eval}(E)$$

Figure 8. Call-by-name with destinations for variable binding

Call-by-name environment semantics. We modify this specification of call-by-name evaluation to use destinations for binding in Fig. 8. There are only two changes from the specification in Fig. 7. The first change affects the third rule. Instead of substituting an expression e_2 into the function that is being returned to the app_1 frame, we generate a new parameter d_2, substitute *that* into the function that is being returned to the app_1 frame, and then create a new persistent proposition $\mathbf{bind}\ d_2\ e_2$ that associates our new destination with the expression e_2.

In this semantics, then, the persistent context Γ will contain propositions

$$\mathbf{bind}\ d_1\ e_1, \ldots, \mathbf{bind}\ d_n\ e_n$$

binding pairwise distinct destinations to expressions. This context represents the environment of evaluation.

We now are working with expressions containing destinations, and so our second change is to add a rule that specifies how we evaluate a bare parameter. This is done by the last rule in Fig. 8. Whenever we reach a parameter d, we know that that parameter is associated with an expression by some persistent proposition

bind d e, so we switch from evaluating the parameter to evaluating the associated expression.

This is the first time we have needed a persistent predicate (**bind**). A binding is never updated, and may be read more than once or not at all, so the persistent context has exactly the right properties to hold it.

$$\mathbf{eval}(\mathsf{lam}\ (\lambda x.\, E\, x)) \twoheadrightarrow \mathbf{return}(\mathsf{lam}\ (\lambda x.\, E\, x))$$

$$\mathbf{eval}(\mathsf{app}\ E_1\ E_2) \twoheadrightarrow \mathbf{comp}(\mathsf{app}_1\ E_2) \bullet \mathbf{eval}(E_1)$$
$$\mathbf{comp}(\mathsf{app}_1\ E_2) \bullet \mathbf{return}(\mathsf{lam}\ (\lambda x.\, E_1'\, x)) \twoheadrightarrow \exists d_2.\, \mathbf{eval}(E_1'\, d_2) \bullet \text{¡}\mathbf{susp}\ d_2\ E_2$$

$$\mathbf{eval}(D) \bullet \text{¡}\mathbf{susp}\ D\ E \twoheadrightarrow \mathbf{comp}(\mathsf{bind}_1\ D) \bullet \mathbf{eval}(E)$$
$$\mathbf{comp}(\mathsf{bind}_1\ D) \bullet \mathbf{return}(V) \twoheadrightarrow \mathbf{return}(V) \bullet !\mathbf{bind}\ D\ V$$
$$\mathbf{eval}(D) \bullet !\mathbf{bind}\ D\ V \twoheadrightarrow \mathbf{return}(V)$$

Figure 9. Call-by-need functions

Call-by-need. The example in Fig. 9 further modifies the environment semantics for the call-by-need language. The goal in a call-by-need semantics is to evaluate the argument to a function *at most once*, and only if needed. Whereas before we associated destinations with persistent propositions **bind** d e, we now associate them with *either* a *linear* proposition **susp** d e if the argument has not been evaluated or else a *persistent* proposition **bind** d v if the argument has been evaluated.

The first time we try to evaluate a destination (first rule for $\mathbf{eval}(D)$) we force the suspension and install a frame noting to bind the destination to the resulting value. The suspension is linear (and therefore consumed when forced), but the eventual value binding is persistent and will apply to any further reference to D. In this specification, computation gets stuck when a so-called black hole is encountered, because neither of the two $\mathbf{eval}(D)$ clauses applies. We could extend this specification to explicitly detect such a condition and signal an error.

The six lines in the specification represent a complete specification of call-by-need, exploiting order, linearity, and persistence in a pleasing and elegant way. It is consistent with published specifications [11].

4.3 Advanced control with destinations

As a last set of example we consider some advanced control constructs, first exceptions and then first-class continuations. Both violate of the modularity we have sought in the previous examples to different degrees, indicating that adding control constructs to SSOS specifications may involve non-local revisions.

$$\mathbf{eval}(\mathsf{try}\ E_1\ E_2) \twoheadrightarrow \mathbf{catch}(E_2) \bullet \mathbf{eval}(E_1)$$
$$\mathbf{eval}(\mathsf{raise}) \twoheadrightarrow \mathbf{fail}$$
$$\mathbf{comp}(F) \bullet \mathbf{fail} \twoheadrightarrow \mathbf{fail}$$
$$\mathbf{catch}(E_2) \bullet \mathbf{fail} \twoheadrightarrow \mathbf{eval}(E_2)$$
$$\mathbf{catch}(E_2) \bullet \mathbf{return}(V) \twoheadrightarrow \mathbf{return}(V)$$

Figure 10. Exceptions in a sequential language

Exceptions. For simplicity, we only consider one exception which is generated by the raise expression and does not carry a value. try e_1 e_2 evaluates e_1 and returns its value if successful. If evaluating e_1 fails, it evaluates e_2 instead and returns its value. The raising of an exception is modeled by a new passive, ordered proposition **fail** which is percolated up the stack by all frames except by a new latent proposition **catch**.

Exceptions entail a certain violation of modularity if the language specification requires latent propositions waiting on more than one result, as, for example, in parallel evaluation of pairs (Fig. 5).

$$\mathbf{eval}\ D\ (\mathsf{lam}\ (\lambda x.\, E\, x)) \twoheadrightarrow \mathbf{return}\ D\ (\mathsf{lam}\ (\lambda x.\, E\, x))$$
$$\mathbf{eval}\ D\ (\mathsf{app}\ E_1\ E_2) \twoheadrightarrow \exists d_1.\, \text{¡}\mathbf{comp}\ D\ (\mathsf{app}_1\ E_2)\ d_1 \bullet \mathbf{eval}\ d_1\ E_1$$
$$\text{¡}\mathbf{comp}\ D\ (\mathsf{app}_1\ E_2)\ D_1 \bullet \mathbf{return}\ D_1\ V_1 \twoheadrightarrow \exists d_2.\, \text{¡}\mathbf{comp}\ D\ (\mathsf{app}_2\ V_1)\ d_2 \bullet \mathbf{eval}\ d_2\ E_2$$
$$\text{¡}\mathbf{comp}\ D\ (\mathsf{app}_2\ (\mathsf{lam}\ (\lambda x.\, E_1'\, x))\ D_2 \bullet \mathbf{return}\ D_2\ V_2 \twoheadrightarrow \mathbf{eval}\ D\ (E_1'\ V_2)$$

Figure 11. Call-by-value in linear destination-passing style

Linear destinations for threading frames. Looking ahead, we will have to modify our attitude regarding the stack of frames in the ordered context when trying to handle first-class continuations. The difficulty is that the frames on the stack represent the continuation, and when continuations are first-class they may be ignored or used multiple times which goes against the nature of the ordered context.

As a first step, we show how we can represent the stack in the linear context by explicitly threading the frames using destinations. In fact, before the availability of the ordered context, SSOS specifications essentially all used this technique.

The active proposition **eval** d e now takes two arguments: the first is a destination d for its value (which will be a parameter) and the second the expression e to evaluate. Similarly, **return** d v returns value v to destination d. Latent **comp** propositions have two destinations: one on which they receive a value, and one

to which they pass their value; this allows order to be recreated with a linked-list structure. Because **eval** and **return** as well as **comp** now thread destinations, all three may be mobile. However, it is only important that frames be mobile, so we leave **eval** and **return** as the only ordered propositions.

$$\begin{array}{l}
\mathbf{eval}\ D\ (\mathsf{callcc}\ (\lambda x.\, E\, x)) \twoheadrightarrow \mathbf{eval}\ D\ (E\ (\mathsf{cont}\ D)) \\
\mathbf{eval}\ D\ (\mathsf{throw}\ E_1\ E_2) \\
\qquad \twoheadrightarrow \exists d_1.\, !\mathbf{comp}\ D\ (\mathsf{throw}_1\ E_2)\ d_1 \bullet \mathbf{eval}\ d_1\ E_1 \\
!\mathbf{comp}\ D\ (\mathsf{throw}_1\ E_2)\ D_1 \bullet \mathbf{return}\ D_1\ V_1 \\
\qquad \twoheadrightarrow \exists d_2.\, !\mathbf{comp}\ D\ (\mathsf{throw}_2\ V_1)\ d_2 \bullet \mathbf{eval}\ d_2\ E_2 \\
!\mathbf{comp}\ D\ (\mathsf{throw}_2\ (\mathsf{cont}\ D'))\ D_2 \bullet \mathbf{return}\ D_2\ V_2 \\
\qquad \twoheadrightarrow \mathbf{return}\ D'\ V_2 \\
\mathbf{eval}\ D\ (\mathsf{cont}\ D') \twoheadrightarrow \mathbf{return}\ D\ (\mathsf{cont}\ D')
\end{array}$$

Figure 12. First-class continuations

First-class continuations. Linear destinations allow continuations to be represented by a destination that points to the top of a stack. To introduce first-class continuations into our language, we define two new source constructs. The expression $\mathsf{callcc}\ (\lambda x.\, e\, x)$ captures the current continuation and substitutes it for x, and the expression $\mathsf{throw}\ e_1\ e_2$ throws the value of e_2 to the continuation designated by e_1.

This turns out to be quite simple, but it is not modular with respect to the preceding call-by-value specification in linear destination passing style. Languages with first-class continuations may return to a particular frame multiple times, but our previous specifications consumed **comp** propositions as soon as a passive **return** proposition interacted with them. By making continuations persistent, they can be invoked multiple times.

5 Additional related work

Much of the most closely related work is mentioned in-line where appropriate. Regarding the logic programming semantics, an interesting related system is MultiSet Rewriting (MSR) which has been used to analyze security protocols and concurrency in general [3]. It entirely abandons the goal on the right-hand side and works only with linear and persistent contexts, which is similar to our operational semantics. We add ordered contexts to significantly extend the expressive power, as demonstrated by our SSOS specifications.

The work also owes a great deal to Chirimar's linear specifications of operational semantics [4], although the underlying fragment of classical linear logic has, to our knowledge, never received a satisfactory operational interpretation.

One can also take an algebraic attitude towards contexts, thinking, for example, of the ordered context ABC as the term $A \bullet B \bullet C$ where "$\bullet$" is an associative operator, and similarly for the linear context. This is close to the attitude taken by rewriting logic semantics [12]; the latent state that is captured by the ordered and persistent contexts in our work can be encoded within a "soup" of state attributes in rewriting logic specifications. To our knowledge, frameworks based on higher-order rewriting lack support for higher-order abstract syntax but have more sophisticated support for reasoning about equality. It is interesting future work to consider how to combine the strengths of both approaches, especially in considering the use of equalities for semantic approximation [25].

Structural operational semantics (SOS) [18] is an ancestor and inspiration for substructural operational semantics, also taking an algebraic view. It is difficult, however, to achieve the kind of modularity we achieve here since the state is reified as a term, and persistence is somewhat awkward to model. Modularity can be recovered in Modular SOS [14] using variables standing for unspecified future extensions; here modularity stems essentially from monotonicity of logical inference.

The simpler substructural specifications, for example, for call-by-value functions, can be seen as isomorphic to stack-based abstract machines, although expressed in logical and more localized form. This analogy may be more difficult to maintain for the more complex specifications. We view our method a synthesis of ideas behind structural operational semantics, abstract machines, and logical specifications.

6 Conclusion

We have presented a fragment of ordered logic, incorporating ordered, linear, and persistent contexts, that allows a forward-chaining committed choice operational interpretation which is both sound and nondeterministically complete. Additionally exploiting a higher-order term language, we illustrate its expressive power by giving elegant, concise, and to some degree modular specifications of various program language constructs in the style of substructural operational semantics, ranging from simple call-by-value functions to first-class continuations, mutable store, and asynchronous communication. The examples are not exhaustive but, we hope, instructive.

In ongoing work, we are developing an implementation of the ordered logic programming language in order to experiment with SSOS specifications that ex-

ploit the notion of order described here.[1]

We believe that the backward-chaining, backtracking semantics for the negative fragment [19] can be profitably combined with the forward-chaining, committed choice semantics presented here for a very rich fragment of ordered logic, generalizing LolliMon [10] and CLF [26, 2]. At present some difficult questions remain regarding the interface between the two directions. The first-cut solutions proposed and implemented in LolliMon have proved to be inadequate in some cases.

Another long-term goal of ours is to exploit the elegance and uniformity of the SSOS specifications to derive principles for metareasoning so we can mechanically establish properties such as type preservation, progress, bisimulation, etc., perhaps along the lines of work on Twelf [17] and Abella [6].

Acknowledgements. We would like to thank the anonymous reviewers for their comments, Jason Reed for a key insight in the proof of completeness, and Noam Zeilberger for pointing us to Laurent's unpublished note [9].

References

[1] J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *J. Log. Comput.*, 2(3):297–347, 1992.

[2] I. Cervesato, F. Pfenning, D. Walker, and K. Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, School of Computer Science, Carnegie Mellon University, Mar. 2002. Revised May 2003.

[3] I. Cervesato and A. Scedrov. Relating State-Based and Process-Based Concurrency through Linear Logic. *Information & Computation*, 2009.

[4] J. L. Chirimar. *Proof Theoretic Approach to Specification Languages*. PhD thesis, University of Pennsylvania, May 1995.

[5] A. P. Felty and A. Momigliano. Hybrid: A definitional two-level approach to reasoning with higher-order abstract syntax. *CoRR*, abs/0811.4367, 2008.

[6] A. Gacek. The Abella interactive theorem prover. In *IJCAR'08*, pages 154–161. Springer, 2008. System description.

[7] J. S. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Inf. Comput.*, 110(2):327–365, 1994.

[8] J. Lambek. The mathematics of sentence structure. *American Mathematical Monthly*, 65:363–386, 1958.

[9] O. Laurent. A proof of the focalization property of linear logic, May 2004. Unpublished note.

[1] More examples and a prototype implementation are available at `http://www.cs.cmu.edu/~rjsimmon/ssos/`

[10] P. López, F. Pfenning, J. Polakow, and K. Watkins. Monadic concurrent linear logic programming. In *PPDP 2000*, pages 35–46, 2005.

[11] J. Maraist, M. Odersky, and P. Wadler. A call-by-need lambda calculus. *Journal of Functional Programming*, 8(3):275–317, May 1998.

[12] J. Meseguer and G. Roşu. The rewriting logic semantics project. In *SOS 2005*, pages 26–56, May 2006.

[13] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Ann. Pure Appl. Logic*, 51(1-2):125–157, 1991.

[14] P. D. Mosses. Modular structural operational semantics. *J. Log. Algebr. Program.*, 60-61:195–228, 2004.

[15] F. Pfenning. Substructural operational semantics and linear destination-passing style. In *APLAS 2004*, page 196, Taipei, Taiwan, 2004. Springer. Abstract of invited talk.

[16] F. Pfenning and C. Schürmann. Algorithms for equality and unification in the presence of notational definitions. In *Types for Proofs and Programs*, pages 179–193, 1998.

[17] F. Pfenning and C. Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *CADE-16*, pages 202–206, Trento, Italy, 1999. Springer.

[18] G. D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

[19] J. Polakow. Linear logic programming with ordered contexts. In *PPDP 2000*, pages 68–79, Montreal, Canada, Sept. 2000. ACM Press.

[20] J. Polakow. *Ordered Linear Logic and Applications*. PhD thesis, Carnegie Mellon University, Aug. 2001. Available as technical report CMU-CS-01-152.

[21] J. Polakow and F. Pfenning. Natural deduction for intuitionistic non-commutative linear logic. In *TLCA 1999*, pages 295–309, L'Aquila, Italy, 1999. Springer-Verlag LNCS 1581.

[22] J. Polakow and F. Pfenning. Relating natural deduction and sequent calculus for intuitionistic non-commutative linear logic. In *MFPS XV*, Apr. 1999. *Electr. Notes Theor. Comput. Sci.*, 20:449–466.

[23] J. Polakow and K. Yi. Proving syntactic properties of exceptions in an ordered logical framework. In *APLAS 2000*, pages 23–32, 2000.

[24] R. J. Simmons and F. Pfenning. Linear logical algorithms. In *ICALP (2)*, pages 336–347, 2008.

[25] R. J. Simmons and F. Pfenning. Linear logical approximations. In *PEPM 2009*. ACM Press, 2009.

[26] K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, School of Computer Science, Carnegie Mellon University, Mar. 2002. Revised May 2003.

[27] K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. Specifying properties of concurrent computations in CLF. *Electr. Notes Theor. Comput. Sci.*, 199:67–87, 2008.