Carnegie Mellon University
Research Showcase @ CMU

Computer Science Department | School of Computer Science

3-2010

Logical approximation for program analysis

Robert J. Simmons
Carnegie Mellon University

Frank Pfenning
Carnegie Mellon University

Follow this and additional works at: http://repository.cmu.edu/compsci

This Article is brought to you for free and open access by the School of Computer Science at Research Showcase @ CMU. It has been accepted for inclusion in Computer Science Department by an authorized administrator of Research Showcase @ CMU. For more information, please contact research-showcase@andrew.cmu.edu.

Higher-Order and Symbolic Computation manuscript No.
(will be inserted by the editor)

Logical approximation for program analysis

Robert J. Simmons · Frank Pfenning

Received: March 1, 2010 / Accepted: date

Abstract The abstract interpretation of programs relates the exact semantics of a programming language to an approximate semantics that can be effectively computed. We show that, by specifying operational semantics in a specification framework based on bottom-up logic programming in ordered logic – a technique we call *substructural operational semantics* (SSOS) – manifestly sound program approximations can be derived by simple and intuitive transformations and approximations of the logic program. As examples, we describe how to derive control flow and alias analyses from the substructural operational semantics of the relevant languages.

1 Introduction

A central goal of logical frameworks is to specify the operational semantics of evolving systems (and programming languages in particular) in a framework that is both logically motivated and allows specifications to be as simple as possible. A secondary goal, which is the focus of this paper, is to leverage the logical basis to develop precise, yet tractable, approximations of the systems we specify. In particular, we would like to be able to provide program analyses for the programming languages we consider.

Many interesting stateful systems have a natural notion of a *ordering* that is fundamental to their behavior. Consider a push-down automaton (PDA) that reads a string of symbols left-to-right while maintaining and manipulating a separate stack of symbols. We can represent any configuration of the PDA as a string with three regions:

$$\langle \text{ the stack } \rangle \langle \text{ the head } \rangle \langle \text{ the string being read } \rangle$$

where the symbols closest to the head are the top of the stack and the symbol waiting to be read from the string. If we represent the head as a token hd, we can describe

This work was supported by the Fundação para a Ciência e a Tecnologia (FCT), Portugal, under a grant from the Information and Communications Technology Institute (ICTI) at Carnegie Mellon University, and by a National Science Foundation Graduate Resource Fellowship for the first author.

Carnegie Mellon University
Pittsburgh, PA
E-mail: {rjsimmon,fp}@cmu.edu

the behavior of a single-state push-down automaton for checking that a string has matching brackets by using two rewriting rules:

$$\begin{aligned} \textit{push}&: \ \mathsf{hd}\ \texttt{[}\ \rightsquigarrow\ \texttt{[}\ \mathsf{hd} \\ \textit{pop}&: \ \texttt{[}\ \mathsf{hd}\ \texttt{]}\ \rightsquigarrow\ \mathsf{hd} \end{aligned}$$

The distinguishing feature of these rewriting rules is that they are *local* – they do not mention the entire stack or the entire string, just the fragment of the beginning of the string or the top of the stack relevant to the given rule. Execution of the PDA on a particular string of tokens then consists of (1) appending the token hd to the beginning of the string, (2) repeatedly performing rewritings until no more rewrites are possible, and (3) checking to see if only a single token hd remains. To give one possible series of transitions:

```
hd [ [ ] [ [ ] ] ]
[ hd [ ] [ [ ] ] ]
[ [ hd ] [ [ ] ] ]
    [ hd [ [ ] ] ]
    [ [ hd [ ] ] ]
    [ [ [ hd ] ] ]
        [ [ hd ] ]
            [ hd ]
                hd
```

Because our goal is to use a framework that is both simple and logically-motivated, we turn to *ordered logic* (originally presented by Lambek [15]), a substructural logic with an intrinsic notion of order. The rewriting rules we considered before can be expressed as propositions in ordered logic, where the tokens hd, [, and] are all treated as atomic propositions. The symbol • (pronounced "fuse") is the binary connective for ordered conjunction (i.e. concatenation) and has lower precedence than ↠, a binary connective for ordered implication.

$$\begin{aligned} \textit{push}&: \ \mathsf{hd} \bullet \texttt{[}\ \twoheadrightarrow\ \texttt{[} \bullet \mathsf{hd} \\ \textit{pop}&: \ \texttt{[} \bullet \mathsf{hd} \bullet \texttt{]}\ \twoheadrightarrow\ \mathsf{hd} \end{aligned}$$

This framework of *ordered logical specifications* is quite powerful; the above example only uses the propositional fragment of ordered logic. If we consider a first-order term language where terms are drawn from the simply-typed lambda calculus (allowing us to use *higher-order abstract syntax* [24]) we can use ordered logic as a framework for the specification of *substructural operational semantics*, a synthesis of structural operational semantics, abstract machines, and logical specifications in which machine states are represented by a series of ordered atomic propositions.

One especially distinguishing feature of substructural operational semantics in ordered logic is the treatment of control stacks. Abstract machine specifications of programming language semantics are traditionally specified with states of the form $(K \rhd E)$, representing an expression E evaluating on the control stack K, and $(K \lhd V)$, representing a value V being returned to the stack. In substructural operational semantics, as in the push-down automaton, we represent the stack K not as a single syntactic object but as a sequence of atomic propositions $\mathsf{comp}(F)$, each of which contain a single stack frame.

We will give the example of a call-by-value operational semantics for the lambda calculus. The evaluation of a lambda expression, which we write in higher order abstract syntax as $\mathsf{lam}(\lambda x.E_0\,x)$, is simple: the expression is already a value, so we return it.

$$\mathsf{eval}(\mathsf{lam}(\lambda x.E_0\,x)) \;\twoheadrightarrow\; \mathsf{retn}(\mathsf{lam}(\lambda x.E_0\,x)) \tag{1}$$

The evaluation of an application $\mathsf{app}\ E_1\ E_2$ requires us to generate a new stack frame $(\mathsf{app}_1\ E_2)$ containing the function argument while we evaluate E_1 to a value.

$$\mathsf{eval}(\mathsf{app}\ E_1\ E_2) \;\twoheadrightarrow\; \mathsf{comp}(\mathsf{app}_1\ E_2) \;\bullet\; \mathsf{eval}(E_1) \tag{2}$$

When a value is returned to a waiting app_1 frame, we switch to evaluating the function argument while storing the returned value on the stack.

$$\mathsf{comp}(\mathsf{app}_1\ E_2) \;\bullet\; \mathsf{retn}(V_1) \;\twoheadrightarrow\; \mathsf{comp}(\mathsf{app}_2\ V_1) \;\bullet\; \mathsf{eval}(E_2) \tag{3}$$

Finally, when an evaluated function argument is returned to a waiting app_2 frame, we substitute the value into the body of the lambda expression and evaluate the result. As usual in higher-order abstract syntax representations, substitution is performed by application.

$$\mathsf{comp}(\mathsf{app}_2(\mathsf{lam}(\lambda x.E_0\,x))) \;\bullet\; \mathsf{retn}(V_2) \;\twoheadrightarrow\; \mathsf{eval}(E_0\,V_2) \tag{4}$$

These four rules constitute a substructural operational semantics specification of the call-by-value lambda calculus; an example of the evaluation of an expression to a value under this specification is given in Figure 1. SSOS specifications in ordered logic are conceptually simple, notationally clean, and provide a modular basis for the specification of many stateful and concurrent programming language features, as discussed in [26].

It is in the context of these ordered SSOS specifications that we will describe a methodology of *logical approximation*, exploiting the logical basis of our SSOS specifications to describe approximations which are correct by construction. We specify a series of general transformations for ordered and linear logical specifications that allow

$$\begin{array}{lll}
 & & \mathsf{eval}(\mathsf{app}\ (\mathsf{lam}(\lambda x.x))\,(\mathsf{app}\ (\mathsf{lam}(\lambda y.y))\,(\mathsf{lam}(\lambda z.e)))) \\
 & \mathsf{comp}(\mathsf{app}_1(\mathsf{app}\ (\mathsf{lam}(\lambda y.y))\,(\mathsf{lam}(\lambda z.e)))) & \mathsf{eval}(\mathsf{lam}(\lambda x.x)) \\
 & \mathsf{comp}(\mathsf{app}_1(\mathsf{app}\ (\mathsf{lam}(\lambda y.y))\,(\mathsf{lam}(\lambda z.e)))) & \mathsf{retn}(\mathsf{lam}(\lambda x.x)) \\
 & \mathsf{comp}(\mathsf{app}_2(\mathsf{lam}(\lambda x.x))) & \mathsf{eval}(\mathsf{app}\ (\mathsf{lam}(\lambda y.y))\,(\mathsf{lam}(\lambda z.e))) \\
\mathsf{comp}(\mathsf{app}_2(\mathsf{lam}(\lambda x.x))) & \mathsf{comp}(\mathsf{app}_1(\mathsf{lam}(\lambda z.e))) & \mathsf{eval}(\mathsf{lam}(\lambda y.y)) \\
\mathsf{comp}(\mathsf{app}_2(\mathsf{lam}(\lambda x.x))) & \mathsf{comp}(\mathsf{app}_1(\mathsf{lam}(\lambda z.e))) & \mathsf{retn}(\mathsf{lam}(\lambda y.y)) \\
\mathsf{comp}(\mathsf{app}_2(\mathsf{lam}(\lambda x.x))) & \mathsf{comp}(\mathsf{app}_2(\mathsf{lam}(\lambda y.y))) & \mathsf{eval}(\mathsf{lam}(\lambda z.e)) \\
\mathsf{comp}(\mathsf{app}_2(\mathsf{lam}(\lambda x.x))) & \mathsf{comp}(\mathsf{app}_2(\mathsf{lam}(\lambda y.y))) & \mathsf{retn}(\mathsf{lam}(\lambda z.e)) \\
 & \mathsf{comp}(\mathsf{app}_2(\mathsf{lam}(\lambda x.x))) & \mathsf{eval}(\mathsf{lam}(\lambda z.e)) \\
 & \mathsf{comp}(\mathsf{app}_2(\mathsf{lam}(\lambda x.x))) & \mathsf{retn}(\mathsf{lam}(\lambda z.e)) \\
 & & \mathsf{eval}(\mathsf{lam}(\lambda z.e)) \\
 & & \mathsf{retn}(\mathsf{lam}(\lambda z.e))
\end{array}$$

Fig. 1 An trace of the intermediate steps in a call-by-value evaluation of the untyped lambda calculus term $(\lambda x.x)((\lambda y.y)(\lambda z.e))$ under the SSOS specification given by rules 1-4.

us to derive a rich family of approximations to those specifications. While defining specific approximations may require insight, the correctness proofs should not; rather, they should follow from a general metatheorem justifying the kinds of approximations we make, together with straightforward termination arguments.

Our approach to the logical approximation of ordered SSOS specifications proceeds in two stages. First, we transform our ordered SSOS specifications into linear SSOS specifications where the adjacency information is represented explicitly. This transformation, which allows all ordered logical specification to be adequately expressed in linear logic, naturally gives rise to a style of SSOS specification called *linear destination-passing style*. Second, we describe a family of simple and general techniques for *approximating* ordered and linear logical specifications as specifications in persistent logics. The resulting approximations are now (non-linear) bottom-up logic programs which can be run to saturation, generalizing proposals by McAllester and Ganzinger [17, 11] with certain higher-order features.

These two steps, the transformation from ordered to linear logic and the approximation as a bottom-up logic program, work hand in hand. We could apply the general approximation methodology directly to ordered logical specifications, but in the case of ordered SSOS specifications, the loss of adjacency information that this would entail would result in approximations that are too coarse. We could (more realistically) ignore ordered logic and write our SSOS specifications directly in linear destination-passing style; this is, in fact, what was done for an earlier version of this paper [32]. However, and our experience with SSOS specification in ordered logic [26] has convinced us that the natural expression of adjacency in ordered logic really is an important element in the specification of programming languages.

In Section 2 we revisit the use of ordered logic as a specification framework and the transformation of ordered logical specifications into linear logical specifications. In Section 3 we discuss the approximation of ordered logic programs by saturating bottom-up logic programs. In Section 4 we apply these techniques to derive a control flow analysis and an alias analysis by logical approximation.

2 Ordered and linear logical specifications

The framework of *ordered logical specifications* that we present is a forward-chaining (or "bottom-up") logic programming language based on ordered logic. This framework is unlike most previous work in logic programming languages for substructural logics.[1] Traditionally, substructural logics have been treated as logic programming languages by giving a backward-chaining (or "top-down") operational semantics (in the style of Prolog) to the *uniform fragment* of the language – essentially the propositions freely generated by all connectives with invertible right rules [20]. Backward chaining operational semantics have been given to the uniform fragment of linear logic [13], ordered logic [28, 29], and bunched logic [4].

Appropriate fragments of forward-chaining logic programming are less straightforward to isolate. The fragment of ordered logic we consider is split into a *negative*

[1] Ordered logic and the more well-known linear logic are called *substructural* logics because they restrict the usual structural rules of contraction, weakening, and exchange. The term *substructural operational semantics* merges structural operational semantics [27], which we seek to generalize, and substructural logics, which we use as our specification framework.

fragment of right-invertible connectives and a *positive* fragment of left-invertible connectives:

$$\begin{array}{lll} \textit{Atomic Props.} & Q ::= p\ t_1 \dots t_n \\ \textit{Neg. Props.} & A ::= \forall x.A \mid S_1 \twoheadrightarrow S_2 \\ \textit{Pos. Props.} & S ::= Q \mid ¡Q \mid !Q \mid S_1 \bullet S_2 \mid \exists x.S \mid \mathbf{1} \mid t = s \end{array}$$

In the examples in the introduction we saw the use of ordered implication $S_1 \twoheadrightarrow S_2$ (where we call S_1 the *premise* and S_2 the *conclusion*), ordered conjunction $S_1 \bullet S_2$, and ordered atomic propositions Q. In addition, our language includes $\mathbf{1}$, the unit of ordered conjunction, existential quantification $\exists x.S$ and universal quantification $\forall x.S$. Following a common convention we will treat uppercase variables as implicitly universally quantified: this is why rule 2 begins with "$\mathsf{eval}(\mathsf{app}\ E_1\ E_2)\dots$" instead of "$\forall e_1.\, \forall e_2.\, \mathsf{eval}(\mathsf{app}\, e_1\, e_2)\dots$." We also have *linear* atomic propositions $¡Q$, and *persistent* atomic propositions $!Q$. Persistent atomic propositions act like normal mathematical facts – when we assert a persistent atomic proposition in the conclusion of a rule, it stays true for the rest of the evolution of the system, so when we match against a persistent fact in the premise of a rule, the application of the rule does not remove that fact from the relevant set of facts. Linear atomic propositions, on the other hand, are ephemeral but not ordered: when they appear in the premise of a rule, the transition produced by the rule will *remove* the linear proposition from the multiset of linear atomic propositions. These connectives form the language of ordered logical specifications described in previous work [26]; we extend that language with a proposition $(t = s)$ for equality between terms, which we will explains shortly.

We can define the meaning of propositions in ordered logic using a sequent calculus presentation of the logic. The judgment associated with ordered propositions is $\Gamma; \Delta; \Omega \vdash_\Sigma A$, which can be read as "under the persistent hypotheses Γ (which are subject to the usual rules of exchange, contraction, and weakening), the linear hypotheses Δ (which do not admit contraction or weakening, only exchange), and the ordered hypotheses Ω (which admit none of these structural rules), A is true." We frequently elide the free parameters Σ.

To give an example, ordered conjunction is defined by the following rules:

$$\frac{\Gamma; \Delta_A; \Omega_L \vdash A \quad \Gamma; \Delta_B; \Omega_R \vdash B}{\Gamma; \Delta_A \bowtie \Delta_B; \Omega_L \Omega_R \vdash A \bullet B}\ \bullet_R \qquad \frac{\Gamma; \Delta; \Omega_L A B \Omega_R \vdash C}{\Gamma; \Delta; \Omega_L (A \bullet B) \Omega_R \vdash C}\ \bullet_L$$

Here $\Delta_A \bowtie \Delta_B$ denotes a nondeterministic merge operation for linear contexts Δ_A and Δ_B. Read in a bottom-up way, the rule $\bullet_R$ suggests that we can attempt to prove the conclusion $A \bullet B$ by separating out the linear context into two parts Δ_A and Δ_B and splitting the ordered context into a left part Ω_L and a right part Ω_R: in addition to the persistent assumptions in Γ, we must use Δ_A and Ω_L to prove A and Δ_B and Ω_R to prove B. Similarly, the rule $\bullet_L$, says that an ordered hypothesis $A \bullet B$ can be replaced in-place with an ordered hypothesis A followed by an ordered hypothesis B.

We will not reprise the full sequent calculus for ordered logic (see [29] or [26] or reconstruct it from the weakly focused rules in Figure 2), but we do extend the presentations there with rules for equality. These rules depend on a notion of *complete sets of unifiers* $\mathrm{csu}(\Sigma, t, s)$. A complete set of unifiers for two terms t and s with free parameters Σ is a set of substitutions for the free parameters Σ with two properties. First, every substitution θ in the set is a unifier of t and s, meaning θt and θs are syntactically identical up to renaming bound variables. Second, every unifier σ of t and s can be represented as the composition of some θ in the complete set of unifiers and

another substitution τ. The $=_L$ rule can have zero or more premises depending on the size of the complete set of unifiers:

$$\frac{}{\Gamma;\cdot;\cdot \vdash_\Sigma t = t}\ =_R \qquad \frac{\{\theta\Gamma;\theta\Delta;\theta\Omega_L\theta\Omega_R \vdash_{\Sigma'} \theta C : (\Sigma' \vdash \theta) \in \mathrm{csu}(\Sigma, t, s)\}}{\Gamma;\Delta;\Omega_L(t = s)\Omega_R \vdash_\Sigma C}\ =_L$$

Here we write $\Sigma' \vdash \theta$ for a unifier θ with free variables in Σ'. For general higher-order unification, calculating a complete set of unifiers $\mathrm{csu}(\Sigma, t, s)$ is undecidable. However, in the higher-order pattern fragment (which includes all first-order unification problems), it is not only decidable but unitary: either there are no unifiers of t and s or there is a unique *most general unifier*. On this fragment, then, we can proceed as if using derived rules $\neq_L$ and $=_{L\mathrm{mgu}}$:

$$\frac{\text{there is no } \mathrm{mgu}(\Sigma, t, s)}{\Gamma;\Delta;\Omega_L(t = s)\Omega_R \vdash_\Sigma C}\ \neq_L \quad \frac{\Sigma' \vdash \theta = \mathrm{mgu}(\Sigma, t, s) \quad \theta\Gamma;\theta\Delta;\theta\Omega_L\,\theta\Omega_R \vdash_{\Sigma'} \theta C}{\Gamma;\Delta;\Omega_L(t = s)\Omega_R \vdash_\Sigma C}\ =_{L\mathrm{mgu}}$$

This definition of equality based on unification follows [5] but is rather uncontroversial and dates back to Girard and Schroeder-Heister [12,30].

2.1 From ordered logic to ordered logical specifications

Now that we have briefly discussed the sequent calculus presentation for the relevant fragment of ordered logic, we can define the framework of *ordered logical specifications* by endowing that fragment with a forward-chaining operational semantics. We ultimately want to treat closed negative propositions as something like rewriting instructions. We call closed negative proposition *rules* and collections of rules *programs* (or *specifications*) if they obey the following four conditions:

- *Range restriction.* Any variables appearing in the premise of a rule must have one *strict occurrence* [25], and every variable occurring in the conclusion of a rule must appear in a premise or as one of the existentially bound parameters in the conclusion. This ensures that higher-order matching is unitary and decidable so that we can always decide whether a particular rule may be applied [31].
- *Pattern unification:* In order to ensure that all occurrences of $=_L$ will have at most one premise, we require that every unification problem that appears in the conclusion of a rule fall in the *pattern fragment* [18].[2]
- *Predicate separation:* Each predicate must be used consistently, appearing either only in ordered, linear, or persistent atomic propositions in a given program. Therefore we can speak of predicates and propositions as being inherently ordered, linear, or persistent.
- *Rule separation:* A rule with ordered conclusions must have ordered premises, and a rule with linear premises must have either linear or ordered premises.

Given these restrictions, the operational semantics of ordered logical algorithms are given by way of a focused sequent calculus in Figure 2. Focusing, introduced by Andreoli [3], is a restriction of the sequent calculus that forces derivations to alternate between two different phases – *focused* phases where non-invertible rules are applied

[2] We do not discuss the interesting question of how to ensure this in general. In our development, we use equality only ways that straightforwardly satisfy this condition.

Initial Rules

$$\frac{}{\Gamma;\cdot;Q \Rightarrow_\Sigma [Q]}\ \textbf{init} \qquad \frac{}{\Gamma;Q;\cdot \Rightarrow_\Sigma [¡Q]}\ \textbf{init}_¡ \qquad \frac{}{\Gamma_L Q \Gamma_R;\cdot;\cdot \Rightarrow_\Sigma [!Q]}\ \textbf{init}!$$

Focusing Rules

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

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

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

$$\frac{}{\Gamma;\cdot;\cdot \Rightarrow [\mathbf{1}]}\ \mathbf{1}_R \qquad \frac{\Gamma;\Delta;\Omega_L\Omega_R \Rightarrow S'}{\Gamma;\Delta;\Omega_L(\mathbf{1})\Omega_R \Rightarrow S'}\ \mathbf{1}_L$$

Modalities

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

Quantifiers

$$\frac{\Gamma;\Delta;\Omega_L[A[t/x]]\Omega_R \Rightarrow_\Sigma S'}{\Gamma;\Delta;\Omega_L[\forall x.A]\Omega_R \Rightarrow_\Sigma S'}\ \forall_L$$

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

Equality

$$\frac{}{\Gamma;\cdot;\cdot \Rightarrow_\Sigma [t=t]}\ =_R \qquad \frac{\{\theta(\Gamma;\Delta;\Omega_L\Omega_R \vdash_\Sigma C) : \theta \in \mathrm{csu}(\Sigma,t,s)\}}{\Gamma;\Delta;\Omega_L(t=s)\Omega_R \Rightarrow_\Sigma S}\ =_L$$

Fig. 2 A weakly focused sequent calculus the fragment of ordered logic that makes up the basis of ordered logical specifications. Order matters for the context Ω; all other contexts are treated as equivalent up to reordering.

as far as possible, and *inversion* phases where invertible rules are applied as far as possible. Our presentation is *weakly focused*, forcing focused phases but not invertible phases to be applied fully. There are three kinds of sequents in the weakly focused presentation:

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

In the focused presentation, Ω contains only positive propositions and ordered atomic propositions, Δ contains only linear atomic propositions, and Γ contains only rules and persistent atomic propositions. The operational semantics of ordered logical specifications is described in terms of these focused sequents. A *state* is an unfocused sequent $\mathbb{S} = (\Gamma_\mathcal{P}\Gamma;\Delta;\Omega \Rightarrow_\Sigma S)$ where the contexts Γ, Δ, and Ω contain only (persistent, linear, and ordered, respectively) atomic propositions, $\Gamma_\mathcal{P}$ is a program, and the conclusion S is a closed positive proposition.

2.1.1 State transitions

We write "$(\Gamma_{\mathcal{P}}\Gamma; \Delta; \Omega \Rightarrow_\Sigma S) \xrightarrow{-+} (\Gamma_{\mathcal{P}}\Gamma'; \Delta'; \Omega' \Rightarrow_{\Sigma'} S)$" (or "$\mathbb{S}_1 \xrightarrow{-+} \mathbb{S}_2$") if $\Omega = \Omega_L\Omega_R$ and there is a sequential derivation of the following form (where $A \in \Gamma_{\mathcal{P}}$):

$$\dfrac{\begin{array}{c}\Gamma_{\mathcal{P}}\Gamma'; \Delta'; \Omega' \Rightarrow_{\Sigma'} S \\ \mathcal{D} \end{array}}{\Gamma_{\mathcal{P}}\Gamma; \Delta; \Omega_L[A]\Omega_R \Rightarrow_\Sigma S}$$

By a *sequential* derivation we mean that the derivation obeys two conditions:

- The assumed sequent $(\Gamma_{\mathcal{P}}\Gamma'; \Delta'; \Omega' \Rightarrow_{\Sigma'} S)$ is used exactly once in the partial derivation. This amounts to requiring that the conclusion S is not proved using right focus and that every occurrence of $=_L$ uses a most general unifier and therefore has exactly one premise.
- The derivation does not include the rule $\mathbf{focus}_L$. This requirement forces the sequential derivation to consist of a partial derivation $\mathcal{D}_{inv}$ using only left invertible rules ($\bullet_L$, $\mathbf{1}_L$, $!_L$, $¡_L$, $\exists_L$, and $=_L$) above a partial derivation $\mathcal{D}_{foc}$ using only the rules acting on left or right focused sequents (**init**, **init**¡, **init**!, $\twoheadrightarrow_L$, $\bullet_R$, $\mathbf{1}_R$, $\forall_L$, $\exists_R$, and $=_R$).

The parameters Σ' in the assumed sequent can be different from the parameters Σ in the conclusion due to new parameters introduced by applications of the $\exists_L$ rule or due to the set of parameters being modified by the application of a most general unifier in the $=_L$ rule.

We write "$(\Gamma_{\mathcal{P}}\Gamma; \Delta; \Omega \Rightarrow_\Sigma S) \xrightarrow{-+} \bot$" (and say that the state $\Gamma_{\mathcal{P}}\Gamma; \Delta; \Omega \Rightarrow_\Sigma S$ aborts) if $\Omega = \Omega_L\Omega_R$ and if there is a complete derivation of the following form (where $A \in \Gamma_{\mathcal{P}}$):

$$\dfrac{\begin{array}{c}\mathcal{D}_{inv} \\ \mathcal{D}_{foc} \end{array}}{\Gamma_{\mathcal{P}}\Gamma; \Delta; \Omega_L[A]\Omega_R \Rightarrow_\Sigma S}$$

where the *complete* derivation $\mathcal{D}_{inv}$ uses only invertible rules ($\bullet_L$, $\mathbf{1}_L$, $!_L$, $¡_L$, $\exists_L$, and $=_L$), and where the sequential derivation $\mathcal{D}_{foc}$ uses only the rules acting on left or right focused sequents (**init**, **init**¡, **init**!, $\twoheadrightarrow_L$, $\bullet_R$, $\mathbf{1}_R$, $\forall_L$, $\exists_R$, and $=_R$). Note that $=_L$ is the only left invertible rule that can have less than one premise, and only then if there is a contradictory unification goal. This can only arise if a contradictory unification goal is introduced in the conclusion of a rule.

Finally, we write "$(\Gamma_{\mathcal{P}}\Gamma; \Delta; \Omega \Rightarrow_\Sigma S) \rightarrow|$" if there is a complete derivation of the following form:

$$\dfrac{\mathcal{D}}{\Gamma_{\mathcal{P}}\Gamma; \Delta; \Omega \Rightarrow [S]}$$

This derivation necessarily stays entirely within right-focused sequents. In fact, it is decidable if $(\Gamma_{\mathcal{P}}\Gamma; \Delta; \Omega \Rightarrow_\Sigma S) \rightarrow|$ holds because the required higher-order matching is decidable.

The connection between ordered logic and the language of ordered logical specifications is established by Theorem 1.

Theorem 1 (Nondeterministic completeness) *If the contexts Γ_0, Δ_0, and Ω_0 contain only (persistent, linear, and ordered, respectively) atomic propositions, and if the context $\Gamma_{\mathcal{P}}$ contains only rules obeying predicate separation and the restriction of*

equality to pattern unification, then there is a derivation of $(\Gamma_{\mathcal{P}}\Gamma_0; \Delta_0; \Omega_0 \vdash_{\Sigma_0} S)$ *in ordered logic if and only if there exists a sequence of transitions*

$$(\Gamma_{\mathcal{P}}\Gamma_0; \Delta_0; \Omega_0 \Rightarrow_{\Sigma_0} S) = \mathbb{S}_0 \xrightarrow{-+} \mathbb{S}_1 \xrightarrow{-+} \ldots \xrightarrow{-+} \mathbb{S}_n$$

under the rules in $\Gamma_{\mathcal{P}}$ *such that either* $\mathbb{S}_n \rightarrow|$ *or* $\mathbb{S}_n \xrightarrow{-+} \bot$.

Proof Immediate consequence of Theorems 2 and 4 in [26] appropriately extended with the additional sequent calculus rules for equality.

Note that the nondeterminstic completeness of ordered logic programs only requires two of the four properties of programs: predicate separation and the restriction of equality to pattern unification. Range restriction is necessary in order to ensure that, given a state $\mathbb{S}$, we can effectively enumerate all states $\mathbb{S}'$ such that $\mathbb{S} \xrightarrow{-+} \mathbb{S}'$ and determine whether $\mathbb{S}_n \xrightarrow{-+} \bot$.

2.1.2 Notational Conventions

Throughout the paper, we will frequently be interested in the *linear fragment* – programs where the ordered context Ω starts off empty, and in the *persistent fragment* – programs where both the ordered context Ω and the linear context Δ start off empty. The restriction of rule separation ensures that states that start in the linear or persistent fragment will only evolve to states in the linear or persistent fragment, respectively.

Because writing $(\mathsf{edge}\, X\, Y \wedge \mathsf{path}\, Y\, Z \supset \mathsf{path}\, X\, Z)$ is a bit more familiar and less cluttered than writing $(!\mathsf{edge}\, X\, Y \bullet !\mathsf{path}\, Y\, Z \twoheadrightarrow !\mathsf{path}\, X\, Z)$, we obey a convention that if we are unambiguously talking about the persistent fragment we will use $\wedge$ instead of $\bullet$, $\supset$ instead of $\twoheadrightarrow$, and Q instead of $!Q$. Similarly, when we are unambiguously talking about the linear fragment, we will use $\otimes$ instead of $\bullet$, $\multimap$ instead of $\twoheadrightarrow$, and Q instead of ¡Q. We claim that a similarly-defined forward-chaining logic programming semantics for a framework of linear or persistent logical specifications would correspond exactly to the semantics that we get by expanding the definitions into the framework of ordered logical specifications. However, to avoid reprising the development in the previous section (twice!) we will not address that claim here.

2.1.3 Equivalence of Rules

Theorem 1 means that any derivation in our fragment of ordered logic can be characterized by a series of transitions driven by the rules $A \in \Gamma_{\mathcal{P}}$. This means that the impact of a rule A can be entirely characterized by the transitions $\mathbb{S} \xrightarrow{-+} \mathbb{S}'$ and $\mathbb{S} \xrightarrow{-+} \bot$ that it gives rise to. This in turn gives rise to a natural notion of equivalence between rules: two rules are equivalent when they give rise to the same transitions. The same idea appears in other focused proof systems as Andreoli's *bipoles* [2] and Chaudhuri's *derived rules* [10].

This becomes particularly relevant when we deal with the introduction of parameters by existential quantification which are are then "bound" using equality. For example, the following two rules are equivalent:

$$\mathsf{at}(X) \supset \mathsf{next}(X, \mathsf{s}(X)) \wedge \mathsf{at}(\mathsf{s}(X)) \tag{5}$$

$$\mathsf{at}(X) \supset \exists y.\ y = \mathsf{s}(X) \wedge \mathsf{next}(X, y) \wedge \mathsf{at}(y) \tag{6}$$

Both rules lead from a state where $\mathsf{at}(t)$ (for arbitrary t) is in the persistent context to a state where $\mathsf{next}(t, \mathsf{s}(t))$ and $\mathsf{at}(\mathsf{s}(t))$ are also in the persistent context. We can see this is the case for the second rule by looking at the sequential derivation that the rule gives rise to.

$$\dfrac{\dfrac{}{\Gamma, \mathsf{at}(t); \cdot; \cdot \Rightarrow_\Sigma [!\mathsf{at}(t)]}\ \mathbf{init}! \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, \mathsf{at}(t), \mathsf{next}(t, \mathsf{s}(t)), \mathsf{at}(\mathsf{s}(t)); \Delta; \Omega_L \Omega_R \Rightarrow_\Sigma S}{\Gamma, \mathsf{at}(t), \mathsf{next}(t, \mathsf{s}(t)); \Delta; \Omega_L (!\mathsf{at}(\mathsf{s}(t)))\Omega_R \Rightarrow_\Sigma S}\ !_L}{\Gamma, \mathsf{at}(t); \Delta; \Omega_L (!\mathsf{next}(t, \mathsf{s}(t)))(!\mathsf{at}(\mathsf{s}(t)))\Omega_R \Rightarrow_\Sigma S}\ !_L}{\Gamma, \mathsf{at}(t); \Delta; \Omega_L (y = \mathsf{s}(t))(!\mathsf{next}(t, y))(!\mathsf{at}(y))\Omega_R \Rightarrow_{\Sigma, y} S}\ =_L}{\Gamma, \mathsf{at}(t); \Delta; \Omega_L (y = \mathsf{s}(t))(!\mathsf{next}(t, y) \bullet !\mathsf{at}(y))\Omega_R \Rightarrow_{\Sigma, y} S}\ \bullet_L}{\Gamma, \mathsf{at}(t); \Delta; \Omega_L (y = \mathsf{s}(t) \bullet !\mathsf{next}(t, y) \bullet !\mathsf{at}(y))\Omega_R \Rightarrow_{\Sigma, y} S}\ \bullet_L}{\Gamma, \mathsf{at}(t); \Delta; \Omega_L (\exists y.\, y = \mathsf{s}(t) \bullet !\mathsf{next}(t, y) \bullet !\mathsf{at}(y))\Omega_R \Rightarrow_\Sigma S}\ \exists_L}{\Gamma, \mathsf{at}(t); \Delta; \Omega_L [!\mathsf{at}(t) \twoheadrightarrow \exists y.\, y = \mathsf{s}(t) \bullet !\mathsf{next}(t, y) \bullet !\mathsf{at}(y)]\Omega_R \Rightarrow_\Sigma S}\ \twoheadrightarrow_L}{\Gamma, \mathsf{at}(t); \Delta; \Omega_L [\forall x.\, !\mathsf{at}(x) \twoheadrightarrow \exists y.\, y = \mathsf{s}(x) \bullet !\mathsf{next}(x, y) \bullet !\mathsf{at}(y)]\Omega_R \Rightarrow_\Sigma S}\ \forall_L}{\Gamma, \mathsf{at}(t); \Delta; \Omega_L \Omega_R \Rightarrow_\Sigma S}\ \mathbf{focus}_L$$

2.2 Transformation into the linear fragment

As we mentioned in the Section 2.1.2, for the purposes of this paper we define linear logical specifications in terms of ordered logical specifications – we furthermore claim that this straightforwardly and unsurprisingly corresponds to a direct definition of linear logic programming. A somewhat more surprising result is that we can faithfully transform ordered logical specifications into linear logical specifications.[3] As discussed in the introduction, this transformation makes adjacency information (which amounts to control flow information) explicit in an SSOS specification. This in turn allows program analyses derived by approximation to utilize information about control flow that would be unavailable if we approximated an ordered SSOS specification directly. Therefore, the correctness of the transformation ensures that we can use the less cluttered style of substructural operational semantics in ordered logic without sacrificing control flow information in the analyses we derive by logical approximation.

We write the translation of a state $\mathbb{S}_i$ into the linear fragment as $[\![\mathbb{S}_i]\!]_{\mathbb{S}}$. Transformed programs may generate spurious extra parameters, so we write $[\![\mathbb{S}_i]\!]^+_{\mathbb{S}}$ to describe the transformation of a state $\mathbb{S}_i$ that also includes free parameters that do not appear in $\mathbb{S}_i$ or $[\![\mathbb{S}_i]\!]_{\mathbb{S}}$. A correct transformation must be one where a transformed state exhibits precisely the same transitions as an untranslated state, as expressed by Theorem 2.

Theorem 2 (Correctness of transformation) *For any three states*

- $\mathbb{S} = (\Gamma_{\mathcal{P}} \Gamma; \Delta; \Omega \Rightarrow_\Sigma S)$,
- $\mathbb{S}_o = (\Gamma_{\mathcal{P}} \Gamma_o; \Delta_o; \Omega_o \Rightarrow_{\Sigma_o} S)$, *and*
- $\mathbb{S}_l = (\Gamma_{\mathcal{P}} \Gamma_l; \Delta_l; \cdot \Rightarrow_{\Sigma_l} S)$,

we have that

- $[\![\mathbb{S}]\!]_{\mathbb{S}} \xrightarrow{-+} \mathbb{S}_l$ *if and only if* $\mathbb{S} \xrightarrow{-+} \mathbb{S}_o$ *and* $\mathbb{S}_l = [\![\mathbb{S}_o]\!]^+_{\mathbb{S}}$,

[3] Moot and Piazza previously noted a closely related result, a correspondence between provability in the propositional Lambek calculus and first-order linear logic [22].

$$
\begin{aligned}
[\![\forall x.A]\!] &= \forall x.[\![A]\!] \\
[\![S_1 \twoheadrightarrow S_2]\!] &= \forall d_L.\forall d_R.[\![S_1]\!]^{d_L}_{d_R} \multimap [\![S_2]\!]^{d_L}_{d_R} \\
[\![S]\!] &= \exists d_L.\exists d_R.[\![S]\!]^{d_L}_{d_R} \\
[\![Q]\!]^{d_L}_{d_R} &= Q(d_L, d_R) \\
[\![¡Q]\!]^{d_L}_{d_R} &= Q \otimes d_L = d_R \\
[\![!Q]\!]^{d_L}_{d_R} &= !Q \otimes d_L = d_R \\
[\![S_1 \bullet S_2]\!]^{d_L}_{d_R} &= \exists d_M.[\![S_1]\!]^{d_L}_{d_M} \otimes [\![S_2]\!]^{d_M}_{d_R} \\
[\![\exists x.S]\!]^{d_L}_{d_R} &= \exists x.[\![S]\!]^{d_L}_{d_R} \\
[\![\mathbf{1}]\!]^{d_L}_{d_R} &= d_L = d_R \\
[\![t = s]\!]^{d_L}_{d_R} &= t = s \otimes d_L = d_R \\
[\![\Gamma_{\mathcal{P}} \Gamma; \Delta; \Omega \Rightarrow_\Sigma S]\!]_{\mathbb{S}} &= ([\![\Gamma_{\mathcal{P}}]\!]\Gamma; \Delta[\![\Omega]\!]^{d_0}_{d_n}; \cdot \Rightarrow_{\Sigma d_0 \ldots d_n} [\![S]\!]) \\
[\![\Gamma_{\mathcal{P}} \Gamma; \Delta; \Omega \Rightarrow_\Sigma S]\!]^+_{\mathbb{S}} &= ([\![\Gamma_{\mathcal{P}}]\!]\Gamma; \Delta[\![\Omega]\!]^{d_0}_{d_n}; \cdot \Rightarrow_{\Sigma \Sigma_w d_0 \ldots d_n} [\![S]\!])
\end{aligned}
$$

Fig. 3 Translation of propositions and states from ordered logic into the linear fragment.

- $[\![\mathbb{S}]\!]_{\mathbb{S}} \xrightarrow{-+} \bot$ *if and only if* $\mathbb{S} \xrightarrow{-+} \bot$, *and*
- $[\![\mathbb{S}]\!]_{\mathbb{S}} \rightarrow|$ *if and only if* $\mathbb{S} \rightarrow|$.

In the remainder of this section we will present the translation (Section 2.2.1) and then discuss the proof of Theorem 2 (Section 2.2.2).

2.2.1 Definitions

Figure 3 presents the transformation of propositions and sequents into the linear fragment. Linear and persistent atomic propositions and contexts are unchanged by the transformation, but we translate ordered atomic propositions $Q = p\ t_1 \ldots t_n$ into linear atomic propositions by giving them two extra arguments: $Q(d_L, d_R) = p\ t_1 \ldots t_n\ d_L\ d_R$. We translate positive propositions S as $[\![S]\!]^{d_L}_{d_R}$, where it is permissible for d_L to be the same as d_R – in other words, the transformation has three inputs. We translate ordered contexts $\Omega = S_1 \ldots S_n$ by introducing $n + 1$ distinct parameters: $[\![\Omega]\!]^{d_0}_{d_n} = [\![S_1]\!]^{d_0}_{d_1} \ldots [\![S_n]\!]^{d_{n-1}}_{d_n}$; therefore when we write $[\![\Omega]\!]^{d_0}_{d_n}$, d_0 is the same as d_n exactly when $(\Omega = \cdot)$.

Most of the complications in our setting are due to the fact that we have persistent and linear atomic propositions. This addition would break the correctness of the transformation into the linear fragment were it not for rule separation. If we do not force rule separation and have a rule like $\mathbf{1} \twoheadrightarrow Q$ which translates as $\forall d_L.\forall d_R.\,(d_L = d_R) \multimap Q(d_L, d_R)$, and the translated program can transition from a state where $\Delta = [\![Q_1 Q_2]\!]^{d_0}_{d_2} = Q_1(d_0, d_1)Q_2(d_1, d_2)$ to a state where $\Delta' = Q_1(d_0, d_1)Q(d_1, d_1)Q_2(d_1, d_2)$. This is not equal to $[\![Q_1 Q Q_2]\!]^{d_0}_{d_2}$ because there are not 4 distinct parameters for the 3 propositions.

Rule separation prevents these problems, but leaves us with a choice: how should we translate a rule like $(¡Q_1 \twoheadrightarrow ¡Q_2)$? We claim that the translated rule $[\![¡Q_1 \twoheadrightarrow ¡Q_2]\!] =$

$\forall d_L.\forall d_R.\, Q_1 \otimes (d_L = d_R) \multimap Q_2 \otimes (d_L = d_R)$ has the correct behavior, but the proof is complicated and inelegant: as a derivation is built bottom-up, the above rule would cause us to temporarily break the invariant that the n ordered propositions can be uniquely "threaded together" with $n+1$ distinct parameters, but by the time we fully apply left invertible rules to reach a state, the invariant will always be restored. The simpler option is to say that the transformation of a program $[\![\Gamma_{\mathcal{P}}]\!]$ leaves every rule in the persistent or linear fragment alone, and only replaces A with $[\![A]\!]$ if A includes some ordered atomic propositions. This has the added nice property that the interpretation of programs is idempotent: $[\![[\![\Gamma_{\mathcal{P}}]\!]]\!] = [\![\Gamma_{\mathcal{P}}]\!]$.

Now that we have described the transformation of an ordered logical specification into the linear fragment, we can discuss the proof of its correctness.

2.2.2 Correctness

In this section, we present the proof of Theorem 2 with respect to successful transitions $\mathbb{S}_1 \xrightarrow{-+} \mathbb{S}_2$. The proof with respect to halting transitions $\mathbb{S}_1 \xrightarrow{-+} \bot$ is similar, and the proof with respect to terminating states $\mathbb{S}_1 \rightarrow|$ is a direct consequence of Lemma 4.

Proof (Correctness of translation for transitions, Theorem 2a) Any transition begins by picking out a rule $A \in \Gamma_{\mathcal{P}}$. Using Lemma 1 to avoid dealing with the the program $\Gamma_{\mathcal{P}}$ everywhere, it suffices to show:

$$\begin{array}{c}\Gamma'; \Delta_l; \cdot \Rightarrow_{\Sigma_l} [\![S]\!] \\ \mathcal{D} \\ \Gamma; \Delta[\![\Omega]\!]; [[\![A]\!]] \Rightarrow_{\Sigma d_0 \ldots d_n} [\![S]\!]\end{array} \quad \text{if and only if} \quad \begin{array}{c}\Gamma'; \Delta_o; \Omega_o \Rightarrow_{\Sigma_o} S \\ \mathcal{E} \\ \Gamma; \Delta; \Omega_L[A]\Omega_R \Rightarrow_{\Sigma} S \\ \\ \Omega = \Omega_L\Omega_R \\ \Delta_l = \Delta_o[\![\Omega_o]\!]_{d_m}^{d_0} \\ \Sigma_l = \Sigma_o\Sigma_w d_0 \ldots d_m\end{array}$$

where both $\mathcal{D}$ and $\mathcal{E}$ are sequential derivations.

If A is in the persistent or linear fragment, then we know that A contains no mention of any of the predicates in $[\![\Omega]\!]_{d_n}^{d_0}$, and so by invariants of this fragment (Lemma 3), it suffices to show:

$$\begin{array}{c}\Gamma'; \Delta'[\![\Omega]\!]_{d_n}^{d_0}; \cdot \Rightarrow_{\Sigma' d_0 \ldots d_n} [\![S]\!] \\ \mathcal{D} \\ \Gamma; \Delta[\![\Omega]\!]_{d_n}^{d_0}; \gamma \Rightarrow_{\Sigma d_0 \ldots d_n} [\![S]\!]\end{array} \quad \text{if and only if} \quad \begin{array}{c}\Gamma'; \Delta'; \Omega_L\Omega_R \Rightarrow_{\Sigma'} S \\ \mathcal{E} \\ \Gamma; \Delta; \Omega_L\gamma\Omega_R \Rightarrow_{\Sigma} S\end{array} \quad \text{and} \quad \begin{array}{c}\Omega = \\ \Omega_L\Omega_R\end{array}$$

Where γ is either $[A]$, implying left focus, or Ω_1, implying left inversion. This proof (in both directions) is a straightforward induction on the structure of the derivation starting at the bottom and working up.

Now we must consider the case where the premise of A contains ordered atomic propositions (the conclusion of the rule may or may not contain ordered atomic propositions). The reverse direction (completeness) is unsurprising:

$$\text{Given} \quad \begin{array}{c}\Gamma'; \Delta'; \Omega' \Rightarrow_{\Sigma'} S \\ \mathcal{E} \\ \Gamma; \Delta; \Omega_L[A]\Omega_R \Rightarrow_{\Sigma} S\end{array} \quad \text{show} \quad \begin{array}{c}\Gamma'; \Delta'[\![\Omega']\!]_{d_m}^{d_0}; \cdot \Rightarrow_{\Sigma' \Sigma_w d_0 \ldots d_m} [\![S]\!] \\ \mathcal{D} \\ \Gamma; \Delta[\![\Omega_L\Omega_R]\!]_{d_n}^{d_0}; [[\![A]\!]] \Rightarrow_{\Sigma d_0 \ldots d_n} [\![S]\!]\end{array}.$$

In the case that $A = \forall x.A'$ we use the induction hypothesis on the subderivation where the focus is on $A'[t/x]$. In the case that $A = S_1 \twoheadrightarrow S_2$, we have a derivation of this form:

$$\dfrac{\begin{array}{cc} & \Gamma'; \Delta'; \Omega' \Rightarrow_{\Sigma'} S \\ \mathcal{E}_1 & \mathcal{E}_2 \\ \Gamma; \Delta_1; \Omega_1 \Rightarrow_{\Sigma} [S_1] & \Gamma; \Delta_2; \Omega_L S_2 \Omega_R \Rightarrow_{\Sigma} S \end{array}}{\Gamma; \Delta_1 \Delta_2; \Omega_L [S_1 \twoheadrightarrow S_2] \Omega_1 \Omega_R \Rightarrow_{\Sigma} S} \twoheadrightarrow_L$$

By $\mathcal{E}_1$ and the correctness of right focus (Lemma 4) we know there is a derivation $\mathcal{D}_1$ of $\Gamma; \Delta_1 [\![\Omega_1]\!]^{d_j}_{d_k}; \cdot \Rightarrow_{\Sigma d_j \dots d_k} [[\![S_1]\!]^{d_j}_{d_k}]$. By $\mathcal{E}_2$ and the completeness of left inversion (Lemma 6) there is a sequential derivation $\mathcal{D}_2$:

$$\begin{array}{c} \Gamma; \Delta' [\![\Omega']\!]^{d_0}_{d_m}; \cdot \Rightarrow_{\Sigma' d_0 \dots d_m} [\![S]\!] \\ \mathcal{D}_2 \\ \Gamma; \Delta_2 [\![\Omega_L]\!]^{d_0}_{d_j} [\![\Omega_R]\!]^{d_k}_{d_n}; [\![S_2]\!]^{d_j}_{d_k} \Rightarrow_{\Sigma d_0 \dots d_j d_k \dots d_n} [\![S]\!] \end{array}$$

We can weaken and rename the parameters in $\mathcal{D}_1$ and $\mathcal{D}_2$ (Lemma 2) to construct the following:

$$\dfrac{\dfrac{\dfrac{\begin{array}{cc} & \Gamma; \Delta' [\![\Omega']\!]^{d_0}_{d_m}; \cdot \Rightarrow_{\Sigma' \Sigma_w d_0 \dots d_m} [\![S]\!] \\ \mathcal{E}_S & \mathcal{E}_1 \\ \Gamma; \Delta_1 [\![\Omega_1]\!]^{d_j}_{d_k}; \cdot \Rightarrow_{\Sigma \Sigma_\Omega} [[\![S_1]\!]^{d_j}_{d_k}] & \Gamma_{\mathcal{P}} \Gamma; \Delta_2 [\![\Omega_L]\!]^{d_0}_{d_j} [\![\Omega_R]\!]^{d_k}_{d_n}; [\![S_2]\!]^{d_j}_{d_k} \Rightarrow_{\Sigma \Sigma_\Omega} [\![S]\!] \end{array}}{\Gamma; \Delta_1 \Delta_2 [\![\Omega_L \Omega_1 \Omega_R]\!]^{d_0}_{d_n}; [[\![S_1]\!]^{d_j}_{d_k} \twoheadrightarrow [\![S_2]\!]^{d_j}_{d_k}] \Rightarrow_{\Sigma \Sigma_\Omega} [\![S]\!]} \twoheadrightarrow_L}{\Gamma; \Delta_1 \Delta_2 [\![\Omega_L \Omega_1 \Omega_R]\!]^{d_0}_{d_n}; [\forall d_R. [\![S_1]\!]^{d_j}_{d_R} \twoheadrightarrow [\![S_2]\!]^{d_j}_{d_R}] \Rightarrow_{\Sigma \Sigma_\Omega} [\![S]\!]} \forall_L}{\Gamma; \Delta_1 \Delta_2 [\![\Omega_L \Omega_1 \Omega_R]\!]^{d_0}_{d_n}; [\forall d_L. \forall d_R. [\![S_1]\!]^{d_L}_{d_R} \twoheadrightarrow [\![S_2]\!]^{d_L}_{d_R}] \Rightarrow_{\Sigma \Sigma_\Omega} [\![S]\!]} \forall_L$$

where $\Sigma_\Omega = \Sigma d_0 \dots d_j \dots d_k \dots d_n$ and Σ_w consists of the parameters used in the translation of Ω_1 but not in the translation of Ω_L or Ω_R. Because $\forall d_L. \forall d_R. [\![S_1]\!]^{d_L}_{d_R} \twoheadrightarrow [\![S_2]\!]^{d_L}_{d_R} = [\![S_1 \twoheadrightarrow S_2]\!]$, this completes the backward direction of the proof.

The forward direction (soundness) is the direction that would fail were it not for rule separation:

$$\text{Given} \quad \begin{array}{c} \Gamma'; \Delta_l; \cdot \Rightarrow_{\Sigma_l} [\![S]\!] \\ \mathcal{D} \\ \Gamma; \Delta [\![\Omega]\!]^{d_0}_{d_n}; [[\![A]\!]] \Rightarrow_{\Sigma d_0 \dots d_n} [\![S]\!] \end{array} \quad \text{show} \quad \begin{array}{c} \Gamma'; \Delta_o; \Omega_o \Rightarrow_{\Sigma'} S \\ \mathcal{E} \\ \Gamma; \Delta; \Omega_L [A] \Omega_R \Rightarrow_{\Sigma} S \\ \\ \Omega = \Omega_L \Omega_R \\ \Delta_l = \Delta_o [\![\Omega_o]\!]^{d_0}_{d_m} \\ \Sigma_l = \Sigma' \Sigma_w d_0 \dots d_m \end{array}$$

In the case that $A = \forall x.A'$ we again use the induction hypothesis on the subderivation where the focus is on $A'[t/x]$. In the case that $A = S_1 \twoheadrightarrow S_2$, because $[\![S_1 \twoheadrightarrow S_2]\!] = \forall d_L. \forall d_R. [\![S_1]\!]^{d_L}_{d_R} \twoheadrightarrow [\![S_2]\!]^{d_L}_{d_R}$, we have a derivation of this form:

$$\dfrac{\dfrac{\dfrac{\mathcal{D}_1 \quad \Gamma';\Delta_l;\cdot \Rightarrow_{\Sigma_l} [\![S]\!]^{d_0}_{d_m} \quad \mathcal{D}_2}{\Gamma;\Delta_1;\cdot \Rightarrow_{\Sigma d_0 \ldots d_n} [\![S_1]\!]^{t_L}_{t_R} \quad \Gamma;\Delta_2;[\![S_2]\!]^{t_L}_{t_R} \Rightarrow_{\Sigma d_0 \ldots d_n} [\![S]\!]} \twoheadrightarrow_L}{\Gamma;\Delta_1\Delta_2;[[\![S_1]\!]^{t_L}_{t_R} \twoheadrightarrow [\![S_2]\!]^{t_L}_{t_R}] \Rightarrow_{\Sigma d_0 \ldots d_n} [\![S]\!]} \forall_L}{\Gamma;\Delta_1\Delta_2;[\forall d_R.[\![S_1]\!]^{t_L}_{d_R} \twoheadrightarrow [\![S_2]\!]^{t_L}_{d_R}] \Rightarrow_{\Sigma d_0 \ldots d_n} [\![S]\!]} \forall_L}{\Gamma;\Delta_1\Delta_2;[\forall d_L.\forall d_R.[\![S_1]\!]^{d_L}_{d_R} \twoheadrightarrow [\![S_2]\!]^{d_L}_{d_R}] \Rightarrow_{\Sigma d_0 \ldots d_n} [\![S]\!]} \forall_L$$

with the caveat that t_L and t_R are not known to be distinct and $\Delta_1\Delta_2 = \Delta[\![\Omega]\!]^{d_0}_{d_n}$. Because $\Delta_1 \subset \Delta[\![\Omega]\!]^{d_0}_{d_n}$, the correctness of right focus (Lemma 4) and $\mathcal{D}_1$ means that $\Delta_1 = \Delta'_1[\![\Omega'_1]\!]^{d_j}_{d_k}$, $t_L = d_j$, $t_R = d_k$, and there is a derivation $\mathcal{E}_1$ of $\Gamma;\Delta'_1;\Omega'_1 \Rightarrow_\Sigma S$. Because all the linear propositions that don't end up in Δ_1 must be in Δ_2, we have $\Delta_2 = \Delta'_2[\![\Omega'_L]\!]^{d_0}_{d_k}[\![\Omega'_R]\!]^{d_k}_{d_n}$ and $\Delta = \Delta'_1\Delta'_2$. If we let $\Omega_L = \Omega'_L$ and $\Omega_R = \Omega'_1\Omega'_R$, then we have $\Omega = \Omega_L\Omega_R$ as required.

By the soundness of left inversion (Lemma 7), $\mathcal{D}_2$, and strengthening of parameters (Lemma 2), we have $\Delta_l = \Delta_o[\![\Omega_o]\!]^{d_0}_{d_m}$, $\Sigma_l = \Sigma'\Sigma_w d_0 \ldots d_m$ (where Σ_w again contains all the parameters used in the translation of Ω_1 but not in the translation of Ω_L or Ω_R), and a derivation $\mathcal{E}_2$:

$$\dfrac{\Gamma';\Delta_o;\Omega_o \Rightarrow_{\Sigma'} S}{\Gamma;\Delta'_2;\Omega'_L S_2 \Omega'_R \Rightarrow_\Sigma S}\ \mathcal{E}_2$$

Using $\twoheadrightarrow_L$, $\mathcal{E}_1$, and $\mathcal{E}_2$ we obtain a sequential derivation ending in $\Gamma;\Delta;\Omega_L[S_1 \twoheadrightarrow S_2]\Omega_R \Rightarrow_\Sigma S$, which completes the forward direction. □

Lemma 1 (Weakening/strenghtening rules) *If $\Gamma_\mathcal{P}$ and $\Gamma_\mathcal{Q}$ contain only closed negative propositions (i.e. rules), and there is a sequential derivation not including* $\mathbf{focus}_R$ *from $\Gamma_\mathcal{P}\Gamma';\Delta';\Omega' \Rightarrow S$ to $\Gamma_\mathcal{P}\Gamma;\Delta;\Omega_L[A]\Omega_R \Rightarrow S$, then there is a sequential derivation (of the same size) from $\Gamma_\mathcal{Q}\Gamma';\Delta';\Omega' \Rightarrow S$ to $\Gamma_\mathcal{Q}\Gamma;\Delta;\Omega_L[A]\Omega_R \Rightarrow S$.*

Proof Straightforward induction on derivations.

Lemma 2 (Weakening/strenghtening parameters) *If Γ, Δ, and Ω contain atomic propositions that do not mention the parameters in Σ', then*

- *if there is a derivation of $\Gamma;\Delta;\Omega \Rightarrow_{\Sigma\Sigma'} [S]$, then there is a derivation (of the same size) of $\Gamma;\Delta;\Omega \Rightarrow_{\Sigma\Sigma''} [S]$, and*
- *if there is a sequential derivation consisting only of left invertible rules from $\Gamma';\Delta';\Omega' \Rightarrow_{\Sigma_t} S$ to $\Gamma;\Delta;\Omega \Rightarrow_{\Sigma\Sigma'} S$, then $\Sigma_t = \Sigma'''\Sigma'$ and there is a sequential derivation (of the same size) consisting only of left invertible rules from $\Gamma';\Delta';\Omega' \Rightarrow_{\Sigma'''\Sigma''} S$ to $\Gamma;\Delta;\Omega \Rightarrow_{\Sigma\Sigma''} S$.*

Proof Straightforward induction on derivations.

Lemma 3 (Invariants of the linear fragment) *If A is a rule in the persistent or linear fragment which contains no mention of any of the predicates in the linear context Δ_t, and $\mathcal{D}$ is a sequential derivation from the state $\Gamma_\mathcal{P}\Gamma';\Delta';\Omega' \Rightarrow S$ to $\Gamma_\mathcal{P}\Gamma;\Delta\Delta_t;\Omega_L[A]\Omega_R \Rightarrow S$, then $\Omega' = \Omega_L\Omega_R$ and $\Delta' = \Delta''\Delta_t$.*

Proof Induction over the structure of the sequential derivation; we have to generalize the induction hypothesis to allow the bottom of the derivation to either have a left focus $[A]$ or an ordered context Ω containing no ordered atomic propositions. We also need a lemma stating a similar property for right-focused derivations, that if S contains no ordered propositions and contains no mention of any of the predicates in Δ_t, then $\Gamma; \Delta; \Omega \Rightarrow [S]$ implies $(\Omega = \cdot)$ and that Δ is disjoint from Δ_t. This lemma is also proved by straightforward induction on the structure of the derivation.

Lemma 4 (Correctness of translation for right focus) *For all contexts Γ, Δ, and Ω containing only atomic propositions:*

- *If there is a derivation of $\Gamma; \Delta; \Omega \Rightarrow [S]$, then there is a derivation of $[\![\Gamma; \Delta; \Omega \Rightarrow [S]]\!]_{\mathbb{S}}$.*
- *If there is a derivation of $\Gamma; \Delta; \cdot \Rightarrow_{\Sigma d_0 \ldots d_n} [[\![S]\!]_s^t]$ and $\Delta \subset \Delta_o [\![\Omega_o]\!]_{d_n}^{d_0}$,*

  then $\Delta = \Delta' [\![\Omega']\!]_{d_k}^{d_j}$ and there is a derivation of $\Gamma; \Delta'; \Omega' \Rightarrow_\Sigma S$. Furthermore, if S contains ordered atomic propositions then $t = d_j$ and $s = d_k$, and if S contains no ordered atomic propositions then $t = s$.

Proof Induction over the structure of right-focused derivations.

Lemma 5 (Invertibility) *For any state $(\Gamma'; \Delta'; \Omega' \Rightarrow_\Sigma S_c)$ and any sequential derivation*

$$\begin{array}{c} \Gamma'; \Delta'; \Omega' \Rightarrow_{\Sigma'} S_c \\ \mathcal{E} \\ \Gamma; \Delta; \Omega_L S \Omega_R \Rightarrow_\Sigma S_c \end{array}$$

- *if $S = S_1 \bullet S_2$, then there is a smaller derivation ending in $\Gamma; \Omega_L S_1 S_2 \Omega_R \Rightarrow_\Sigma S_c$,*
- *if $S = ¡Q$, then there is a smaller derivation ending in $\Gamma; \Delta Q; \Omega_L \Omega_R \Rightarrow_\Sigma S_c$,*
- *if $S = !Q$, then there is a smaller derivation ending in $\Gamma Q; \Delta; \Omega_L \Omega_R \Rightarrow_\Sigma S_c$,*
- *if $S = \exists x.S'$, then there is a smaller derivation ending in $\Gamma; \Delta; \Omega_L S'[a/x] \Omega_R \Rightarrow_{\Sigma a} S_c$,*
- *if $S = \mathbf{1}$, then there is a smaller derivation ending in $\Gamma; \Delta; \Omega_L \Omega_R \Rightarrow_\Sigma S_c$, and*
- *if $S = (t = s)$, then t and s have a single most general unifier θ with domain Σ and range Σ'' and there is a smaller derivation ending in $\theta\Gamma; \theta\Delta; \theta\Omega_L \theta\Omega_R \Rightarrow_{\Sigma''} \theta S_c$*

Proof Induction on the length of the derivation. The "smaller" part is critical, as it justifies feeding the resulting derivation into the induction hypothesis.

Lemma 6 (Completeness of translation for left inversion) *For any contexts Γ', Δ', Ω', Γ, Δ, Ω_L, and Ω_R containing only atomic propositions,*

$$\textit{given} \quad \begin{array}{c} \Gamma'; \Delta'; \Omega' \Rightarrow_{\Sigma'} S \\ \mathcal{D} \\ \Gamma; \Delta; \Omega_L \Omega_1 \Omega_R \Rightarrow_\Sigma S \end{array} \quad \textit{there exists} \quad \begin{array}{c} [\![\Gamma'; \Delta'; \Omega' \Rightarrow_{\Sigma'} S]\!]_{\mathbb{S}} \\ \mathcal{E} \\ \Gamma; \Delta [\![\Omega_L]\!]_{d_j}^{d_0} [\![\Omega_R]\!]_{d_n}^{d_k}; [\![\Omega_1]\!]_{d_k}^{d_j} \Rightarrow_{\Sigma\Sigma_\Omega} [\![S]\!] \end{array}$$

where $\Sigma_\Omega = d_0 \ldots d_j \ldots d_k \ldots d_n$.

Proof Lexicographic induction: either the length of the derivation $\mathcal{D}$ gets smaller or stays the same while the size of the ordered context Ω_1 decreases. We proceed by case analysis on the structure of Ω_1. We give three representative cases:

If $(\Omega_1 = \cdot)$, then $\mathcal{D}$ cannot take any steps, and the result is immediate (with $\Gamma = \Gamma'$, $\Delta = \Delta'$, and $\Omega' = \Omega_L \Omega_R$.

If $(\Omega_1 = (S_1 \bullet S_2)\Omega)$, then by invertibility (Lemma 5) we have a smaller derivation ending in $\Gamma; \Delta; \Omega_L S_1 S_2 \Omega \Omega_R \Rightarrow_\Sigma S$. We can get a transformed derivation ending in:

- $\Gamma; \Delta[\![\Omega_L]\!]^{d_0}_{d_j}[\![\Omega_R]\!]^{d_k}_{d_n}; [\![S_1 S_2 \Omega]\!]^{d_j}_{d_k} \Rightarrow_{\Sigma\Sigma_\Omega} [\![S]\!]$ by the induction hypothesis
- $\Gamma; \Delta[\![\Omega_L]\!]^{d_0}_{d_j}[\![\Omega_R]\!]^{d_k}_{d_n}; [\![S_1]\!]^{d_j}_{d_{j+1}}[\![S_2]\!]^{d_{j+1}}_{d_{j+2}}[\![\Omega]\!]^{d_{j+2}}_{d_k} \Rightarrow_{\Sigma\Sigma_\Omega} [\![S]\!]$ by the definition of $[\![]\!]^{d_j}_{d_k}$
- $\Gamma; \Delta[\![\Omega_L]\!]^{d_0}_{d_j}[\![\Omega_R]\!]^{d_k}_{d_n}; ([\![S_1]\!]^{d_j}_{d_{j+1}} \bullet [\![S_2]\!]^{d_{j+1}}_{d_{j+2}})[\![\Omega]\!]^{d_{j+2}}_{d_k} \Rightarrow_{\Sigma\Sigma_\Omega} [\![S]\!]$ by $\bullet_L$
- $\Gamma; \Delta[\![\Omega_L]\!]^{d_0}_{d_j}[\![\Omega_R]\!]^{d_k}_{d_n}; (\exists d_{j+1}.\, [\![S_1]\!]^{d_j}_{d_{j+1}} \bullet [\![S_2]\!]^{d_{j+1}}_{d_{j+2}})[\![\Omega]\!]^{d_{j+2}}_{d_k} \Rightarrow_{\Sigma\Sigma_\Omega} [\![S]\!]$ by $\exists_L$
- $\Gamma; \Delta[\![\Omega_L]\!]^{d_0}_{d_j}[\![\Omega_R]\!]^{d_k}_{d_n}; [\![(S_1 \bullet S_2)\Omega]\!]^{d_j}_{d_k} \Rightarrow_{\Sigma\Sigma_\Omega} [\![S]\!]$ by the definition of $[\![]\!]^{d_j}_{d_k}$

This completes the case.

If $(\Omega_1 = Q\Omega)$, then we use the induction hypothesis on the same derivation where $(\Omega'_L = \Omega_L Q)$ and $(\Omega'_1 = \Omega)$ – this is the case where we use the lexicographic induction, as the ordered context gets smaller but the size of the derivation stays the same. We get a transformed derivation ending in:

- $\Gamma; \Delta[\![\Omega_L Q]\!]^{d_0}_{d_j}[\![\Omega_R]\!]^{d_k}_{d_n}; [\![\Omega]\!]^{d_j}_{d_k} \Rightarrow_{\Sigma\Sigma_\Omega} [\![S]\!]$ by the induction hypothesis
- $\Gamma; \Delta[\![\Omega_L]\!]^{d_0}_{d_{j-1}} Q(d_{j-1}, d_j)[\![\Omega_R]\!]^{d_k}_{d_n}; [\![\Omega]\!]^{d_j}_{d_k} \Rightarrow_{\Sigma\Sigma_\Omega} [\![S]\!]$ by the definition of $[\![]\!]^{d_j}_{d_k}$
- $\Gamma; \Delta[\![\Omega_L]\!]^{d_0}_{d_{j-1}}[\![\Omega_R]\!]^{d_k}_{d_n}; (¡Q(d_{j-1}, d_j))[\![\Omega]\!]^{d_j}_{d_k} \Rightarrow_{\Sigma\Sigma_\Omega} [\![S]\!]$ by ${¡}_L$
- $\Gamma; \Delta[\![\Omega_L]\!]^{d_0}_{d_{j-1}}[\![\Omega_R]\!]^{d_k}_{d_n}; [\![Q\Omega]\!]^{d_{j-1}}_{d_k} \Rightarrow_{\Sigma\Sigma_\Omega} [\![S]\!]$ by the definition of $[\![]\!]^{d_j}_{d_k}$

This completes the case.

Lemma 7 (Soundness of translation for left inversion) *For any contexts Γ', Δ', Ω', Γ, Δ, Ω_L, and Ω_R containing only atomic propositions,*

$$\begin{array}{ccc} \Gamma'; \Delta'; \cdot \Rightarrow_{\Sigma'} [\![S]\!]^{d_0}_{d_m} & & \Gamma'; \Delta''; \Omega'' \Rightarrow_{\Sigma''} S \\ \mathcal{D} & & \mathcal{E} \\ \textit{given} \quad \Gamma; \Delta[\![\Omega_L]\!]^{d_0}_{d_j}[\![\Omega_R]\!]^{d_k}_{d_n}; [\![\Omega]\!]^{d_j}_{d_k} \Rightarrow_{\Sigma\Sigma_\Omega} [\![S]\!] & \textit{there exists} & \Gamma; \Delta; \Omega_L \Omega \Omega_R \Rightarrow_{\Sigma\Sigma_\Omega} S \\ \Sigma_\Omega = d_0 \ldots d_j \ldots d_k \ldots d_n & & \Delta' = \Delta''[\![\Omega'']\!]^{d_0}_{d_m} \\ & & \Sigma' = \Sigma'' d_0 \ldots d_m \end{array}$$

Proof Induction on the length of the derivation $\mathcal{D}$, repeatedly invoking invertibility (Lemma 5) as in the proof of Lemma 6.

2.3 Linear destination-passing style

To conclude this section, we will make an observation about the result of passing ordered SSOS specifications to through the transformation we have described. If we transform the ordered SSOS specification of the call-by-value lambda calculus from the introduction (rules 1-4) into a linear logical specification, we get the following:

$$\mathsf{eval}\,(\mathsf{lam}(\lambda x.E\,x))\,D\,D' \multimap \mathsf{retn}\,(\mathsf{lam}(\lambda x.E\,x))\,D\,D'$$

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

$$(\exists d_1.\,\mathsf{comp}\,(\mathsf{app}_1\,E_2)\,D\,d_1 \otimes \mathsf{retn}\,V_1\,d_1\,D') \multimap \exists d_2.\,\mathsf{comp}\,(\mathsf{app}_2\,V_1)\,D\,d_2 \otimes \mathsf{eval}\,E_2\,d_2\,D'$$

$$(\exists d_2.\,\mathsf{comp}\,(\mathsf{app}_2(\mathsf{lam}(\lambda x.E_0\,x)))\,D\,d_2 \otimes \mathsf{retn}\,V_2\,d_2\,D') \multimap \mathsf{eval}\,(E_0\,V_2)\,D\,D'$$

Note that the third argument of $\mathsf{eval}/\mathsf{retn}$ is essentially meaningless – it is always passed on intact from the premise the conclusion. This simply reflects the fact that our control stack grows out to the left, and we are never concerned with what is to the right of an eval or retn atomic proposition. Additionally, a rule that binds an existential parameter in its premise is equivalent to one that binds the same parameter universally. By removing the vestigial D' and universally quantifying where appropriate, we can rewrite this program into an essentially equivalent one:

$$\mathsf{eval}\ (\mathsf{lam}(\lambda x.E\,x))\ D \multimap \mathsf{retn}\ (\mathsf{lam}(\lambda x.E\,x))\ D \tag{7}$$

$$\mathsf{eval}\ (\mathsf{app}\ E_1\ E_2)\ D \multimap \exists d_1.\ \mathsf{comp}\ (\mathsf{app}_1\ E_2)\ D\ d_1 \otimes \mathsf{eval}\ E_1\ d_1 \tag{8}$$

$$\mathsf{comp}\ (\mathsf{app}_1\ E_2)\ D\ D_1 \otimes \mathsf{retn}\ V_1\ D_1 \multimap \exists d_2.\ \mathsf{comp}\ (\mathsf{app}_2\ V_1)\ D\ d_2 \otimes \mathsf{eval}\ E_2\ d_2 \tag{9}$$

$$\mathsf{comp}\ (\mathsf{app}_2(\mathsf{lam}(\lambda x.E_0\,x)))\ D\ D_2 \otimes \mathsf{retn}\ V_2\ D_2 \multimap \mathsf{eval}\ (E\,V_2)\ D \tag{10}$$

This program is significant because it is an example of a linear SSOS specification using *linear destination-passing style* – the parameters introduced by the translation are called *destinations* because we think of D in $\mathsf{eval}\,E\,D$ as the eventual destination of the result of evaluating E. Linear destination-passing style was the original form of substructural operational semantics specifications before ordered logic was considered as a framework [23, 9].

The fact that linear destination-passing style arises naturally from the transformation of an ordered SSOS specification into linear logic is a new observation, and is interesting in its own right.[4] For the purposes of our current discussion, the transformation to destination-passing style is important primarily because the destinations make control flow information explicit. As we will see, this explicit representation of control flow is what will make it possible to derive program approximations that are sensitive to control flow.

3 Saturating forward-chaining logic programming

We have presented the language of ordered logical specifications, which we think of as a state transition system. We usually think of implementing these specification languages by a forward-chaining logic programming language with a *committed choice* semantics. This means that if the state can evolve in two different but possibly mutually exclusive ways, we arbitrarily pick one transition and do not reconsider that choice, and once a state is reached from which no transitions are possible, evaluation halts (we say it has reached *quiescence*) [16, 32, 26]. This makes sense for ordered and linear logical specifications, but when we consider persistent specifications the story is different. The use of a persistent proposition in a premise does not remove that persistent proposition

[4] One way to interpret this formal relationship between ordered SSOS specifications and linear SSOS specifications using destination-passing style is to think of the *linear* specifications as primary and ordered SSOS specifications as a convenient syntax for them. However, one of the goals of SSOS specification is to classify programming language features by the substructural properties needed to encode them: ordered logic is in some sense the most restrictive variant, naturally providing specifications of features like ambient state and parallelism but not features like first-class continuations for which destination-passing appears to be critical [26]. An intriguing direction for future work is to see whether this formal connection can be used to modularly combine ordered SSOS specifications with linear SSOS specifications of features (such as first-class continuations) that seem to only be amenable to SSOS specifications in destination-passing style.

from the context. This means that if we make a given transition once, we can make it again (deriving a new copy of all the facts in the conclusion), and it also means that those new facts will not matter, because one copy of a fact is as good as two.

A state $\mathbb{S}$ in which any rule application only re-derives known facts is said to be *saturated*. A *saturating* semantics for logic programs allows only transitions that derive previously unknown facts. Saturating logic programs need not always terminate, but if a program reaches a saturated state it will terminate. A program with existential parameters in the conclusion will usually only terminate if the parameters are unified with ground terms through the use of equality – otherwise we could productively apply the rule to create a new parameter along with a new fact containing that parameter. Effective ways of implementing saturating logic programs are dealt with elsewhere (see, for example, [17]), and we will not discuss them here.[5]

In this section, we describe an approximation methodology in which we can approximate logical specifications as persistent logical specifications. If those persistent logical specifications can also be run as terminating saturating logic programs, we obtain a way of approximating the behavior of logically specified systems.

3.1 Approximating ordered logic programs as saturating logic programs

Our approximation strategy is extremely simple: essentially, an approximation of an ordered or linear logical specification is generated by making all atomic propositions persistent, removing premises, and adding conclusions. Of particular practical importance are added conclusions that equate parameters introduced by existential quantification and ground terms: all such parameters must be equated with ground terms in order to interpret a persistent specification as a saturating logic program.

First, we give a precise definition of what it means for a program to be an approximate version of another program.

Definition 1 A program Γ_a is an *approximate version* of another program $\Gamma_\mathcal{P}$ if Γ_a is in the persistent fragment, and if, for each rule in Γ_a:

- the existential parameters are identical to the existential parameters of the corresponding rule in $\Gamma_\mathcal{P}$
- the premises are a subset of the premises of the corresponding rule in $\Gamma_\mathcal{P}$, and
- the conclusions are a superset of the conclusions of the corresponding rule in $\Gamma_\mathcal{P}$.

Next we give a definition of what it means for a state to be an approximate version (we use the word "generalization") of another state or of a family of states. For the purposes of defining generalizations of a state ($\Gamma_\mathcal{P}\Gamma; \Delta; \Omega \Rightarrow_\Sigma S$) we are uninterested in the program $\Gamma_\mathcal{P}$ or the conclusion S. We use a different notation describing states as tuples $\langle \Sigma, \Gamma, \Delta, \Omega \rangle$ to emphasize this. A series of states $\mathbb{S}_1 \xrightarrow{-+} \ldots \xrightarrow{-+} \mathbb{S}_n$ is called a *trace*.

Definition 2 A state $\langle \Sigma_g, \Gamma_g, \cdot, \cdot \rangle$ is a *generalization* of a state $\langle \Sigma, \Gamma, \Delta, \Omega \rangle$ if there is a substitution function $\Sigma_g \vdash \theta : \Sigma$ such that, for all propositions $A \in \Gamma, \Delta, \Omega$, there exists a proposition $A_g \in \Gamma_g$ such that $\theta A = A_g$.

[5] One interesting aspect of saturation is the interaction of saturation and equality. Is a state containing only the persistent atomic proposition $Q(x, y)$ saturated if there is a rule $Q(x, y) \supset x = y$? We take the position that it is, though this slightly complicates the meta-approximation theorem in the next section.

Definition 3 A state $\mathbb{S}_a$ is an *abstraction* of a program $\Gamma_{\mathcal{P}}$ with an initial state $\mathbb{S}_0$ if, for any trace $\mathbb{S}_0 \xrightarrow{-+} \ldots \xrightarrow{-+} \mathbb{S}'$, $\mathbb{S}_a$ is a generalization of $\mathbb{S}'$.

The meta-approximation theorem relates the definition of *abstraction* above to the concept of an *approximate version* of a program as specified by Definition 1.

Theorem 3 (Meta-approximation) *If Γ_a is an approximate version of $\Gamma_{\mathcal{P}}$, $\mathbb{S}_0 = \langle \Sigma_0, \Gamma_0, \Delta_0, \Omega_0 \rangle$ is an initial state of $\Gamma_{\mathcal{P}}$, and if $\langle \Sigma'_0, \theta(\Gamma_0 \Delta_0 \Omega_0), \cdot, \cdot \rangle \xrightarrow{-+} \ldots \xrightarrow{-+} \mathbb{S}_a$ for some $\Sigma'_0 \vdash \theta : \Sigma_0$ under the approximate program Γ_a where $\mathbb{S}_a$ is saturated, and if there is no trace of the approximate version that aborts, then $\mathbb{S}_a$ is an abstraction of $\Gamma_{\mathcal{P}}$ with initial state $\mathbb{S}_0$.*

Proof Consider a trace $\langle \Sigma_0, \Gamma_0, \Delta_0, \Omega_0 \rangle \xrightarrow{-+} \ldots \xrightarrow{-+} \mathbb{S}_n$ of the original program. By Lemma 9 (Simulation) there is a trace $\langle \Sigma_0, \theta(\Gamma_0 \Delta_0 \Omega_0), \cdot, \cdot \rangle \xrightarrow{-+} \ldots \xrightarrow{-+} \mathbb{S}'_n$ of the approximate program such that $\mathbb{S}'_n$ generalizes $\mathbb{S}_n$ – according to Lemma 9 this trace could also abort, but we assume in the statement of the theorem that this will not happen. Then, by Lemma 11 (Saturation), we know that the saturated state $\mathbb{S}_a$ is a generalization of $\mathbb{S}'_n$. Because generalization is transitive, $\mathbb{S}_a$ is a generalization of $\mathbb{S}_n$, which is what we needed to show.

The meta-approximation theorem relies on four lemmas that are described below. The first two lemmas establish that an approximate version of a program can simulate the program it approximates, and next two formalize the notion that, in an approximate version of a program, the saturated database at the conclusion of a complete program trace captures all of the "behaviors" of that approximate program. These two facts in combination mean that all of the behaviors of a program are captured by the saturated database at the conclusion of a complete trace of its approximate version.

Lemma 8 (One-step simulation) *If $\mathbb{S} \xrightarrow{-+} \mathbb{S}^+$ by focusing on the rule $A \in \Gamma_{\mathcal{P}}$, and if Γ_a is an approximate version of $\Gamma_{\mathcal{P}}$ and $\mathbb{S}_g$ is a generalization of $\mathbb{S}$, then by focusing on a rule $A_a \in \Gamma_a$ either $\mathbb{S}_g \xrightarrow{-+} \mathbb{S}_g^+$ such that $\mathbb{S}_g^+$ is a generalization of $\mathbb{S}^+$, or else $\mathbb{S}_g \xrightarrow{-+} \bot$.*

Proof We consider the sequential derivation representing the transition in the focused sequent calculus. We will show that if there is a sequential derivation from $\Gamma'; \Delta'; \Omega' \Rightarrow_{\Sigma'} S$ to $\Gamma; \Delta; \Omega_L[\sigma A]\Omega_R \Rightarrow_{\Sigma} S$ and $\langle \Sigma_g, \Gamma_g, \cdot, \cdot \rangle$ is a generalization of $\langle \Sigma, \Gamma, \Delta, \Omega_L \Omega_R \rangle$ (associated with the substitution $\Sigma_g \vdash \theta : \Sigma$), then there is a sequential derivation from $\Gamma'_g; \cdot; \cdot \Rightarrow_{\Sigma'_g} S$ to $\Gamma_g; \cdot; [\theta(\sigma A_a)] \Rightarrow_{\Sigma_g} S$ such that $\langle \Sigma'_g, \Gamma'_g, \cdot, \cdot \rangle$ is a generalization of $\langle \Sigma', \Gamma', \Delta', \Omega' \rangle$. The substitution σ just tracks what terms have been substituted for the universally quantified variables in A.

As usual, the case where the last rule is $\forall_L$ is a fairly straightforward application of the induction hypothesis, and the critical case is when the last rule is $\twoheadrightarrow_L$. In that case, we have the following:

$$\dfrac{\begin{array}{cc} & \Gamma'; \Delta'; \Omega' \Rightarrow_{\Sigma'} S \\ \mathcal{D}_1 & \mathcal{D}_2 \\ \Gamma; \Delta_1; \Omega_1 \Rightarrow_{\Sigma} [\sigma S_1] & \Gamma; \Delta_2; \Omega_L(\sigma S_2)\Omega_R \Rightarrow_{\Sigma} S \end{array}}{\Gamma; \Delta_1 \Delta_2; \Omega_L[\sigma S_1 \twoheadrightarrow \sigma S_2]\Omega_1 \Omega_R \Rightarrow_{\Sigma} S} \twoheadrightarrow_L$$

The approximate version of $S_1 \twoheadrightarrow S_2$ is $S'_1 \twoheadrightarrow S'_2$, and we have $\mathcal{E}_1 :: \Gamma_g; \cdot; \cdot \Rightarrow_{\Sigma_g} \theta(\sigma S_1)$ by induction over the structure of S'_1. Because all the components of S'_1 also appear

in S_1, when we need to establish that $\Gamma_g; \cdot; \cdot \Rightarrow_{\Sigma_g} [!(\theta(\sigma Q))]$, we know that $\mathcal{D}_1$ must contain a derivation of $!(\sigma Q)$, $¡(\sigma Q)$, or (σQ). This means that (σQ) is somewhere in the original state, and therefore $\theta(\sigma Q) \in \Gamma_g$, which completes the case. The cases where S_1' is not an atomic proposition are similar.

What remains to be shown is that there is a sequential derivation $\mathcal{E}_2$ from $\Gamma_g'; \cdot; \cdot \Rightarrow_{\Sigma_g'} S$ to $\Gamma_g; \cdot; \theta(\sigma S_2') \Rightarrow_{\Sigma_g} S$ where $\langle \Sigma_g', \Gamma_g', \cdot, \cdot \rangle$ is a generalization of $\langle \Sigma', \Gamma', \Delta', \Omega' \rangle$ – from this we can conclude with $\twoheadrightarrow_L$. We generalize the conclusions S_2 and S_2' to contexts Ω_2 and Ω_2' and prove that, given a sequential derivation $\mathcal{D}_2$ from $\Gamma'; \Delta'; \Omega' \Rightarrow_{\Sigma'} S$ to $\Gamma; \Delta_2; \Omega_L(\sigma\Omega_2)\Omega_R \Rightarrow_{\Sigma} S$ where $\langle \Sigma_g, \Gamma_g, \cdot, \cdot \rangle$ is a generalization of $\langle \Sigma, \Gamma, \Delta, \Omega \rangle$, there is a sequential derivation $\mathcal{E}_2$ from $\Gamma_g'; \cdot; \cdot \Rightarrow_{\Sigma_g'} S$ to $\Gamma_g; \cdot; \theta(\sigma\Omega_2') \Rightarrow_{\Sigma_g} S$ where $\langle \Sigma_g', \Gamma_g', \cdot, \cdot \rangle$ is a generalization of $\langle \Sigma', \Gamma', \Delta', \Omega' \rangle$. The proof proceeds by lexicographic induction over first the structure of $\mathcal{D}_2$ and second the structure of Ω_2', much as in Lemma 6.

The most interesting case is where Ω_2' is $(t = s)\Omega'$. It may be the case that $\theta(\sigma t)$ and $\theta(\sigma s)$ are not unifiable, in which case the generalization aborts (as the lemma allows). Assume, on the other hand, that $\theta(\sigma t)$ and $\theta(\sigma s)$ have a most general unifier τ, where $(\Sigma_g' \vdash \tau : \Sigma_g)$. If the equality $t = s$ appears only in the approximate version, then because $\langle \Sigma_g, \Gamma_g, \cdot, \cdot \rangle$ is a generalization of $\langle \Sigma, \Gamma, \Delta, \Omega \rangle$, $\langle \Sigma_g', \tau\Gamma_g, \cdot, \cdot \rangle$ is also generalization of $\langle \Sigma, \Gamma, \Delta, \Omega \rangle$, so we can apply the induction hypothesis ($\mathcal{D}_2$ is the same size and Ω_2' is smaller) to get $\mathcal{E}_2$ ending in $(\tau\Gamma_g; \cdot; \tau(\theta(\sigma\Omega')) \Rightarrow_{\Sigma_g'} \tau S)$, and we conclude by applying $=_L$. If, on the other hand, the equality $t = s$ appears in both the original and approximate versions, then we have a second most general unifier $(\Sigma'' \vdash \tau' : \Sigma)$ of σt and σs by inversion on $\mathcal{D}_2$. To apply the induction hypothesis we must show that $\langle \Sigma_g', \tau\Gamma_g, \cdot, \cdot \rangle$ is a generalization of $\langle \Sigma'', \tau'\Gamma, \tau'\Delta, \tau'\Omega \rangle$, which amounts to showing that if $A \in (\Gamma \cup \Delta \cup \Omega)$, then $\theta(\tau' A) \in \tau\Gamma_g$. We have that $\theta A \in \Gamma_g$ by the generalization we do know about, so certainly $\tau(\theta A) \in \tau\Gamma_g$. Because τ' is the mgu of two terms and τ is the mgu of θ applied to those terms, $\theta\tau' = \tau\theta$ so $\theta(\tau' A) = \tau(\theta A)$ and the generalization holds. Therefore, we can apply the induction hypothesis ($\mathcal{D}_2$ is smaller) and conclude as before.

Lemma 9 (Simulation) *If $\langle \Sigma_0, \Gamma_0, \Delta_0, \Omega_0 \rangle \xrightarrow{-+} \ldots \xrightarrow{-+} \mathbb{S}_n$ is a trace of a program $\Gamma_\mathcal{P}$, if $\Sigma_0' \vdash \theta : \Sigma$, and if Γ_a is an approximate version of $\Gamma_\mathcal{P}$, then there exists a trace of the program Γ_a starting from $\langle \Sigma_0', \theta(\Gamma\Delta\Omega), \cdot, \cdot \rangle$ that either aborts or reaches a state $\mathbb{S}_n'$ that is a generalization of $\mathbb{S}_n$.*

Proof By induction on the length of the program trace. The base case is immediate and the inductive case follows from Lemma 8.

Lemma 10 (Monotonicity) *If a program $\Gamma_\mathcal{P}$ and a state $\mathbb{S}$ contain no linear or ordered predicates, then if $\mathbb{S} \xrightarrow{-+} \ldots \xrightarrow{-+} \mathbb{S}'$, then $\mathbb{S}'$ is a generalization of $\mathbb{S}$.*

Proof By induction on the length of the program trace. Because every step only adds new facts and applies substitutions to existing facts, this amounts to the composition of those substitutions.

Lemma 11 (Saturation) *If the program $\Gamma_\mathcal{P}$ contains no linear or ordered predicates, $\mathbb{S}_0 \xrightarrow{-+} \ldots \xrightarrow{-+} \mathbb{S}_a$ where $\mathbb{S}_a$ is saturated, and $\mathbb{S}_0 \xrightarrow{-+} \ldots \xrightarrow{-+} \mathbb{S}_n$, then $\mathbb{S}_a$ is a generalization of $\mathbb{S}_n$.*

Proof By induction on the general trace $\mathbb{S}_0, \ldots, \mathbb{S}_n$. The base case follows from Lemma 10 – $\mathbb{S}_a$ generalizes $\mathbb{S}_0$ because the latter evolved from the former.

In the inductive case, $\mathbb{S}_a$ is a generalization of $\mathbb{S}_n$, and $\mathbb{S}_n$ that evolves to $\mathbb{S}_{n+1}$. We need to show $\mathbb{S}_a$ is a generalization of $\mathbb{S}_{n+1}$. By Lemma 8, $\mathbb{S}_a$ evolves to $\mathbb{S}_a^+$, which is a generalization of $\mathbb{S}_{n+1}$. Because generalization is transitive, it suffices to show that $\mathbb{S}_a$ is a generalization of $\mathbb{S}_a^+$. But $\mathbb{S}_a$ is saturated, so every parameter or proposition in $\mathbb{S}_a^+$ is equal to a parameter or proposition in $\mathbb{S}_a$.

Lemmas 9 and 11 give us the pieces needed to complete the proof of the meta-approximation theorem.

3.2 Termination and Skolemization

A consequence of the meta-approximation theorem is that we are interested in the termination of saturating logic programs, because a terminating approximate version of a program gives us a way to generate a single state $\mathbb{S}_a$ that captures all possible behaviors of the original program. Important classes of programs are known to terminate in all cases, such as those in the so-called "Datalog fragment" where the only terms in the program are variables and constants. The approximations we consider do not fall into these fragments, but general reasoning about the termination of saturating logic programs is not difficult. Because each transition in a saturating logic program must derive at least one new fact, any logic program that can derive only finitely many facts from any (finite) initial state will necessarily terminate.

A common design pattern in saturating logic programs is to enumerate the subterms of a term and then perform a computation on those subterms; because a term has finitely many subterms, we can be sure such programs terminate. If we wanted to enumerate the subterms of arithmetical expressions that take the form

$$E ::= \mathsf{num}\ N \mid \mathsf{plus}\ E_1\ E_2 \mid \mathsf{times}\ E_1\ E_2$$

we could use the following rules:

$$\mathsf{subterms}(\mathsf{plus}\ E_1\ E_2) \supset \mathsf{subterms}(E_1) \wedge \mathsf{subterms}(E_2) \tag{11}$$

$$\mathsf{subterms}(\mathsf{times}\ E_1\ E_2) \supset \mathsf{subterms}(E_1) \wedge \mathsf{subterms}(E_2) \tag{12}$$

But what if we want to add to the language of expressions functions and function application?

$$E ::= \ldots \mid \mathsf{app}\ E_1\ E_2 \mid \mathsf{lam}(\lambda x.E\,x) \mid x$$

The rule for app is obvious, but the immediate subterm of $\mathsf{lam}(\lambda x.E\,x)$ is $(\lambda x.E\,x)$, which is a higher-order term – in order to get a subterm that is a term of base type, we must substitute something for the bound variable x. One approach would be to generate a new existential parameter y to substitute for x (as usual, we perform substitution by application).

$$\mathsf{subterms}(\mathsf{app}\ E_1\ E_2) \supset \mathsf{subterms}(E_1) \wedge \mathsf{subterms}(E_2) \tag{13}$$

$$\mathsf{subterms}(\mathsf{lam}(\lambda x.E\,x)) \supset \exists y.\ \mathsf{subterms}(E\,y) \tag{14}$$

This is a non-terminating logic program: if the rule for lam can be applied at all it can be applied repeatedly, generating a new name each time. However, the quantifier dependence between E (which is implicitly universally quantified) and y in this rule suggests Skolemization: instead of generating a new name every time a rule is applied, we replace y by a Skolem function var which depends on E.

$$\mathsf{subterms}(\mathsf{lam}(\lambda x.E\,x)) \supset \mathsf{subterms}(E(\mathsf{var}(\lambda x.E\,x))) \tag{15}$$

In this representation, a variable is effectively a *pointer* back to its binding site; as long as we never follow those pointers by matching against var in the premise of a rule, then we can bound the number of subterms we will consider. This ensures that we will only consider a finite number of facts of the form $\mathsf{subterms}(E)$, which in turn ensures termination.

3.3 Example

To conclude this section, we give a simple example of how Skolemization can be integrated with our approximation strategy. Consider a linear logical specification where a linear proposition $\mathsf{at}(x)$ represents a piece of stateful information (that we are at position x) and a persistent proposition $!\mathsf{next}(x, y)$ represents a piece of persistent information (we were at position x immediately before being at position y). We can represent this system with the following rule:

$$\mathsf{at}(X) \multimap \exists y.\, !\mathsf{next}(X, y) \otimes \mathsf{at}(y) \tag{16}$$

We will consider the process of successively approximating this rule to get a terminating saturating logic program. The most obvious approximate version of rule 16 simply turns all propositions persistent:

$$\mathsf{at}(X) \supset \exists y.\, \mathsf{next}(X, y) \wedge \mathsf{at}(y) \tag{17}$$

While rule 17 is a persistent logical specification, it is not a saturating logic program because it of the existentially quantified variable y in the conclusion. As discussed in the previous section, we can perform Skolemization by introducing a Skolem function s which depends on x.

$$\mathsf{at}(X) \supset \mathsf{next}(X, \mathsf{s}(X)) \wedge \mathsf{at}(\mathsf{s}(X)) \tag{18}$$

One problem here is that rule 18 doesn't quite fit our criteria for being an approximate version of rule 16. An equivalent rule, one where the existential parameter is left in place but then equated with $\mathsf{s}(X)$, is an approximate version of rule 16.

$$\mathsf{at}(X) \supset \exists y.\, \mathsf{next}(X, y) \wedge \mathsf{at}(y) \wedge (y = \mathsf{s}(X)) \tag{19}$$

A different problem still remains. This approximate version of rule 16 is not saturating – for any state, we can always generate a larger one. To deal with this problem, we observe that the Skolem function need not depend on all universally quantified variables. In this particular case, instead of the Skolem function $\mathsf{s}(n)$ (for "successor"), we can think about the Skolem function p (for "positive") and define the following rule:

$$\mathsf{at}(X) \supset \exists y.\, y = \mathsf{p} \wedge \mathsf{next}(X, y) \wedge \mathsf{at}(y) \tag{20}$$

Rule 20 is both an approximate version of rule 16 according to our criteria as well as a saturating logic program that will terminate: starting from the state where $\mathsf{at}(0)$ is true, we will learn $\mathsf{at}(\mathsf{p})$, $\mathsf{next}(0, \mathsf{p})$, and $\mathsf{next}(\mathsf{p}, \mathsf{p})$ – if we think of p as a stand-in for the positive-numbers, this represents that the successor of 0 is a positive number and that the successor of a positive number is a positive number. Because we can effectively Skolemize existential variables with the addition of conclusions relating to equality, the use of restricted Skolem functions with equality is our standard means of managing existential quantification and finding an approximation which is manageable as a terminating logic program.

Therefore, when we use Skolemization with higher-order functions as described above, it will look less like rule 15 and more like rule 21:

$$\mathsf{subterms}(\mathsf{lam}(\lambda x.E\,x)) \supset \exists y.\ \mathsf{subterms}(E\,y) \wedge (y = \mathsf{var}(\lambda x.E\,x)) \tag{21}$$

In the rest of the paper, our default position when encountering a parameter substituted into a higher-order function $(\lambda x.E\,x)$ will be to equate that parameter with $\mathsf{var}(\lambda x.E\,x)$, even if the rule has other universally quantified variables.

4 Approximating SSOS specifications for program analysis

So far, we have discussed four pieces: ordered logical specifications and substructural operational semantics (Section 2), the transformation of ordered logical specifications to linear logical specifications (Section 2.2), the approximation of logical specifications by persistent logical specifications (Section 3.1), and the use of Skolemization in particular to ensure that persistent logical specifications can be interpreted as saturating logic programs which always terminate (Section 3.2). In Section 2.3 we connected the former two pieces by showing how applying the transformation to ordered SSOS specifications gave rise to linear SSOS specifications exhibiting destination-passing style, and in Section 3.3 we showed how the latter two pieces fit together by deriving a terminating approximation from a linear logical specification. In this section, we will present two examples that connect all four pieces. First, in Section 4.1, we will take a variant of the call-by-value operational semantics that we considered in the introduction and Section 2.3 and derive a context-insensitive control flow analysis. Second, in Section 4.2, we will take an SSOS specification of a monadic functional language with Lisp-like mutable cons cells and derive an interprocedural alias analysis.

4.1 Control flow analysis

Figure 4 contains three steps in our approximation of a call-by-value program semantics to a context-insensitive control flow analysis. The starting point, the ordered SSOS specification, differs from the previous presentation (rules 1-4) in two ways. The first difference, the addition of a new stack frame call which is introduced when a function is applied in rule 26 and eliminated in rule 27. It will allow a more precise approximation by identifying call sites. The second difference is more interesting: the use of what was previously called an "environment semantics" for SSOS specifications [26]. In rule 26, instead of substituting the argument of a function for the function's bound variable as we did in rule 4, we generate a new parameter y, substitute *that* for the bound variable, and then generate a persistent fact $!\mathsf{bind}\,y\,V_2$ that permanently associates the

Call-by-value SSOS specification with destinations for binding

$$\mathsf{eval}(X) \bullet \mathsf{!bind}\, X\, V \twoheadrightarrow \mathsf{retn}(V) \tag{22}$$
$$\mathsf{eval}(\mathsf{lam}(\lambda x.E\, x)) \twoheadrightarrow \mathsf{retn}(\mathsf{lam}(\lambda x.E\, x)) \tag{23}$$
$$\mathsf{eval}(\mathsf{app}\ E_1\ E_2) \twoheadrightarrow \mathsf{comp}(\mathsf{app}_1\ E_2) \bullet \mathsf{eval}(E_1) \tag{24}$$
$$\mathsf{comp}(\mathsf{app}_1\ E_2) \bullet \mathsf{retn}(V_1) \twoheadrightarrow \mathsf{comp}(\mathsf{app}_2\ V_1) \bullet \mathsf{eval}(E_2) \tag{25}$$
$$\mathsf{comp}(\mathsf{app}_2(\mathsf{lam}(\lambda x.E_0\, x)) \bullet \mathsf{retn}(V_2) \twoheadrightarrow \exists y.\ \mathsf{comp}(\mathsf{call}) \bullet \mathsf{eval}(E_0\, y) \bullet \mathsf{!bind}\, y\, V_2 \tag{26}$$
$$\mathsf{comp}(\mathsf{call}) \bullet \mathsf{retn}(V_0) \twoheadrightarrow \mathsf{retn}(V_0) \tag{27}$$

Translation to the linear fragment

$$\mathsf{eval}\ X\ D\ D_1 \otimes \mathsf{!bind}\, X\, V \otimes (D_1 = D') \multimap \mathsf{retn}\ V\ D\ D' \tag{28}$$
$$\mathsf{eval}\ (\mathsf{lam}(\lambda x.E\, x))\ D\ D' \multimap \mathsf{retn}\ (\mathsf{lam}(\lambda x.E\, x))\ D\ D' \tag{29}$$
$$\mathsf{eval}\ (\mathsf{app}\ E_1\ E_2)\ D\ D' \multimap \exists d_1.\ \mathsf{comp}\ (\mathsf{app}_1\ E_2)\ D\ d_1 \otimes \mathsf{eval}\ E_1\ d_1\ D' \tag{30}$$
$$\mathsf{comp}\ (\mathsf{app}_1\ E_2)\ D\ D_1 \otimes \mathsf{retn}\ V_1\ D_1\ D' \multimap \exists d_2.\ \mathsf{comp}\ (\mathsf{app}_2\ V_1)\ D\ d_2 \otimes \mathsf{eval}\ E_2\ d_2\ D' \tag{31}$$
$$\mathsf{comp}\ (\mathsf{app}_2(\mathsf{lam}(\lambda x.E_0\, x))\ D\ D_2 \otimes \mathsf{retn}\ V_2\ D_2\ D' \tag{32}$$
$$\multimap \exists y.\ \exists d_0.\ \mathsf{comp}\ \mathsf{call}\ D\ d_0 \otimes \exists d'.\ \mathsf{eval}\ (E_0\, y)\ d_0\ d' \otimes \mathsf{!bind}\, y\, V_2 \otimes (d' = D')$$
$$\mathsf{comp}\ \mathsf{call}\ D\ D_1 \otimes \mathsf{retn}\ V_0\ D_1\ D' \multimap \mathsf{retn}\ V_0\ D\ D' \tag{33}$$

Approximation to context-insensitive control flow analysis

$$\mathsf{eval}\ X\ D \wedge \mathsf{bind}\, X\, V\ \supset\ \mathsf{retn}\, V\, D \tag{34}$$
$$\mathsf{eval}\ (\mathsf{lam}(\lambda x.E\, x))\ D\ \supset\ \mathsf{retn}\ (\mathsf{lam}(\lambda x.E\, x))\ D \tag{35}$$
$$\mathsf{eval}\ (\mathsf{app}\ E_1\ E_2)\ D\ \supset\ \exists d_1.\ \mathsf{comp}\ (\mathsf{app}_1\ E_2)\ D\ d_1 \wedge \mathsf{eval}\ E_1\ d_1 \wedge (d_1 = E_1) \tag{36}$$
$$\mathsf{comp}\ (\mathsf{app}_1\ E_2)\ D\ D_1 \wedge \mathsf{retn}\ V_1\ D_1 \tag{37}$$
$$\supset\ \exists d_2.\ \mathsf{comp}\ (\mathsf{app}_2\ V_1)\ D\ d_2 \wedge \mathsf{eval}\ E_2\ d_2 \wedge (d_2 = E_2)$$
$$\mathsf{comp}\ (\mathsf{app}_2(\mathsf{lam}(\lambda x.E_0\, x))\ D\ D_2 \wedge \mathsf{retn}\ V_2\ D_2 \tag{38}$$
$$\supset\ \exists y.\ \exists d_0.\ \mathsf{comp}\ \mathsf{call}\ D\ d_0 \wedge \mathsf{eval}\ (E_0\, y)\ d_0 \wedge \mathsf{bind}\, y\, V_2 \wedge (y = \mathsf{var}(\lambda x.E_0\, x)) \wedge (d_0 = E_0\, y)$$
$$\mathsf{comp}\ \mathsf{call}\ D\ D_0 \wedge \mathsf{retn}\ V_0\ D_0\ \supset\ \mathsf{retn}\ V_0\ D \tag{39}$$

Fig. 4 Three stages of deriving a context-insensitive control-flow analysis from an SSOS specification.

parameter with the argument V_2. A second rule (rule 22) ensures that when we come across one of these parameters in the course of evaluation, we can "look up" the associated value by finding the (unique) fact associated with it in the form of a persistent proposition $\mathsf{!bind}\, X\, V$. This modification makes binding information amenable to approximation in roughly the same way destinations make control information amenable to approximation.

The first step in deriving a control flow analysis is the transformation of the ordered SSOS specification into a linear SSOS specification. We give the result of our translation in rules 28-33.[6] As we observed before, the vestigial D' destination is not needed, and

[6] This is precisely the result of the transformation we discuss in Section 2.2 with one exception: we replace existential quantification in the premise with universal quantification over the rule.

whenever we mention a persistent bind proposition we must put an equality in the rule even though the rule is obviously equivalent to a rule that does not have that equality in it.

In rules 34-39, we take a suitably equivalent version of the linear specification (one that does not have the vestigial third argument for eval and retn, nor the unnecessary uses of equality) and derive a persistent approximation that can be run as a saturating logic program. In rule 38, as our methodology suggests, we equate the parameter y with the Skolemization of the function it is being substituted into, $\mathsf{var}(\lambda x.E_0\, x)$. Therefore, assuming we start a saturating logic program with the single atomic proposition $\mathsf{eval}\, E\, D$, we only deal with subterms of E as defined by Section 3.2, and we can be certain that we have a saturating logic program that will terminate on any input. Furthermore, all rules maintain the invariant that when evaluating $\mathsf{eval}\, E\, D$, the parameter D is always equated with E – we could simplify the program further by only having a single argument to eval. The retn relation in the saturated program, then, is critical: $\mathsf{retn}\, V\, D$, where $D = E$ for some subexpression E, says that during the course of evaluation E may evaluate to the value V.

One question that flow analysis is intended to answer is, "for any given call site in the source program, what are the functions that might be invoked at that location?" Each fact of the form $\mathsf{comp}\,\mathsf{call}\, E\, E_0$ represents that the evaluation of E (the application expression that we are trying to evaluate to a value) may result in an attempt to evaluate E_0 (the body of the function that has been called). This means that the call site represented by the expression E will invoke only functions E_0 such that $\mathsf{comp}\,\mathsf{call}\, E\, E_0$ is in the saturated state. Conversely, the absence of such a fact means that the function with body E_0 can never be invoked from the call site E.

There is one important caveat to this analysis. If we take the program

$$\mathsf{app}\,(\mathsf{lam}(\lambda x.x))\,(\mathsf{app}\,(\mathsf{lam}(\lambda y.y))\,V)$$

we might expect a reasonable control flow analysis to notice that only $\mathsf{lam}(\lambda y.y)$ is passed to the function $\mathsf{lam}(\lambda x.x)$ and that only V is passed to the function $\mathsf{lam}(\lambda y.y)$. Because of our use of higher-order abstract syntax, $\mathsf{lam}(\lambda x.x)$ and $\mathsf{lam}(\lambda y.y)$ are syntactically identical (names of bound variables don't matter). This is obviously not a problem with correctness, but it means that our analysis may be less precise than expected. One solution would be to add distinct labels to term. Adding a label on the inside of every lambda-abstraction would seem to suffice, and in any real example labels would already be present in the form of source-code positions or line numbers. The next example, alias analysis, discusses the use of these labels.

4.2 Alias analysis

We will use our approximation methodology to derive an interprocedural alias analysis in the context of a simple language of straight-line functions and Lisp-like mutable pairs; the resulting approximation bears a strong resemblance to the object-oriented alias analysis presented as a logic program in [1, Chapter 12.4].

The language has the following syntax:

$$\begin{aligned} E &::= \mathsf{return}\, L\, X \mid \mathsf{let}\, L\, M\, (\lambda x.E\, x) \\ M &::= \mathsf{fun}\, (\lambda x.E_0\, x) \mid \mathsf{call}\, F\, X \mid \mathsf{newpair} \mid \mathsf{proj}\, X\, C \mid \mathsf{set}\, X\, C\, Y \\ C &::= \mathsf{fst} \mid \mathsf{snd} \end{aligned}$$

Expressions E should be thought of as sequences of let-bindings `let x = M in E` that bind the result of a command M in the remainder of a program or else choose a variable as the returned value of the function. Commands are either procedure definitions ($\mathsf{fun}\,(\lambda x.E_0\,x)$), procedure calls ($\mathsf{call}\,F\,X$ calls the procedure F with the argument X), create pairs (newpair), projections from the first or second component of a pair ($\mathsf{proj}\,X\,C$), or assignments to the first or second component of a pair ($\mathsf{set}\,X\,C\,Y$). Finally, each return statement or command is given a label L, which we can think of as a line number from the original program.

The rules for functions are unsurprising; as in the previous section, we use destinations for binding. We have comp and eval predicates. as before, though we can do without retn.[7]

$$\mathsf{eval}(\mathsf{let}\,L\,(\mathsf{fun}\,(\lambda x_0.E_0\,x_0))\,(\lambda x.E\,x)) \twoheadrightarrow \exists y.\ \mathsf{eval}(E\,y) \bullet !\mathsf{bind}\,y\,(\mathsf{lam}(\lambda x_0.E_0\,x_0))) \quad (40)$$

$$\mathsf{eval}(\mathsf{let}\,L\,(\mathsf{call}\,F\,X)\,(\lambda x.E\,x)) \bullet !\mathsf{bind}\,F\,(\mathsf{lam}(\lambda x_0.E_0\,x_0)) \bullet !\mathsf{bind}\,X\,V \quad (41)$$
$$\twoheadrightarrow \exists y.\ \mathsf{comp}(\mathsf{call}_1(\lambda x.E\,x)) \bullet \mathsf{eval}(E_0\,y) \bullet !\mathsf{bind}\,y\,V$$

$$\mathsf{comp}(\mathsf{call}_1(\lambda x.E\,x)) \bullet \mathsf{eval}(\mathsf{return}\,L\,X) \bullet !\mathsf{bind}\,X\,V \twoheadrightarrow \exists y.\ \mathsf{eval}(E\,y) \bullet !\mathsf{bind}\,y\,V \quad (42)$$

The rules for mutable pairs use linear atomic propositions for the first time in an ordered logical specification. Each destination D created by a newpair command is associated with two linear atomic propositions: ¡cell D fst V_1 contains the first projection V_1, and ¡cell D snd V_2 contains the second projection V_2, both of which are initially set to null.

$$\mathsf{eval}(\mathsf{let}\,L\,\mathsf{newpair}\,(\lambda x.E\,x)) \quad (43)$$
$$\twoheadrightarrow \exists y.\,\exists d.\ \mathsf{eval}(E\,y) \bullet !\mathsf{bind}\,y\,(\mathsf{loc}\,d) \bullet ¡\mathsf{cell}\,d\,\mathsf{fst}\,\mathsf{null} \bullet ¡\mathsf{cell}\,d\,\mathsf{snd}\,\mathsf{null}$$

$$\mathsf{eval}(\mathsf{let}\,L\,(\mathsf{proj}\,X\,C)\,(\lambda x.E\,x)) \bullet !\mathsf{bind}\,X\,(\mathsf{loc}\,D) \bullet ¡\mathsf{cell}\,D\,C\,V \quad (44)$$
$$\twoheadrightarrow \exists y.\ \mathsf{eval}(E\,y) \bullet !\mathsf{bind}\,y\,V \bullet ¡\mathsf{cell}\,D\,C\,V$$

$$\mathsf{eval}(\mathsf{let}\,L\,(\mathsf{set}\,X\,C\,Y)\,(\lambda x.E\,x)) \bullet !\mathsf{bind}\,X\,(\mathsf{loc}\,D) \bullet !\mathsf{bind}\,Y\,V \bullet ¡\mathsf{cell}\,D\,C\,V' \quad (45)$$
$$\twoheadrightarrow \exists y.\ \mathsf{eval}(E\,y) \bullet !\mathsf{bind}\,y\,\mathsf{null} \bullet ¡\mathsf{cell}\,D\,C\,V$$

When we approximate the specification in rules 40-45, our methodology forces our hand almost completely: once we equate every existential y with the Skolemization of the function it is being substituted into, there are only two existentially generated parameters left to consider: the destination generated when we create a new pair in (rule 43), and the destination created by the translation to linear destination-passing style in the rule handling procedure calls (rule 41). One option is just equate the destinations with the Skolemized function we are using in the same rule: this is shown in Figure 5, and the termination arguments are much the same as they were in the control-flow analysis.

Because the labels uniquely identify subterms of the original program, this alias analysis is not subject to the same caveat as the control flow analysis we considered before. Furthermore, these labels also allow us to consider an alternative to Skolemization. With one exception, the function call in rule 47, every Skolem function we generate is of the form $\mathsf{var}(\lambda x.E\,x)$ that originates from an expression ($\mathsf{let}\,L\,M\,(\lambda x.E\,x)$). In other

[7] Doing away with retn simplifies our presentation, but it is out of line with previous work [26] where we discuss a classification of predicates in SSOS specifications as *active*, *passive*, or *latent*. Because of rule 42 we cannot classify eval this way, though this could be corrected by splitting rule 42 into two rules, one which generates a retn and another which mentions comp.

$$\mathsf{eval}\ (\mathsf{let}\ L\ (\mathsf{fun}\ (\lambda x_0.E_0\ x_0))\ (\lambda x.E\ x))\ D \tag{46}$$
$$\supset \exists y.\ \mathsf{eval}\ (E\ y)\ D \wedge \mathsf{bind}\ y\ (\mathsf{lam}(\lambda x_0.E_0\ x_0))) \wedge (y = \mathsf{var}(\lambda x.E\ x))$$
$$\mathsf{eval}\ (\mathsf{let}\ L\ (\mathsf{call}\ F\ X)\ (\lambda x.E\ x))\ D \wedge \mathsf{bind}\ F\ (\mathsf{lam}(\lambda x_0.E_0\ x_0)) \wedge \mathsf{bind}\ X\ V \tag{47}$$
$$\supset \exists y.\ \exists d_0.\ \mathsf{comp}\ (\mathsf{call}_1(\lambda x.E\ x))\ D\ d_0 \wedge \mathsf{eval}\ (E_0\ y)\ d_0 \wedge \mathsf{bind}\ y\ V \wedge (y = d_0 = \mathsf{var}(\lambda x_0.E_0\ x_0))$$
$$\mathsf{comp}\ (\mathsf{call}_1(\lambda x.E\ x))\ D\ D_0 \wedge \mathsf{eval}\ (\mathsf{return}\ L\ X)\ D_0 \wedge \mathsf{bind}\ X\ V \tag{48}$$
$$\supset \exists y.\ \mathsf{eval}\ (E\ y)\ D \wedge \mathsf{bind}\ y\ V \wedge (y = \mathsf{var}(\lambda x.E\ x))$$
$$\mathsf{eval}\ (\mathsf{let}\ L\ \mathsf{newpair}\ (\lambda x.E\ x))\ D \tag{49}$$
$$\supset \exists y.\ \exists d.\ \mathsf{eval}\ (E\ y)\ D \wedge \mathsf{bind}\ y\ (\mathsf{loc}\ d) \wedge \mathsf{cell}\ d\ \mathsf{fst}\ \mathsf{null} \wedge \mathsf{cell}\ d\ \mathsf{snd}\ \mathsf{null} \wedge (y = d = \mathsf{var}(\lambda x.E\ x))$$
$$\mathsf{eval}\ (\mathsf{let}\ L\ (\mathsf{proj}\ X\ C)\ (\lambda x.E\ x))\ D \wedge \mathsf{bind}\ X\ (\mathsf{loc}\ D_X) \wedge \mathsf{cell}\ D_X\ C\ V \tag{50}$$
$$\supset \exists y.\ \mathsf{eval}\ (E\ y)\ D \wedge \mathsf{bind}\ y\ V \wedge \mathsf{cell}\ D_X\ C\ V \wedge (y = \mathsf{var}(\lambda x.E\ x))$$
$$\mathsf{eval}\ (\mathsf{let}\ L\ (\mathsf{set}\ X\ C\ Y)\ (\lambda x.E\ x))\ D \wedge \mathsf{bind}\ X\ (\mathsf{loc}\ D_X) \wedge \mathsf{bind}\ Y\ V \tag{51}$$
$$\supset \exists y.\ \mathsf{eval}\ (E\ y)\ D \wedge \mathsf{bind}\ y\ \mathsf{null} \wedge \mathsf{cell}\ D_X\ C\ V \wedge (y = \mathsf{var}(\lambda x.E\ x))$$

Fig. 5 Approximating the monadic language with mutable references by uniformly equating every existential variable with a Skolem function.

$$\mathsf{eval}\ (\mathsf{let}\ L\ \mathsf{newpair}\ (\lambda x.E\ x))\ D \tag{52}$$
$$\supset \mathsf{eval}\ (E\ L)\ D \wedge \mathsf{bind}\ L\ (\mathsf{loc}\ L) \wedge \mathsf{cell}\ L\ \mathsf{fst}\ \mathsf{null} \wedge \mathsf{cell}\ L\ \mathsf{snd}\ \mathsf{null}$$
$$\mathsf{eval}\ (\mathsf{let}\ L\ (\mathsf{proj}\ X\ C)\ (\lambda x.E\ x))\ D \wedge \mathsf{bind}\ X\ (\mathsf{loc}\ L_X) \wedge \mathsf{cell}\ L_X\ C\ V \tag{53}$$
$$\supset \mathsf{eval}\ (E\ L)\ D \wedge \mathsf{bind}\ L\ V \wedge \mathsf{cell}\ L_X\ C\ V$$
$$\mathsf{eval}\ (\mathsf{let}\ L\ (\mathsf{set}\ X\ C\ Y)\ (\lambda x.E\ x))\ D \wedge \mathsf{bind}\ X\ (\mathsf{loc}\ L_X) \wedge \mathsf{bind}\ Y\ V \tag{54}$$
$$\supset \mathsf{eval}\ (E\ L)\ D \wedge \mathsf{bind}\ L\ \mathsf{null} \wedge \mathsf{cell}\ L_X\ C\ V$$

Fig. 6 A version of Figure 5 labels instead of Skolemized functions, and where equivalent versions of the rules that do not explicitly mention equality are used.

words, each of these Skolem functions, can be uniquely associated with a label L, so we can think about using this label instead of $\mathsf{var}(\lambda x.E\ x)$. In Figure 6 we show a modified version of the rules for mutable state where labels are used instead of Skolem functions.

One benefit of the use of labels is that it makes the answer to some of the primary questions asked of an alias analysis much clearer. For instance, we want alias analysis to tell us whether the first or second component of a pair created at label L_1 can ever reference a pair created at label L_2. In the modified version of the rules, the first component of a pair created at label L_1 might reference a pair created at label L_2 if $\mathsf{cell}\ L_1\ \mathsf{fst}\ (\mathsf{loc}\ L_2)$ appears in the saturated database, and likewise for the second component.

5 Conclusion

We have defined a framework of ordered, linear, and persistent atomic propositions with higher-order terms and equality assertions. This framework is suitable for writing interpreters for programming languages in the style of substructural operational semantics. These specifications in ordered logic can be automatically transformed to linear logical specifications and then approximated as persistent logical specifications. By running these persistent logical specifications as saturating logic programs, we generate static

analyses of programs written in those languages. The relative ease of encoding two rather different analyses, alias analysis and control flow analysis, suggests that our technique can be used to derive other program analyses.

Our methodology presented Skolemization and the addition of equality constraints as essentially the only tool for restricting persistent logic programs in order to ensure termination. Another possibility considered in the conference version of this paper was to use explicit congruence rules such as allowing the term $\mathsf{s}(\mathsf{s}(0))$ to be equal to 0 in a non-contradictory way (and thereby only considering natural numbers modulo 2) and allowing lists that agree on their first k elements to be treated as equal. This is a powerful addition, but its interaction with the sequent calculus rules we gave for equality, which rely on the definition of higher-order unification, is not straightforward. There are two ways of thinking about equality in proof search and logic programming. The one we have adopted in this work is based on unification and is usually attributed to Girard and Schroeder-Heister [12,30]. Another view that is closer to the one from the conference version of this paper is based on constraints, has been explored in different settings by Virga and Saraswat et al. [34,14].

A richer set of approximation tools based on constraints or a more congruence-closure-like conception of equality could be developed entirely within the realm of saturating logic programs and still integrated into the development we have presented here. We have considered a two step methodology in this work: an automatic transformation of an ordered specification into a linear specification followed by a manual approximation of the linear specification to obtain a terminating bottom-up logic program. We could adapt our methodology to a three step process: an automatic transformation of an ordered specifications into a linear specifications, an automatic derivation of a kind of "collecting semantics" by simply turning linear predicates persistent, and finally a manual approximation of the resulting non-terminating saturating logic program by a terminating logic program. This means that the work we have presented here can potentially be extended with more powerful approximation techniques without dealing with the added complexity introduced by substructural logics.

5.1 Related work

This work is similar to work by Bozzano et al. [7,8] in both its goals and its methodology. They encode distributed systems and communication protocols in a framework that is roughly equivalent to the linear fragment of our specification framework without equality. Abstractions of those programs are then used to verify properties of concurrent protocols that were encoded in the logic [6]. However, the style they use to encode protocols is significantly different from our SSOS style of specification, and a general purpose approximation is used, in contrast to our methodology of describing a whole class of approximations. Furthermore, Bozzano et al.'s methods are designed to consider properties of systems as a whole, not static analyses of individual inputs as is the case in our work.

A fundamentally different kind of approximation of linear logic programs via predicate substitution has been described by [19]. Miller's approximations remain linear, which we have ruled out so far in order to obtain a simple meta-approximation theorem.

Another line of related work is the use of the ∇ quantifier [21] for name generation, which we could have used here instead of the existential quantifier, although the difference does not become significant until we carry out formal meta-reasoning [33]. As a

point of future work we conjecture it may be possible to apply our use of Skolemization in the realm of reasoning with and about generic judgments.

This article focuses on the process of deriving saturating logic programs from operational semantics specifications, but the work suggests many directions of research for saturating (i.e., bottom-up) logic programs in the persistent fragment. It is not obvious how the metacomplexity theorems for bottom-up logic programs originally presented by McAllester [17] should be generalized to a setting with equality and higher-order terms.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques and Tools, Second Edition. Pearson Education, Inc (2007)
2. Andreoli, J.: Focussing and proof construction. Annals of Pure and Applied Logic **107**(1-3), 131–163 (2001)
3. Andreoli, J.M.: Logic programming with focusing proofs in linear logic. J. Log. Comput. **2**(3), 297–347 (1992)
4. Armelín, P., Pym, D.: Bunched logic programming. In: IJCAR'01: Proceedings of the First International Joint Conference on Automated Reasoning, pp. 289–304. Springer LNCS 2083, Siena, Italy (2001)
5. Baelde, D., Miller, D.: Least and greatest fixed points in linear logic. In: N. Dershowitz, A. Voronkov (eds.) LPAR'07: Logic for Programming, Artificial Intelligence, and Reasoning, pp. 92–106. Springer LNCS 4790 (2007)
6. Bozzano, M., Delzanno, G.: Automated protocol verification in linear logic. In: PPDP'02: Proceedings of the 4th International Conference on Principles and Practice of Declarative Programming, pp. 38–49. ACM Press (2002)
7. Bozzano, M., Delzanno, G., Martelli, M.: An effective fixpoint semantics for linear logic programs. TPLP **2**(1), 85–122 (2002)
8. Bozzano, M., Delzanno, G., Martelli, M.: Model checking linear logic specifications. TPLP **4**(5-6), 573–619 (2004)
9. Cervesato, I., Pfenning, F., Walker, D., Watkins, K.: A concurrent logical framework II: Examples and applications. Tech. Rep. CMU-CS-02-102, School of Computer Science, Carnegie Mellon University (2002). Revised May 2003
10. Chaudhuri, K.: The Focused Inverse Method for Linear Logic. Ph.D. thesis, Carnegie Mellon University (2006)
11. Ganzinger, H., McAllester, D.A.: Logical algorithms. In: ICLP'02: Proceedings of the 18th International Conference on Logic Programming, pp. 209–223. Springer LNCS 2401, London, UK (2002)
12. Girard, J.Y.: A fixpoint theorem in linear logic (1992). An email posting to the mailing list linear@cs.stanford.edu
13. Hodas, J.S., Miller, D.: Logic programming in a fragment of intuitionistic linear logic. Inf. Comput. **110**(2), 327–365 (1994)
14. Jagadeesan, R., Nadathur, G., Saraswat, V.A.: Testing concurrent systems: An interpretation of intuitionistic logic. In: FSTTCS'05: Proceedings of the 25th International Conference on Foundations of Software Technology and Theoretical Computer Science, pp. 517–528. Springer LNCS 3821 (2005)
15. Lambek, J.: The mathematics of sentence structure. American Mathematical Monthly **65**, 363–386 (1958)
16. López, P., Pfenning, F., Polakow, J., Watkins, K.: Monadic concurrent linear logic programming. In: P. Barahona, A.P. Felty (eds.) PPDP'00, pp. 35–46. ACM (2005)
17. McAllester, D.A.: On the complexity analysis of static analyses. J. ACM **49**(4), 512–537 (2002)
18. Miller, D.: Uniform proofs as a foundation for logic programming. Annals of Pure and Applied Logic **51**(1-2), 125–157 (1991)
19. Miller, D.: A proof-theoretic approach to the static analysis of logic programs. In: Reasoning in Simple Type Theory: Festschrift in Honor of Peter B. Andrews on His 70th Birthday, Studies in Logic 17, pp. 423–442. College Publications, London (2008)

20. Miller, D., Nadathur, G., Pfenning, F., Scedrov, A.: Uniform proofs as a foundation for logic programming. Ann. Pure Appl. Logic **51**(1-2), 125–157 (1991)
21. Miller, D., Tiu, A.: A proof theory for generic judgments. ACM Transactions on Computational Logic **6**(4), 749–783 (2005)
22. Moot, R., Piazza, M.: Linguistic applications of first order intuitionistic linear logic. Journal of Logic, Language and Information **10**(2), 211–232 (2001)
23. Pfenning, F.: Substructural operational semantics and linear destination-passing style. In: W.N. Chin (ed.) APLAS'04: Proceedings of the 2nd Asian Symposium on Programming Languages and Systems, p. 196. Springer-Verlag LNCS 3302, Taipei, Taiwan (2004). Abstract of invited talk
24. Pfenning, F., Elliott, C.: Higher-order abstract syntax. In: PLDI, pp. 199–208 (1988)
25. Pfenning, F., Schürmann, C.: Algorithms for equality and unification in the presence of notational definitions. In: T. Altenkirch, W. Naraschewski, B. Reus (eds.) Types for Proofs and Programs, pp. 179–193. Springer LNCS 1657 (1998)
26. Pfenning, F., Simmons, R.J.: Substructural operational semantics as ordered logic programming. In: LICS, pp. 101–110. IEEE Computer Society (2009)
27. Plotkin, G.D.: A structural approach to operational semantics. Journal of Logic and Algebraic Programming **60-61**, 17–139 (2004). Reprinted with corrections from Aarhus University technical report DAIMI FN-19.
28. Polakow, J.: Linear logic programming with ordered contexts. In: PPDP 2000, pp. 68–79. ACM Press, Montreal, Canada (2000)
29. Polakow, J.: Ordered linear logic and applications. Ph.D. thesis, Carnegie Mellon University (2001). Available as technical report CMU-CS-01-152
30. Schroeder-Heister, P.: Rules of definitional reflection. In: M. Vardi (ed.) Eight Annual Symposium on Logic and Computer Science, pp. 222–232. IEEE Computer Scociety Press (1993)
31. Schürmann, C.: Automating the meta theory of deductive systems. Ph.D. thesis, Department of Computer Science, Carnegie Mellon University (2000). Available as Technical Report CMU-CS-00-146
32. Simmons, R.J., Pfenning, F.: Linear logical algorithms. In: Proceedings of the 35th International Colloquium on Automata, Languages and Programming (ICALP'08), pp. 336–345. Springer LNCS 5126, Reykjavik, Iceland (2008)
33. Tiu, A.: A logic for reasoning about generic judgments. In: Proceedings of the First International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'06), pp. 3–18 (2007). Published in ENTCS 174(5), June 2007
34. Virga, R.: Higher-order rewriting with dependent types. Ph.D. thesis, Carnegie Mellon University (1999)