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 2, 2010 / Revised: November 7, 2010 / Accepted: January 10, 2011

Abstract The abstract interpretation of programs relates the exact semantics of a programming language to a finite approximation of those semantics. In this article, we describe an approach to abstract interpretation that is based in logic and logic programming.

Our approach consists of faithfully representing a transition system within logic and then manipulating this initial specification to create a logical approximation of the original specification. The objective is to derive a logical approximation that can be interpreted as a terminating forward-chaining logic program; this ensures that the approximation is finite and that, furthermore, an appropriate logic programming interpreter can implement the derived approximation.

We are particularly interested in the specification of the operational semantics of programming languages in ordered logic, a technique we call *substructural operational semantics* (SSOS). We show that manifestly sound control flow and alias analyses can be derived as logical approximations of the substructural operational semantics of relevant languages.

1 Introduction

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

This work was supported by the Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) through the Carnegie Mellon Portugal Program under Grant NGN-44, and by a National Science Foundation Graduate Resource Fellowship for the first author.

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

A general recipe for constructing a sound program analysis is to (1) specify the operational semantics of the underlying programming language via an interpreter, and (2) specify a terminating approximation of the interpreter itself. This is the basic idea behind *abstract interpretation* [13] which provides techniques for constructing approximations (for example, by exhibiting a Galois connection between concrete and abstract domains). The correctness proof must establish the appropriate relationship between the concrete and abstract computations and show that the abstract computation terminates. We need to vary both the specification of the operational semantics and the form of the approximation in order to obtain various kinds of program analyses, sometimes with considerable ingenuity.

In this paper we propose a new class of instances of the general schema of abstract interpretation. We encode the transition rules of an evolving system in a specification framework based on ordered linear logic. We then apply logically justified techniques for manipulating and approximating the specification to yield approximations that are correct by construction. Furthermore, these approximations have the form of forward-chaining (or "bottom-up") logic programs which can be run to saturation, generalizing proposals by McAllester and Ganzinger [14, 21] with certain higher-order features.

Our particular interest is in the representation and static analysis of programming languages; we illustrate our approach by deriving control flow and alias analyses by logical approximation. Defining these specific approximations requires insight, but the correctness proofs do not, because they follow from a general metatheorem justifying the kinds of approximations we make, together with straightforward termination arguments.

1.1 A simple example

Many interesting stateful systems have a natural notion of *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:

$$[\text{ the stack }]\ [\text{ the head }]\ [\text{ the string being read }]$$

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

$$\mathsf{hd}\ \mathsf{<} \quad \rightsquigarrow \quad \mathsf{<}\ \mathsf{hd} \qquad (\mathsf{push})$$

$$\mathsf{<}\ \mathsf{hd}\ \mathsf{>} \quad \rightsquigarrow \quad \mathsf{hd} \qquad (\mathsf{pop})$$

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 relevant fragment of the beginning of the string and the top of the stack. 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. This is one possible series of transitions:

```
hd < < > < < > > >
< hd < > < < > > >
< < hd > < < > > >
    < hd < < > > >
    < < hd < > > >
    < < < hd > > >
        < < hd > >
            < hd >
                hd
```

We will use this simple example to give an overview of our approach, which has three steps: first, we represent the system in ordered linear logic; second, we translate the representation into linear logic; and third, we find an approximation of the system that can be interpreted as a forward-chaining logic program.

1.1.1 Representation in ordered linear logic

Because our goal is to use a framework that is both simple and logically motivated, we turn to *ordered logic* (originally presented by Lambek [19]), a logic where hypotheses have an intrinsic notion of order. Such logics are called *substructural* because they omit some or all of the structural properties of exchange, weakening and contraction.

The rewriting rules we considered above 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); it binds more tightly than ↠, a binary connective for ordered implication.

$$\mathsf{hd} \bullet \mathsf{<} \twoheadrightarrow \mathsf{<} \bullet \mathsf{hd} \qquad (\mathsf{push}_1)$$

$$\mathsf{<} \bullet \mathsf{hd} \bullet \mathsf{>} \twoheadrightarrow \mathsf{hd} \qquad (\mathsf{pop}_1)$$

We call collections of rules like this *specifications*. In this paper, we use a variant of ordered linear logic [37], a generalization of both Lambek's ordered logic and dual intuitionistic linear logic [4]. This logic is first-order, which lets us generalize the specification above to an arbitrary collection of left and right brackets – the string "`[<>()]`" could be represented by the following sequence of ordered atomic propositions:

$$(\mathsf{left\ square})\ (\mathsf{left\ angle})\ (\mathsf{right\ angle})\ (\mathsf{left\ paren})\ (\mathsf{right\ paren})\ (\mathsf{right\ square}) \qquad (1)$$

The following rules describe the more general push-down automaton (following common convention, the upper-case X is implicitly universally quantified):

$$\mathsf{hd} \bullet \mathsf{left}\ X \twoheadrightarrow \mathsf{stack}\ X \bullet \mathsf{hd} \qquad (\mathsf{push}_2)$$

$$\mathsf{stack}\ X \bullet \mathsf{hd} \bullet \mathsf{right}\ X \twoheadrightarrow \mathsf{hd} \qquad (\mathsf{pop}_2)$$

Note that we follow a common convention in this example: while we use the fuse connective to indicate adjacent tokens in the rules above, we do not use any marker to indicate adjacent tokens in the sequence (1) above. Sequences of atomic propositions like the one above will be subsequently referred to as *ordered contexts*.

The encoding of state transitions as implications in substructural logics in this way is a powerful and general technique; we discuss some related work in the conclusion. The particular rewriting interpretation of the ordered linear logic that we will use, an extension of the one introduced in [34], is outlined in Section 2.

1.1.2 Translation into linear logic

After we represent the transitions of our system as propositions in ordered linear logic, the next step of our methodology is to translate our rules from ordered linear logic into linear logic. In linear logic, we deal not with sequences of atomic propositions but with multisets of atomic propositions – there is no inherent notion of order. Therefore, the translation adds two arguments, called *destinations*, to every ordered atomic proposition. These destinations represent the proposition's "left-hand side" and "right-hand side," and together they describe what position the atomic proposition would have in the ordered context. The ordered context labeled (1) above looks something like this under the destination-adding translation:

$$(\mathsf{left\ square}\, d_1\, d_2)\ \ (\mathsf{left\ angle}\, d_2\, d_3)\ \ (\mathsf{right\ angle}\, d_3\, d_4)\ \ (\mathsf{left\ paren}\, d_4\, d_5)\ \ \ldots \qquad (2)$$

In Section 3, we present the destination-adding translation from ordered linear logic to linear logic and show that it is sound. When this translation is applied to the rules describing our push-down automaton, we get the following:

$$\mathsf{hd}\, L\, M\ \otimes\ \mathsf{left}\, X\, M\, R\ \multimap\ \exists m.\ \mathsf{stack}\, X\, L\, m\ \otimes\ \mathsf{hd}\, m\, R \qquad (\mathsf{push_3})$$
$$\mathsf{stack}\, X\, L\, M_1\ \otimes\ \mathsf{hd}\, M_1\, M_2\ \otimes\ \mathsf{right}\, X\, M_2\, R\ \multimap\ \mathsf{hd}\, L\, R \qquad (\mathsf{pop_3})$$

Note the $\exists m$ in the conclusion of rule $\mathsf{push_3}$. Existential quantification in the conclusion of a rule represents the creation of a fresh new parameter – in fact, there are no constants (like paren or angle) or term constructors for destinations; they are always represented by parameters. By using the rule $\mathsf{push_3}$ we can therefore have a transition from a state containing the linear atomic propositions $(\mathsf{hd}\, d_0\, d_1)$ and $(\mathsf{left\ square}\, d_1\, d_2)$ to a state containing the linear atomic propositions $(\mathsf{stack\ square}\, d_0\, d')$ and $(\mathsf{hd}\, d'\, d_2)$, where d' is a freshly generated parameter. The former state has the parameters d_0, d_1, and d_2 free, and the latter state has the parameters d_0, d', and d_2 free.

1.1.3 Approximation as a logic program

After the destination-adding translation, we next take the step of "forgetting" about linearity, which produces the following two rules in a standard first-order intuitionistic logic ($A \supset B$ is the way we write "A implies B"):

$$\mathsf{hd}\, L\, M\ \wedge\ \mathsf{left}\, X\, M\, R\ \supset\ \exists m.\ \mathsf{stack}\, X\, L\, m\ \wedge\ \mathsf{hd}\, m\, R \qquad (\mathsf{push_4})$$
$$\mathsf{stack}\, X\, L\, M_1\ \wedge\ \mathsf{hd}\, M_1\, M_2\ \wedge\ \mathsf{right}\, X\, M_2\, R\ \supset\ \mathsf{hd}\, L\, R \qquad (\mathsf{pop_4})$$

Forgetting about linearity produces an approximation – the resulting sepecification is sound, but not complete, with respect to the original specification. It is a slight digression, but it is worth mentioning that there is a class of logical specifications with two important properties: (1) the resulting logical specification is sound *and complete* with respect to the linear (and ordered) logical specifications, and (2) the resulting logical specification can be effectively interpreted as a forward-chaining (or "bottom-up") logic program. An interpreter for a forward-chaining logic program takes a collection of facts and exhaustively derives new facts until no new facts can be derived, at which point the program is said to reach *saturation*. These specifications are of particular interest to the natural language processing community, as described in Shieber, Schabes, and Pereira's classic work on *deductive parsing* [43].

In our setting, the two properties that deductive parsing relies on do not hold in general. Our running example illustrates this: the rules above cannot be executed as a forward-chaining logic program because of the existential quantifier in the conclusion of rule $\mathsf{push_4}$. This existential quantifier can always be used to generate a distinct new parameter and, therefore, two distinct new facts, so that a naïve attempt at saturation might look like this:

Start with the facts:	$(\mathsf{hd}\ d_0\ d_1)$, $(\mathsf{left\ angle}\ d_1\ d_2)$, and $(\mathsf{right\ angle}\ d_2\ d_3)$	
From $\mathsf{push_4}$, add the facts:	$(\mathsf{stack\ angle}\ d_0\ d')$ and $(\mathsf{hd}\ d'\ \mathsf{d_1})$	(d' is fresh)
From $\mathsf{push_4}$, add the facts:	$(\mathsf{stack\ angle}\ d_0\ d'')$ and $(\mathsf{hd}\ d''\ d_1)$	(d'' is fresh)
From $\mathsf{push_4}$, add the facts:	$(\mathsf{stack\ angle}\ d_0\ d''')$ and $(\mathsf{hd}\ d'''\ d_1)$	(d''' is fresh)
...	...	...

However, we can approximate this specification by taking the existential quantifier in the conclusion of $\mathsf{push_4}$ and equating the stray parameter it generates to some concrete term. For instance, if we set m equal to M, we get the following rule:

$$\mathsf{hd}\ L\ M\ \wedge\ \mathsf{left}\ X\ M\ R\ \supset\ \mathsf{stack}\ X\ L\ M\ \wedge\ \mathsf{hd}\ M\ R \qquad (\mathsf{push_{4M}})$$

In this case, switching out $\mathsf{push_4}$ for $\mathsf{push_{4M}}$ actually yields a precise approximation that exactly captures the behavior of the original specification, which is not possible in general. On the other hand, if we set m equal to L, we get the following rule:

$$\mathsf{hd}\ L\ M\ \wedge\ \mathsf{left}\ X\ M\ R\ \supset\ \mathsf{stack}\ X\ L\ L\ \wedge\ \mathsf{hd}\ L\ R \qquad (\mathsf{push_{4L}})$$

If the initial collection of facts contains a single atomic proposition $(\mathsf{hd}\ d_0\ d_1)$ in addition to all the left and right facts, then both $\mathsf{push_{4L}}$ and $\mathsf{pop_4}$ maintain the invariant that, as new facts are derived, the first argument of hd and the second and third arguments of stack are equal to d_0. These arguments are therefore uninteresting, and we can just remove them from the approximate specification, resulting in this specification:

$$\mathsf{hd}\ M\ \wedge\ \mathsf{left}\ X\ M\ R\ \supset\ \mathsf{stack}\ X\ \wedge\ \mathsf{hd}\ R \qquad (\mathsf{push_5})$$

$$\mathsf{stack}\ X\ \wedge\ \mathsf{hd}\ M_2\ \wedge\ \mathsf{right}\ X\ M_2\ R\ \supset\ \mathsf{hd}\ R \qquad (\mathsf{pop_5})$$

This logical approximation of the original specification accepts every string where, for every form of bracket X, at least one left X appears before any of the right X, so the string "`[]]](()`" would be accepted but the string "`][[]`" would not, as the right bracket appears before any left bracket.

Section 4 covers strategies for approximating logical specifications and the meta-approximation theorem that ensures the correctness of these strategies.

1.2 Substructural operational semantics

The preceding example explained our methodology, but we are not primarily interested in representing things like PDAs, and we are not at all interested in deriving overly generous parenthesis checking algorithms. What we *are* interested in is the representation of the operational semantics of programming languages. We represent operational semantics in our ordered linear logical framework using a style known as *substructural*

operational semantics (SSOS).[1] SSOS is a synthesis of structural operational semantics, abstract machines, and logical specifications where machine states are represented by collections of atomic propositions.

A 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 (where K is a series of frames $F_1, \ldots, F_n$), and $(K \lhd V)$, representing a value V being returned to K. In SSOS specifications we represent the stack K not as a single syntactic object but as a sequence of ordered atomic propositions $\mathsf{comp}(F)$, each of which contains a single stack frame. The name "comp" was chosen because we think of the atomic proposition as representing a suspended **comp**utation. An expression E being evaluated on a stack is represented by an atomic proposition $\mathsf{eval}(E)$, and the state $(F_1, \ldots, F_n \rhd E)$ is represented by the sequence of ordered atomic propositions $(\mathsf{comp}(F_1) \ \ldots \ \mathsf{comp}(F_n)\, \mathsf{eval}(E))$. Similarly, a value V being returned to a stack (the V in $K \lhd V$) is represented by an ordered atomic proposition $\mathsf{retn}(V)$.

Our running example will be a call-by-value operational semantics for an untyped lambda calculus; we will eventually derive a control flow analysis from this specification. The syntax of the lambda calculus is represented using *higher-order abstract syntax* [32], so we represent a lambda term as $\mathsf{lam}(\lambda x.E_0\, x)$. The evaluation of a lambda expression is simple: a function is already a value, so we return it.

$$\mathsf{eval}(\mathsf{lam}(\lambda x.E_0\, x)) \ \twoheadrightarrow \ \mathsf{retn}(\mathsf{lam}(\lambda x.E_0\, x)) \qquad (\mathsf{e/lam})$$

The evaluation of an application $\mathsf{app}\, E_1\, E_2$ requires us to generate a new stack frame $(\mathsf{app}_1\, E_2)$ that suspends the function argument E_2 while E_1 is being evaluated to a value.

$$\mathsf{eval}(\mathsf{app}\, E_1\, E_2) \ \twoheadrightarrow \ \mathsf{comp}(\mathsf{app}_1\, E_2) \ \bullet \ \mathsf{eval}(E_1) \qquad (\mathsf{e/app})$$

When a value is returned to a waiting app_1 frame, we switch to evaluating the function argument while storing the returned value V_1 on the stack. The value V_1 had better be a function $\mathsf{lam}(\lambda x.\, E_0\, x)$, but we do not actually assert this.

$$\mathsf{comp}(\mathsf{app}_1\, E_2) \ \bullet \ \mathsf{retn}(V_1) \ \twoheadrightarrow \ \mathsf{comp}(\mathsf{app}_2\, V_1) \ \bullet \ \mathsf{eval}(E_2) \qquad (\mathsf{e/app_1})$$

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 – $E_0\, V_2$ can be understood as $E_0[V_2/x]$.

$$\mathsf{comp}(\mathsf{app}_2(\mathsf{lam}(\lambda x.E_0\, x))) \ \bullet \ \mathsf{retn}(V_2) \ \twoheadrightarrow \ \mathsf{eval}(E_0\, V_2) \qquad (\mathsf{e/app_2})$$

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. As before, each machine state is represented by an ordered context, so the "fuse" connective that appears in the

[1] The term *substructural operational semantics* merges structural operational semantics [35], which we seek to generalize, and substructural logic, which we use as our specification framework.

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

Fig. 1 A 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 in Section 1.2.

rules $\mathsf{e/app}$, $\mathsf{e/app_1}$, and $\mathsf{e/app_2}$ does not appear 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 [34].

1.3 Outline

The outline of this paper mirrors the discussion in Section 1.1. In Section 2 we revisit the use of ordered linear logic as a specification framework, and in Section 3 we discuss the translation from ordered linear logic into linear logic. In Section 4 we discuss approximation of ordered linear logical specifications, as well as the conditions under which those approximations can be run as saturating, forward-chaining logic programs. Throughout the paper we show how a control flow analysis can be derived from the previous SSOS specification of the call-by-value lambda calculus, and in Section 5 we apply the same techniques to derive an alias analysis by logical approximation.

2 Representation in ordered linear logic

As we described in the introduction, we are interested in using propositions in ordered linear logic, such as $(\mathsf{<} \bullet \mathsf{hd} \bullet \mathsf{>} \twoheadrightarrow \mathsf{hd})$, to represent transitions in systems such as the push-down automaton that we specified in Section 1.1. In order to explain the logical interpretation of propositions in ordered logic, we need to give the proof rules for ordered logic; a relevant subset of these rules is given in Figure 2. Sequents have the form $(\Gamma; \Omega \vdash C)$ where Ω is a sequence of hypotheses that must be used exactly once in the specified order and Γ is a set of *valid* or *persistent* hypotheses that may be used any number of times in any order.

In Figure 3 we can see the logical interpretation of the two push-down automaton transitions $\mathsf{<hd<>>} \rightsquigarrow \mathsf{<<hd>>} \rightsquigarrow \mathsf{<hd>}$ using the proof rules in Figure 2. (We assume Γ contains the two rules $\mathsf{pop_1}$ and $\mathsf{push_1}$ from Section 1.1.) We read the PDA transitions off of these two derivations by examining them from the bottom to the top:

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

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

$$\frac{}{\Gamma; Q \vdash Q}\ \textbf{init} \qquad \frac{A \in \Gamma \quad \Gamma; \Omega_L\ A\ \Omega_R \vdash C}{\Gamma; \Omega_L\Omega_R \vdash C}\ \textbf{copy}$$

Fig. 2 A subset of the sequent calculus rules for ordered logic. The metavariable Q in the **init** rule stands for an arbitrary atomic proposition.

$$\dfrac{\dfrac{}{(\mathsf{<} \bullet \mathsf{hd} \bullet \mathsf{>} \twoheadrightarrow \mathsf{hd}) \in \Gamma} \qquad \dfrac{\dfrac{}{\Gamma; \mathsf{<} \vdash \mathsf{<}}\ \textbf{init} \quad \dfrac{\dfrac{}{\Gamma; \mathsf{hd} \vdash \mathsf{hd}}\ \textbf{init} \quad \dfrac{}{\Gamma; \mathsf{>} \vdash \mathsf{>}}\ \textbf{init}}{\Gamma; \mathsf{hd} \mathsf{>} \vdash \mathsf{hd} \bullet \mathsf{>}}\ \bullet_R}{\Gamma; \mathsf{<hd>} \vdash \mathsf{<} \bullet \mathsf{hd} \bullet \mathsf{>}}\ \bullet_R \qquad \Gamma; \mathsf{<hd>} \vdash C}{\Gamma; \mathsf{<} (\mathsf{<} \bullet \mathsf{hd} \bullet \mathsf{>} \twoheadrightarrow \mathsf{hd}) \mathsf{<hd>>} \vdash C}\ {\twoheadrightarrow}_L}{\Gamma; \mathsf{<<hd>>} \vdash C}\ \textbf{copy}$$

$$\vdots$$

$$\dfrac{\dfrac{}{(\mathsf{hd} \bullet \mathsf{<} \twoheadrightarrow \mathsf{<} \bullet \mathsf{hd}) \in \Gamma} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma; \mathsf{hd} \vdash \mathsf{hd}}\ \textbf{init} \quad \dfrac{}{\Gamma; \mathsf{<} \vdash \mathsf{<}}\ \textbf{init}}{\Gamma; \mathsf{hd} \mathsf{<} \vdash \mathsf{hd} \bullet \mathsf{<}}\ \bullet_R \qquad \dfrac{\vdots}{\Gamma; \mathsf{<<hd>>} \vdash C}}{\Gamma; \mathsf{<} (\mathsf{<} \bullet \mathsf{hd}) \mathsf{>>} \vdash C}\ \bullet_L}{\Gamma; \mathsf{<} (\mathsf{hd} \bullet \mathsf{<} \twoheadrightarrow \mathsf{<} \bullet \mathsf{hd}) \mathsf{hd<>>} \vdash C}\ {\twoheadrightarrow}_L}{\Gamma; \mathsf{<hd<>>} \vdash C}\ \textbf{copy}$$

Fig. 3 A derivation in ordered logic (split up into two parts) that represents the push-down automaton transitions $\mathsf{<hd<>>} \rightsquigarrow \mathsf{<<hd>>}$ (bottom) and $\mathsf{<<hd>>} \rightsquigarrow \mathsf{<hd>}$ (top). The open leaf of the lower tree is the same as the root of the upper tree, so the two derivations could be fused.

the first state of the PDA, $\mathsf{<hd<>>}$, is encoded in the sequent $(\Gamma; \mathsf{<hd<>>} \vdash C)$ at the base of the derivation, the second state $\mathsf{<<hd>>}$ is encoded in the middle sequent $(\Gamma; \mathsf{<<hd>>} \vdash C)$, and the third state $\mathsf{<hd>}$ is encoded in the sequent $(\Gamma; \mathsf{<hd>} \vdash C)$ in the upper-right portion of Figure 3. This last sequent does not have a derivation to prove it, so the derivation in Figure 3 is "open" or incomplete.

The sequent calculus in Figure 2 does a perfectly good job of defining a logic. However, we are interested in using logic to represent transition systems, and the rules in Figure 2 do not quite serve this purpose. The particular derivation presented in Figure 3 cleanly separates the parts that belong to the two different PDA transitions. However, the logic presented in Figure 2 does not enforce this clean separation. For instance, we can alter the derivation by moving the "upper" use of the **copy** proof rule down, so that derivation begins with two applications of the **copy** proof rule, and we can similarly move the application of the $\bullet_L$ proof rule in the bottom derivation up to the top derivation, perhaps into the branch where the conclusion is $\mathsf{<} \bullet \mathsf{hd} \bullet \mathsf{>}$.

In order to introduce a well-defined notion of transition, we instead base our specification framework on a restricted form of sequent calculus with a notion of *focus*. Focusing, introduced by Andreoli [2], classifies propositions as either *positive* or *negative*. A positive proposition S, which can be an atomic proposition Q or an ordered conjunction $(S_1 \bullet S_2)$, can be put in *right focus* in a sequent $(\Gamma; \Omega \Rightarrow [S])$. The *only*

way we can prove such a sequent is by applying a proof rule that breaks down S. The following are the right focus rules for the focused version of the logic in Figure 2:

$$\frac{\Gamma; \Omega_L \Rightarrow [S_1] \quad \Gamma; \Omega_R \Rightarrow [S_2]}{\Gamma; \Omega_L \Omega_R \Rightarrow [S_1 \bullet S_2]}\ \bullet_R \qquad \frac{}{\Gamma; Q \Rightarrow [Q]}\ \textbf{init}$$

When we are focused on the right, it is not possible to apply left rules; therefore, the only way a sequent ($\Gamma; \Omega \Rightarrow [\mathsf{<} \bullet \mathsf{hd} \bullet \mathsf{>}]$) will be provable is if $\Omega = \mathsf{<hd>}$. Left focus, written as ($\Gamma; \Omega_L[A]\Omega_R \Rightarrow C$), has a similar role for negative propositions A – the only negative proposition A we have considered so far is $S_1 \twoheadrightarrow S_2$.[2] Left focus prevents the **copy** rule from being applied twice in a row, because we can only apply the **copy**-like rule (now called $\textbf{focus}_L$) when we are not in focus, and then the copied proposition goes into focus:

$$\frac{A \in \Gamma \quad \Gamma; \Omega_L[A]\Omega_R \Rightarrow C}{\Gamma; \Omega_L \Omega_R \Rightarrow C}\ \textbf{focus}_L \qquad \frac{\Gamma; \Omega_1 \Rightarrow [S_1] \quad \Gamma; \Omega_L\, S_2\, \Omega_R \Rightarrow C}{\Gamma; \Omega_L[S_1 \twoheadrightarrow S_2]\Omega_1\Omega_R \Rightarrow C}\ \twoheadrightarrow_L$$

An important property of the focused system is that it is both sound and complete with respect to an unfocused sequent calculus, which means that there is a derivation of ($\Gamma; \Omega \vdash A$) under proof rules like those in Figure 2 if and only if a focused derivation of ($\Gamma; \Omega \Rightarrow A$) exists. This result follows from the internal soundness and completeness of the focused proof system, which we verify by proving the admissibility of cut and identity principles. A full discussion of this point would take us too far afield; we refer the interested reader to [34,47] for details.

2.1 Ordered logical specifications

The previous discussion was intended to motivate the design of a logical framework based on propositional ordered logic. In this section, we will present the full specification framework, which is based on first-order ordered linear logic [37]. This framework is a slight generalization of the ordered logical framework previously presented in [34]. In Section 2.3 we give a state transition interpretation of ordered logic programming using this system; this interpretation defines a notion of transition that exactly corresponds to the notion of transition in the push-down automata and programming languages we are representing.

As discussed, we categorize propositions in the focused framework as either *negative* propositions A (which we call *rules*) or as *positive* propositions S. Atomic propositions Q are first-order and so can contain terms t.

Atomic Propositions	$Q, Q_l, Q_p ::= p\ t_1 \ldots t_n$
Negative Propositions	$A, B ::= \forall x.A \mid S_1 \twoheadrightarrow S_2$
Positive Propositions	$S ::= Q \mid ¡Q_l \mid\ !Q_p \mid S_1 \bullet S_2 \mid \exists x.S \mid \mathbf{1} \mid t \doteq s$

[2] The most general form of right ordered implication is $S \twoheadrightarrow A$; the rule $\twoheadrightarrow_L$ would, in this case, remain focused on A in the second premise, and there would be a second rule **blur** with premise ($\Gamma; \Omega_L\, S\, \Omega_R \Rightarrow C$) and conclusion ($\Gamma; \Omega_L[S]\Omega_R \Rightarrow C$). Our definition of $\twoheadrightarrow_L$ consolidates the more general $\twoheadrightarrow_L$ and **blur** rules in a manner suitable for our framework.

Initial Rules

$$\frac{}{\Gamma; \cdot; Q \Rightarrow_\Sigma [Q]}\ \mathbf{init} \qquad \frac{}{\Gamma; Q_l; \cdot \Rightarrow_\Sigma [¡Q_l]}\ \mathbf{init}_¡ \qquad \frac{Q_p \in \Gamma}{\Gamma; \cdot; \cdot \Rightarrow_\Sigma [!Q_p]}\ \mathbf{init}!$$

Focusing Rules

$$\frac{A \in \Gamma \quad \Gamma; \Delta; \Omega_L[A]\Omega_R \Rightarrow_\Sigma C}{\Gamma; \Delta; \Omega_L\Omega_R \Rightarrow_\Sigma C}\ \mathbf{focus}_L \qquad \frac{\Gamma; \Delta; \Omega \Rightarrow_\Sigma [S]}{\Gamma; \Delta; \Omega \Rightarrow_\Sigma S}\ \mathbf{focus}_R$$

Ordered Implication

$$\frac{\Gamma; \Delta; \Omega\, S_1 \Rightarrow_\Sigma S_2}{\Gamma; \Delta; \Omega \Rightarrow_\Sigma S_1 \twoheadrightarrow S_2}\ \twoheadrightarrow_R \qquad \frac{\Gamma; \Delta_1; \Omega_1 \Rightarrow_\Sigma [S_1] \qquad \Gamma; \Delta_2; \Omega_L S_2 \Omega_R \Rightarrow_\Sigma C}{\Gamma; \Delta_1\Delta_2; \Omega_L[S_1 \twoheadrightarrow S_2]\Omega_1\Omega_R \Rightarrow_\Sigma C}\ \twoheadrightarrow_L$$

Conjunction

$$\frac{\Gamma; \Delta_1; \Omega_L \Rightarrow_\Sigma [S_1] \quad \Gamma; \Delta_2; \Omega_R \Rightarrow_\Sigma [S_2]}{\Gamma; \Delta_1\Delta_2; \Omega_L\Omega_R \Rightarrow_\Sigma [S_1 \bullet S_2]}\ \bullet_R \qquad \frac{\Gamma; \Delta; \Omega_L S_1 S_2 \Omega_R \Rightarrow_\Sigma C}{\Gamma; \Delta; \Omega_L(S_1 \bullet S_2)\Omega_R \Rightarrow_\Sigma C}\ \bullet_L$$

$$\frac{}{\Gamma; \cdot; \cdot \Rightarrow_\Sigma [\mathbf{1}]}\ \mathbf{1}_R \qquad \frac{\Gamma; \Delta; \Omega_L\Omega_R \Rightarrow_\Sigma C}{\Gamma; \Delta; \Omega_L(\mathbf{1})\Omega_R \Rightarrow_\Sigma C}\ \mathbf{1}_L$$

Modalities

$$\frac{\Gamma Q; \Delta; \Omega_L\Omega_R \Rightarrow_\Sigma C}{\Gamma; \Delta; \Omega_L(!Q)\Omega_R \Rightarrow_\Sigma C}\ !_L \qquad \frac{\Gamma; \Delta Q; \Omega_L\Omega_R \Rightarrow_\Sigma C}{\Gamma; \Delta; \Omega_L(¡Q)\Omega_R \Rightarrow_\Sigma C}\ ¡_L$$

Quantifiers

$$\frac{\Gamma; \Delta; \Omega \Rightarrow_{\Sigma,a} A[a/x]}{\Gamma; \Delta; \Omega \Rightarrow_\Sigma \forall x.A}\ \forall^a_R \qquad \frac{\Gamma; \Delta; \Omega_L[A[t/x]]\Omega_R \Rightarrow_\Sigma C}{\Gamma; \Delta; \Omega_L[\forall x.A]\Omega_R \Rightarrow_\Sigma C}\ \forall_L$$

$$\frac{\Gamma; \Delta; \Omega \Rightarrow_\Sigma [S[t/x]]}{\Gamma; \Delta; \Omega \Rightarrow_\Sigma [\exists x.S]}\ \exists_R \qquad \frac{\Gamma; \Delta; \Omega_L(S[a/x])\Omega_R \Rightarrow_{\Sigma,a} C}{\Gamma; \Delta; \Omega_L(\exists x.S)\Omega_R \Rightarrow_\Sigma C}\ \exists^a_L$$

Equality

$$\frac{}{\Gamma; \cdot; \cdot \Rightarrow_\Sigma [t \doteq t]}\ \doteq_R \qquad \frac{\theta\Gamma; \theta\Delta; \theta\Omega_L\, \theta\Omega_R \Rightarrow_{\Sigma'} \theta S \quad \text{for all} \quad \Sigma' \vdash \theta : \Sigma \quad \text{s.t.} \quad \theta t = \theta s}{\Gamma; \Delta; \Omega_L(t \doteq s)\Omega_R \Rightarrow_\Sigma S}\ \doteq_L$$

Fig. 4 A weakly focused sequent calculus for the fragment of ordered linear 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.

We also define the structure of contexts.

Persistent contexts	$\Gamma ::= \cdot \mid A \mid Q_p \mid \Gamma\,\Gamma'$
Linear contexts	$\Delta ::= \cdot \mid Q_l \mid \Delta\,\Delta'$
Ordered contexts	$\Omega ::= \cdot \mid S \mid \Omega\,\Omega'$
Parameter contexts	$\Sigma ::= \cdot \mid x \mid \Sigma\Sigma'$

Persistent, linear, and parameter contexts are treated as equivalent up to reordering (with "$\cdot$" representing an empty context), but order matters in ordered contexts – when representing our PDA example in an ordered context, it would not be desirable to treat ($\mathsf{hd}\,{<}\,{>}$) and ($\mathsf{hd}\,{>}\,{<}$) as equivalent contexts! Additionally, all the parameters in a context Σ are required to be distinct.

We have seen examples 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. Our specification framework also includes $\mathbf{1}$, the unit of ordered conjunction, existential quantification $\exists x.S$, and universal quantification $\forall x.S$. In addition

to ordered atomic propositions we have *linear* atomic propositions ¡Q_l and *persistent* atomic propositions !Q_p. These atomic propositions are treated as syntactically distinct from one another, so that (for example) a persistent atomic proposition is always preceded by a "!" and an ordered atomic proposition never is. This restriction, elsewhere referred to as *separation* [34,45], has an interpretation in the proof theory of ordered linear logic [47].

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.[3] Linear atomic propositions, on the other hand, act like consumable resources but are not ordered: while the rules $(Q_1 \bullet Q_2 \twoheadrightarrow S)$ and $(Q_2 \bullet Q_1 \twoheadrightarrow S)$ are *not* equivalent, the rules $(¡Q_1 \bullet ¡Q_2 \twoheadrightarrow S)$ and $(¡Q_2 \bullet ¡Q_1 \twoheadrightarrow S)$ are. This is a direct consequence of the aforementioned fact that we treat linear contexts Δ as equivalent up to reordering but do not treat ordered contexts Ω the same way.

The logic is defined in terms of the three kinds of sequents: unfocused sequents $(\Gamma; \Delta; \Omega \Rightarrow_\Sigma C)$, left-focused sequents $(\Gamma; \Delta; \Omega_L[A]\Omega_R \Rightarrow_\Sigma C)$, and right-focused sequents $(\Gamma; \Delta; \Omega \Rightarrow_\Sigma [S])$ – the metavariable C can stand for either a positive or a negative proposition. The rules of focused ordered linear logic are given in Figure 4. As usual for first-order sequent calculi, we stipulate that all contexts and propositions in a sequent are well-defined only if all the free parameters are bound in Σ; the proof rules $\exists^a_L$ and $\forall^a_R$ bind new parameters a that must not be free in the conclusion.

The only addition to our logic relative to the framework of ordered logical specifications described in previous work [34] is a proposition $(t \doteq s)$ describing equality between terms. The notation $\Sigma' \vdash \theta : \Sigma$ means that θ, a substitution, acts like a function that can take terms, propositions and contexts well-defined under the parameters Σ to terms, propositions, and contexts well-defined under the parameters in Σ' by replacing all parameters in Σ uniformly with terms well-defined in Σ'. A substitution θ is a unifier of the terms t and s if θt is identical to θs (up to renaming of bound variables). The left rule $\doteq_L$ uses unifiers and is *infinitary* – there is a premise for *every* unifier of the terms t and s, potentially infinitely many. This presentation of $\doteq_L$ is a variant of the standard presentation, which involves *complete sets of unifiers* [15,40].

There are well-know conditions, such as the restriction to first-order terms or to the *pattern fragment* [25], under which terms are known to always have either no unifiers or a *most general unifier* of t and s (written "mgu(t, s)"), a unifier θ with the property that, for any other unifier σ, $\sigma = \sigma' \circ \theta$ for some substitution σ'. If we syntactically restrict unification in this way, the infinitary rule can be replaced by the two rules below while retaining the completeness of the logic:

$$\frac{t \text{ and } s \text{ have no unifier}}{\Gamma; \Delta; \Omega_L(t \doteq s)\Omega_R \Rightarrow_\Sigma S} \not\doteq_L \qquad \frac{\Sigma' \vdash \theta : \Sigma = \text{mgu}(t, s) \quad \theta\Gamma; \theta\Delta; \theta\Omega_L\, \theta\Omega_R \Rightarrow_{\Sigma'} \theta S'}{\Gamma; \Delta; \Omega_L(t \doteq s)\Omega_R \Rightarrow_\Sigma S} \doteq'_L$$

In our case, we can make syntactic restrictions that ensure that most general unifiers are *always* defined, which means that the logic is complete even without the $\not\doteq_L$ rule. We use equality only in two restricted ways, both of which straightforwardly ensure

[3] A more standard terminology is to call these propositions "intuitionistic" rather than "persistent," but this is inappropriate for our setting because the ordered linear logic we are using is also an intuitionistic logic. We will therefore consistently use the word "persistent" to describe propositions that act like normal (intuitionistic) mathematical truth.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma; (\mathsf{link}\, d_1 d_2); \cdot \Rightarrow_{d_1 d_2} [\text{¡}(\mathsf{link}\, d_1 d_2)]}\ \mathbf{init}_{\text{¡}} \quad \dfrac{\Gamma; (\mathsf{a}\, d)\,(\mathsf{a}\, d); \cdot \Rightarrow_d C}{\Gamma; (\mathsf{a}\, d_1)\,(\mathsf{a}\, d_2); d_1 \doteq d_2 \Rightarrow_{d_1 d_2} C}\ \doteq'_L}{\Gamma; (\mathsf{a}\, d_1)\,(\mathsf{link}\, d_1 d_2)\,(\mathsf{a}\, d_2); [\text{¡}(\mathsf{link}\, d_1 d_2) \twoheadrightarrow d_1 \doteq d_2] \Rightarrow_{d_1 d_2} C}\ \twoheadrightarrow_L}{\Gamma; (\mathsf{a}\, d_1)\,(\mathsf{link}\, d_1 d_2)\,(\mathsf{a}\, d_2); [\forall D'.\, \text{¡}(\mathsf{link}\, d_1\, D') \twoheadrightarrow d_1 \doteq D'] \Rightarrow_{d_1 d_2} C}\ \forall_L}{\Gamma; (\mathsf{a}\, d_1)\,(\mathsf{link}\, d_1 d_2)\,(\mathsf{a}\, d_2); [\forall D.\, \forall D'.\, \text{¡}(\mathsf{link}\, D\, D') \twoheadrightarrow D \doteq D'] \Rightarrow_{d_1 d_2} C}\ \forall_L}{\Gamma; (\mathsf{a}\, d_1)\,(\mathsf{link}\, d_1 d_2)\,(\mathsf{a}\, d_2); \cdot \Rightarrow_{d_1 d_2} C}\ \mathbf{focus}_L$$

Fig. 5 Sequential derivation representing the transition from $(\mathsf{a}\, d_1)\,(\mathsf{link}\, d_1 d_2)\,(\mathsf{a}\, d_2)$ to $(\mathsf{a}\, d)\,(\mathsf{a}\, d)$ using the rule $(\text{¡}(\mathsf{link}\, D\, D') \twoheadrightarrow D \doteq D')$ – the implicit quantification of D and D' is made explicit in the derivation.

the existence of most general unifiers. One situation is for *definitions* like $(\exists x.\, x \doteq t)$ in which x was introduced locally and so must be a parameter. The other is in situations where we are equating two destinations D and D' that are syntactically known to *always* parameters (recall the introduction where we said that there were no constants or constructors for destinations). Two parameters in Σ are always trivially unifiable.

One reason we want this notion of equality is to express the unification of distinct parameters. Consider that we have a linear context $(\mathsf{a}\, d_1)\,(\mathsf{link}\, d_1\, d_2)\,(\mathsf{a}\, d_2)$ containing three linear atomic propositions, where d_1 and d_2 are distinct parameters. We can apply a rule $(\text{¡}(\mathsf{link}\, D\, D') \twoheadrightarrow D \doteq D')$ to remove the link and unify the parameters d_1 and d_2, resulting in the context $(\mathsf{a}\, d)\,(\mathsf{a}\, d)$. This transition is represented by the derivation in Figure 5. In order to fit the derivation horizontally on the page, we omitted the first premise of $\mathbf{focus}_L$ (which is $(\forall D.\, \forall D'.\, \text{¡}(\mathsf{link}\, D\, D') \twoheadrightarrow D \doteq D') \in \Gamma$) and the first premise of $\doteq'_L$ (which is $(d \vdash (d/d_1, d/d_2) : d_1 d_2) = \mathrm{mgu}(d_1, d_2)$).

A discussion of the standard metatheoretic results for this framework is outside the scope of this article but treated elsewhere [47]. Proposition 1 presents some of these metatheoretic results.

Proposition 1 (Metatheory) *Given the complete definition of non-focused ordered linear logic with equality in [47], the following hold:*

1. *Cut^+ – $\Gamma; \Delta; \Omega \Rightarrow_\Sigma [S]$ and $\Gamma; \Delta'; \Omega_L\, S\, \Omega_R \Rightarrow_\Sigma C$ imply $\Gamma; \Delta\, \Delta'; \Omega_L\, \Omega\, \Omega_R \Rightarrow_\Sigma C$.*
2. *Cut^- – $\Gamma; \Delta; \Omega \Rightarrow_\Sigma A$ and $\Gamma; \Delta'; \Omega_L[A]\Omega_R \Rightarrow_\Sigma C$ imply $\Gamma; \Delta\, \Delta'; \Omega_L\, \Omega\, \Omega_R \Rightarrow_\Sigma C$.*
3. *$Identity^+$ – $\Gamma; \cdot; S \Rightarrow_\Sigma S$.*
4. *$Identity^-$ – $\Gamma, A; \cdot; \cdot \Rightarrow_\Sigma A$.*
5. *Soundness & completeness of focusing – $\Gamma; \Delta; \Omega \Rightarrow_\Sigma C$ if and only if $\Gamma; \Delta; \Omega \vdash_\Sigma C$, where $\Gamma; \Delta; \Omega \vdash_\Sigma C$ is provability in "normal" (i.e. unfocused) ordered linear logic.*

2.2 An environment semantics for SSOS specifications

Persistent atomic propositions and existential quantification in our framework allow us to give an alternate specification of the call-by-value lambda calculus presented in Section 1.2. In rule $\mathsf{e/app_2}$ from our original SSOS specification, when a value was ready to be applied to a function we substituted the value into the function:

$$\mathsf{comp}(\mathsf{app}_2(\mathsf{lam}(\lambda x.E_0\, x))) \bullet \mathsf{retn}(V_2) \twoheadrightarrow \mathsf{eval}(E_0\, V_2) \qquad (\mathsf{e/app_2})$$

$$\mathsf{eval}(X) \bullet \mathsf{!bind}\, X\, V \;\twoheadrightarrow\; \mathsf{retn}(V) \qquad (\mathsf{env/var})$$
$$\mathsf{eval}(\mathsf{lam}(\lambda x.E_0\, x)) \;\twoheadrightarrow\; \mathsf{retn}(\mathsf{lam}(\lambda x.E_0\, x)) \qquad (\mathsf{env/lam})$$
$$\mathsf{eval}(\mathsf{app}\; E_1\; E_2) \;\twoheadrightarrow\; \mathsf{comp}(\mathsf{app}_1\; E_2) \bullet \mathsf{eval}(E_1) \qquad (\mathsf{env/app})$$
$$\mathsf{comp}(\mathsf{app}_1\; E_2) \bullet \mathsf{retn}(V_1) \;\twoheadrightarrow\; \mathsf{comp}(\mathsf{app}_2\; V_1) \bullet \mathsf{eval}(E_2) \qquad (\mathsf{env/app}_1)$$
$$\mathsf{comp}(\mathsf{app}_2(\mathsf{lam}(\lambda x.E_0\, x))) \bullet \mathsf{retn}(V_2) \;\twoheadrightarrow\; \exists y.\, \mathsf{comp}(\mathsf{call}) \bullet \mathsf{eval}(E_0\, y) \bullet \mathsf{!bind}\, y\, V_2 \qquad (\mathsf{env/app}_2)$$
$$\mathsf{comp}(\mathsf{call}) \bullet \mathsf{retn}(V) \;\twoheadrightarrow\; \mathsf{retn}(V) \qquad (\mathsf{env/call})$$

Fig. 6 An SSOS environment semantics for the call-by-value lambda calculus.

Using existential quantification we can instead generate a new parameter y, substitute *that* for the bound variable in E_0, and then generate a persistent fact $\mathsf{!bind}\, y\, V_2$ that permanently associates the parameter with the argument V_2.

$$\mathsf{comp}(\mathsf{app}_2(\mathsf{lam}(\lambda x.E_0\, x))) \bullet \mathsf{retn}(V_2) \;\twoheadrightarrow\; \exists y.\mathsf{eval}(E_0\, y) \bullet \mathsf{!bind}\, y\, V_2 \qquad (\mathsf{env/app}_2{}')$$

A second rule (env/var) 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 persistent proposition $\mathsf{!bind}\, X\, V$; our use of existential quantification ensures that there is at most one such atomic proposition for every parameter.

The complete specification, shown in Figure 6, has one other change – we introduce a stack frame comp(call) in rule $\mathsf{env/app}_2$. Because call has no arguments (unlike app_1 and app_2) this rule is not operationally meaningful. The only effect this rule has is to leave a token on the control stack that says "a function call started here," and that token is later consumed by rule env/call when the function returns a value. We introduce this additional stack frame and rule here because marking the point where a function returns is critical to the control flow analysis we derive later on.

This style of specification was called an "environment semantics" for SSOS specifications in previous work – the persistent context acts as a global environment containing the values of all function arguments. Many other extensions are possible and are discussed in [34], for instance a call-by-need specification that uses linear resources to represent a suspended computation and persistent resources (in the style of this environment semantics) to contain the memoized result of evaluating a suspended computation.

2.3 Transitions in ordered logical specifications

We have defined a sequent calculus for ordered linear logic that is well-behaved (at least under the condition that equality $t \doteq s$ only arises in situations where t and s have a most general unifier). In this section, we discuss a few other conditions that will allow us to treat closed negative propositions as something like rewriting instructions. We have been calling closed negative propositions *rules*; we call collections of rules *specifications* if each rule $(\forall x_1 \ldots \forall x_n.\, S_1 \twoheadrightarrow S_2)$ obeys the following two conditions:

– *Range restriction.* Each universally bound variable x_i, as well as each existentially bound variable in S_1, must have one *strict occurrence* in the premise S_1 [33]. This ensures that higher-order matching is unitary and decidable so that we can always decide whether a particular rule may be applied [41].

– *Rule separation:* A rule with ordered atomic propositions in the conclusion S_2 must have at least one ordered atomic proposition in the premise S_1, and a rule with linear atomic propositions in the conclusion must have at least one ordered or linear atomic proposition in the premise.

Range restriction is necessary in order for logical specifications to have an operational interpretation as forward-chaining logic programs (and rule separation is helpful [45]). Rule separation is a necessary precondition for the translation of ordered logical specifications into linear logical specifications that we consider in the next section.

The operational interpretation of ordered logical algorithms is derived from the focused sequent calculus. If we look at the derivations given in Figures 3 and 5, they have a particular form: a rule is copied into the context, and focused rules are applied as far as possible, at which point unfocused rules are applied as far as possible. A fully focused sequent calculus would enforce both of these steps, but the sequent calculus presented in Figure 4 is only *weakly focused*: it forces focused rules, but not unfocused rules, to be applied as far as possible. The transition semantics that we define presently will enforce the eager application of unfocused left rules as well.

Definition 1 A *state* is an unfocused sequent $(\Gamma_{\mathcal{P}}\Gamma; \Delta; \Omega \Rightarrow_\Sigma S)$ where the contexts Γ, Δ, and Ω contain only (persistent, linear, and ordered, respectively) atomic propositions, $\Gamma_{\mathcal{P}}$ is a specification, and the conclusion S is a closed positive proposition. We use $\mathbb{S}$ as a metavariable for states.

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

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

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.

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 there is a sequential derivation of the following form:

$$\begin{array}{c} \Gamma_{\mathcal{P}}\Gamma'; \Delta'; \Omega' \Rightarrow_{\Sigma'} S \\ \mathcal{D} \\ \Gamma_{\mathcal{P}}\Gamma; \Delta; \Omega \Rightarrow_\Sigma S \end{array}$$

A sequential derivation (a *phase* in focusing terminology) is an application of $\mathbf{focus}_L$ followed by a series of focused rules ($\mathbf{init}$, $\mathbf{init}_¡$, $\mathbf{init}!$, $\twoheadrightarrow_L$, $\bullet_R$, $\mathbf{1}_R$, $\forall_L$, $\exists_R$, and $\doteq_R$) which are then followed by a series of unfocused left rules ($\bullet_L$, $\mathbf{1}_L$, $!_L$, $¡_L$, $\exists_L$, and $\doteq'_L$), leaving exactly one unproven sequent at the "top" that is also a state. We additionally require that every unfocused left rule in a sequential derivation be applied to the *leftmost* non-atomic proposition in Ω.

According to the rules in Figure 4, *every* focusing phase has exactly one leaf. This would not be the case if we introduced a broader family of connectives (such as additive disjunction) or if we relaxed our restrictions on the use of equality. It is, in fact, the need for focusing phases to generally be sequential derivations that largely determined the fragment of ordered logic used for our framework.

2.3.1 Adequacy of transitions

We have now defined $\mathbb{S} \xrightarrow{-+} \mathbb{S}'$, a logically-defined notion of transition. We also have a notion of transition in our parenthesis-checking push-down automaton and a notion of transition in our SSOS specification that we used to give the example trace in Figure 1. We say that a specification *adequately represents* a transition system (such as a PDA or a programming language) if

1. There is a state $\mathbb{S}_X = (\Gamma; \Delta; \Omega \Rightarrow_\Sigma S)$ in the framework for every state X in the transition system,
2. $X = Y$ if and only if $\mathbb{S}_X = \mathbb{S}_Y$,
3. $\mathbb{S}_X \xrightarrow{-+} \mathbb{S}'$ if and only if $X \rightsquigarrow Y$ in the transition system and $\mathbb{S}' = \mathbb{S}_Y$, and
4. If the transition system has a notion of halting state, then X halts if and only if $\mathbb{S}_X \rightarrow|$.

The first three conditions can be satisfied for the push-down automaton if we fix any S and let a PDA state like $(\mathsf{hd}\,{<}\,{>})$ be represented by the state $(\Gamma_\mathcal{P}; \cdot; \mathsf{hd}\,{<}\,{>} \Rightarrow S)$. If we want to represent the "accepting" behavior of the PDA, we can let $S = \mathsf{hd}$, because obviously $(\Gamma_\mathcal{P}; \cdot; \mathsf{hd} \Rightarrow \mathsf{hd}) \rightarrow|$. We are not always concerned about capturing the termination behavior of a system; often we are satisfied to capture just its transitions. In the PDA example we could set $S = \mathbf{1}$ even though, under that definition, there is *no* PDA state X where $\mathbb{S}_X \xrightarrow{-+} \ldots \xrightarrow{-+} \mathbb{S} \rightarrow|$. This definition still adequately captures the transition behavior, though not the termination behavior, of the PDA.

We say that a rule A gives rise to the transition $\mathbb{S} \xrightarrow{-+} \mathbb{S}'$ if we can build a sequential derivation from $\mathbb{S}'$ to $\mathbb{S}$ by focusing on A; verifying the adequacy of transitions is a matter of examining the different sequential derivations that the specification gives rise to. Adequacy then allows us to think of two rules as equivalent if they give rise to the same sequential derivations. For instance, the fact that the linear contexts $(\Delta\, Q_1\, Q_2)$ and $(\Delta\, Q_2\, Q_1)$ are equivalent is the justification for saying that $(¡Q_1 \bullet ¡Q_2 \twoheadrightarrow S)$ and $(¡Q_2 \bullet ¡Q_1 \twoheadrightarrow S)$ are equivalent rules. We will frequently leverage this notion when dealing with conclusions about equality: we can say that the rule $(\mathsf{a}(X) \twoheadrightarrow \exists z.\, \mathsf{b}(z) \bullet X \doteq z)$ is equivalent to a rule $(\mathsf{a}(X) \twoheadrightarrow \mathsf{b}(X))$ because both rules give rise to the same sequential derivations.

2.3.2 Logical correctness of transitions

The connection between the rules of weakly focused ordered linear logic and the transitions in our framework of ordered logical specifications is established by Theorem 1. Proposition 1 together with Theorem 1 then establishes the connection between (non-focused) ordered linear logic and the framework of ordered logical specifications.

Definition 2 A sequence of states $\mathbb{S}_1 \xrightarrow{-+} \ldots \xrightarrow{-+} \mathbb{S}_n$ is called a *trace* (or a *partial trace*; if $\mathbb{S}_n \rightarrow|$ it is called a *complete trace*.

Theorem 1 (Nondeterministic completeness) *If $(\Gamma_\mathcal{P}\Gamma; \Delta; \Omega \Rightarrow_\Sigma S)$ is a state, then there is a derivation of $(\Gamma_\mathcal{P}\Gamma; \Delta; \Omega \Rightarrow_\Sigma S)$ if and only if there exists a complete trace $(\Gamma_\mathcal{P}\Gamma; \Delta; \Omega \Rightarrow_\Sigma S) = \mathbb{S}_0 \xrightarrow{-+} \mathbb{S}_1 \xrightarrow{-+} \ldots \xrightarrow{-+} \mathbb{S}_n \rightarrow|$.*

Proof The reverse direction, that given a series of transitions there exists a derivation, is immediate from the fact that the steps-to relation $\mathbb{S} \xrightarrow{-+} \mathbb{S}'$ was defined according to the

proof rules of weakly-focused ordered linear logic given in Figure 4. The complication of the forward direction comes because the proof rules in Figure 4 do not require that non-focused left rules be applied exhaustively, much less in a left-to-right order.

In order to establish the forward direction, we first need a lemma that all the unfocused left rules are invertible. That is, given a derivation of $(\Gamma_{\mathcal{P}}\Gamma; \Delta; \Omega_L S \Omega_R \Rightarrow_\Sigma S')$,

1. if $S = S_1 \bullet S_2$, then there is a smaller derivation ending in $\Gamma; \Delta; \Omega_L S_1 S_2 \Omega_R \Rightarrow_\Sigma S'$,
2. if $S = ¡Q$, then there is a smaller derivation ending in $\Gamma; \Delta Q; \Omega_L \Omega_R \Rightarrow_\Sigma S'$,
3. if $S = !Q$, then there is a smaller derivation ending in $\Gamma Q; \Delta; \Omega_L \Omega_R \Rightarrow_\Sigma S'$,
4. if $S = \exists x.S$, then there is a smaller derivation ending in $\Gamma; \Delta; \Omega_L S[a/x] \Omega_R \Rightarrow_{\Sigma a} S'$,
5. if $S = \mathbf{1}$, then there is a smaller derivation ending in $\Gamma; \Delta; \Omega_L \Omega_R \Rightarrow_\Sigma S'$, and
6. if $S = (t \doteq s)$ and $\Sigma' \vdash \theta : \Sigma$ is the most general unifier of t and s, then there is a smaller derivation ending in $\theta\Gamma; \theta\Delta; \theta\Omega_L \theta\Omega_R \Rightarrow_{\Sigma'} \theta S'$

Each of the above statements depends on a general lemma that, for any non-atomic S, there is no derivation of $(\Gamma; \Delta; \Omega_L S \Omega_R \Rightarrow_\Sigma [S'])$, and the proof of each of the above statements is also mutually inductive with the proof of an analogous statement for left-focused sequents. The "smaller" part is critical, as in the proof below we induct over the size of derivations and pass the result of the invertibility lemma to the induction hypothesis.

To prove the forward direction of Theorem 1, we are given a derivation of some state $\mathbb{S} = (\Gamma_{\mathcal{P}}\Gamma; \Delta; \Omega \Rightarrow_\Sigma S)$ and we must show that $\mathbb{S} \xrightarrow{-+} \ldots \xrightarrow{-+} \mathbb{S}_n \rightarrow|$. If the last rule in the derivation is $\mathbf{focus}_R$, then $\mathbb{S} \rightarrow|$ and we are done. Otherwise, the last rule can only be $\mathbf{focus}_L$. It will suffice to construct a smaller derivation of $\mathbb{S}' = (\Gamma_{\mathcal{P}}\Gamma'; \Delta'; \Omega' \Rightarrow_{\Sigma'} S)$ and a sequential derivation from $\mathbb{S}'$ to $\mathbb{S}$; if we can do this, then by the induction hypothesis $\mathbb{S}' \xrightarrow{-+} \ldots \xrightarrow{-+} \mathbb{S}_n \rightarrow|$ and by the existence of a sequential derivation $\mathbb{S} \xrightarrow{-+} \mathbb{S}'$.

We have assumed the last rule in the derivation of $\mathbb{S}$ was $\mathbf{focus}_L$, so the derivation of $\mathbb{S}$ consists of zero or more instances of the $\forall_L$ rule followed by the rule $\twoheadrightarrow_L$, which has as its second premise a sub-derivation of $(\Gamma_{\mathcal{P}}\Gamma; \Delta''; \Omega_L S'' \Omega_R \Rightarrow_\Sigma S)$. Using the invertibility lemma, we can then break down the left-most non-atomic proposition in the context until there are only atomic propositions in the ordered context. Once we have done so, we have a smaller derivation of $\mathbb{S}'$ and a sequential derivation from $\mathbb{S}'$ to $\mathbb{S}$, so we are done. □

2.4 Linear and persistent logical specifications

Throughout the paper, we will frequently be interested in the sub-framework of *linear logical specifications* in which specifications contain no ordered atomic propositions as well as *persistent logical specifications* in which specifications contain neither ordered nor linear atomic propositions.

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 a persistent logical specification we will use $\wedge$ instead of $\bullet$, $\supset$ instead of $\twoheadrightarrow$, and Q instead of $!Q$. Similarly, when we are unambiguously talking about a linear logical specification, we will use $\otimes$ instead of $\bullet$, $\multimap$ instead of $\twoheadrightarrow$, and Q instead of $¡Q$. We claim that a similarly-defined transition semantics for a framework of linear or persistent logical specifications would correspond

$$\begin{aligned}
[\![\forall x.A]\!] &= \forall x.[\![A]\!] & [\![S_1 \bullet S_2]\!]^{d_L}_{d_R} &= \exists d_M.[\![S_1]\!]^{d_L}_{d_M} \otimes [\![S_2]\!]^{d_M}_{d_R} \\
[\![S_1 \twoheadrightarrow S_2]\!] &= \forall d_L.\forall d_R.[\![S_1]\!]^{d_L}_{d_R} \multimap [\![S_2]\!]^{d_L}_{d_R} & [\![\exists x.S]\!]^{d_L}_{d_R} &= \exists x.[\![S]\!]^{d_L}_{d_R} \\
[\![S]\!] &= \exists d_L.\exists d_R.[\![S]\!]^{d_L}_{d_R} & [\![\mathbf{1}]\!]^{d_L}_{d_R} &= d_L \doteq d_R \\
[\![Q]\!]^{d_L}_{d_R} &= Q\, d_L\, d_R & [\![t \doteq s]\!]^{d_L}_{d_R} &= t \doteq s \otimes d_L \doteq d_R \\
[\![¡Q]\!]^{d_L}_{d_R} &= Q \otimes d_L \doteq d_R \\
[\![!Q]\!]^{d_L}_{d_R} &= !Q \otimes d_L \doteq d_R
\end{aligned}$$

$$\begin{aligned}
[\![\Gamma_{\mathcal{P}}\Gamma; \Delta; \Omega \Rightarrow_\Sigma S]\!] &= ([\![\Gamma_{\mathcal{P}}]\!]\Gamma; \Delta[\![\Omega]\!]^{d_0}_{d_n}; \cdot \Rightarrow_{\Sigma d_0 \ldots d_n} [\![S]\!]) \\
[\![\Gamma_{\mathcal{P}}\Gamma; \Delta; \Omega \Rightarrow_\Sigma S]\!]^+ &= ([\![\Gamma_{\mathcal{P}}]\!]\Gamma; \Delta[\![\Omega]\!]^{d_0}_{d_n}; \cdot \Rightarrow_{\Sigma \Sigma_w d_0 \ldots d_n} [\![S]\!])
\end{aligned}$$

Fig. 7 Translation of propositions and states from ordered logic into the linear fragment.

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.

3 Translation into linear logic

We just discussed, in Section 2.4, that it is possible to define linear logical specifications in terms of ordered logical specifications. A more interesting result, which we now consider, is that we can faithfully translate ordered logical specifications into linear logical specifications. In this section, we present a "destination-adding translation" from ordered logical specifications to linear logical specifications.

The translation, which was used informally in the introduction but is given explicitly in Figure 7, translates ordered atomic propositions into linear atomic propositions and then adds two arguments to these propositions to make the lost adjacency information explicit – $[\![\mathsf{hd}]\!]^d_{d'} = \mathsf{hd}\, d\, d'$ in our PDA example, and $[\![\mathsf{left}\, X]\!]^d_{d'} = \mathsf{left}\, X\, d\, d'$. We write the translation of a state $\mathbb{S}$ into the linear fragment as $[\![\mathbb{S}]\!]$. Translated specifications may generate spurious extra parameters, so we write $[\![\mathbb{S}]\!]^+$ to describe the translation of a state $\mathbb{S}$ that also includes free parameters that do not appear in $\mathbb{S}$ or $[\![\mathbb{S}]\!]$. 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 $[\![\Omega]\!]^{d_0}_{d_n}$, the value of d_0 is the same as d_n exactly when $(\Omega = \cdot)$.

Similar translations of the Lambek calculus into ordered logic, all of which are to some degree reflections of van Benthem's relational models of ordered logic [5], have been explored previously [30, 29]. However, previous work has only used the propositional Lambek calculus without linear or persistent atomic propositions and without the unit of ordered conjunction $\mathbf{1}$, and the addition of these propositions complicates matters significantly. Without the restrictions we made on propositions, and without the requirement of rule separation, translated specifications would not behave the same way as untranslated specifications. For example, the rule $\mathbf{1} \twoheadrightarrow Q$, which does not obey rule separation, translates as $(\forall d_L.\forall d_R.\, (d_L \doteq d_R) \multimap Q\, d_L\, d_R)$. Therefore, a translated specification 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 new state is not equal to any translated context: there are not 4 distinct parameters for the 3 propositions. Rule separation prevents this problem.

We have now defined everything except for the translation of specifications $[\![\Gamma_{\mathcal{P}}]\!]$. A rule A that mentions ordered atomic propositions needs to be translated to $[\![A]\!]$, but a rule such as $(¡Q_1 \twoheadrightarrow ¡Q_2)$ that does *not* mention ordered atomic propositions can either be translated as $[\![¡Q_1 \twoheadrightarrow ¡Q_2]\!] = (\forall d_L.\forall d_R.\, Q_1 \otimes (d_L \doteq d_R) \multimap Q_2 \otimes (d_L \doteq d_R))$, or else it can be left in the specification unchanged. It simplifies the proof of correctness if we leave a rule like $(¡Q_1 \twoheadrightarrow ¡Q_2)$ in the specification unchanged, so we define the translation of $[\![\Gamma_{\mathcal{P}}]\!]$ to be a new specification where all the rules A mentioning ordered atomic propositions have been replaced by $[\![A]\!]$ and where all the rules not mentioning ordered atomic proposition remain unchanged. This has the added nice property that the translation of specifications is idempotent: $[\![[\![\Gamma_{\mathcal{P}}]\!]]\!] = [\![\Gamma_{\mathcal{P}}]\!]$.

The application of the translation defined in Figure 7 to the PDA specification given by the rules push_2 and pop_2 yields the following specification:

$$(\exists d_m.\, ¡\mathsf{hd}\, D_l\, d_m \;\bullet\; ¡\mathsf{left}\, X\, d_m\, D_r) \;\twoheadrightarrow\; (\exists d'_m.\, ¡\mathsf{stack}\, X\, D_l\, d'_m \;\bullet\; ¡\mathsf{hd}\, d_m\, D_r)$$
$$(\exists d_{m1}.\, ¡\mathsf{stack}\, X\, D_l\, d_{m1} \;\bullet\; (\exists d_{m2}.\, ¡\mathsf{hd}\, d_{m1}\, d_{m2} \;\bullet\; ¡\mathsf{right}\, X\, d_{m2}\, D_r)) \;\twoheadrightarrow\; ¡\mathsf{hd}\, D_l\, D_r$$

This specification is equivalent (in the sense of giving rise to the same sequential derivations) to the rules pop_3 and push_3 given in the introduction.

3.1 Correctness of the translation

Having given the translation of propositions, states, and specifications from the ordered logical framework into the linear logical sub-framework, we can state the correctness criteria. The proof of Theorem 2, which is somewhat tedious, appears in Appendix A.

Theorem 2 (Correctness of translation) *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

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

The critical point to observe about Theorem 2 is that it can *almost* be composed with the criteria of adequacy (for both complete and partial traces) that we discussed in Section 2.3.1: if an ordered logical specification adequately represents some transition system, then the translation of that specification into a linear logical specification *also* adequately represents the transition system. The only thing standing in the way is the fact that we have $\mathbb{S}_l = [\![\mathbb{S}_o]\!]^+$ instead of $\mathbb{S}_l = [\![\mathbb{S}_o]\!]$ because the translation may introduce spurious extra parameters. If we relax our definition of adequacy to ignore parameters that are not present in any of the contexts, however, then what Theorem 2 gives us is precisely that an adequate ordered logical specification of a system is translated into an adequate linear logical specification of the same system.

$$(\exists d_1.\, \mathsf{eval}\ X\ D\ d_1 \otimes !\mathsf{bind}\ X\ V \otimes d_1 \doteq D') \multimap \mathsf{retn}\ V\ D\ D'$$
$$\mathsf{eval}\ (\mathsf{lam}(\lambda x.E\ x))\ D\ D' \multimap \mathsf{retn}\ (\mathsf{lam}(\lambda x.E\ x))\ D\ D'$$
$$\mathsf{eval}\ (\mathsf{app}\ E_1\ E_2)\ D\ D' \multimap \exists d_1.\, \mathsf{comp}\ (\mathsf{app}_1\ E_2)\ D\ d_1 \otimes \mathsf{eval}\ E_1\ d_1\ D'$$
$$(\exists d_1.\, \mathsf{comp}\ (\mathsf{app}_1\ E_2)\ D\ d_1 \otimes \mathsf{retn}\ V_1\ d_1\ D') \multimap \exists d_2.\, \mathsf{comp}\ (\mathsf{app}_2\ V_1)\ D\ d_2 \otimes \mathsf{eval}\ E_2\ d_2\ D'$$
$$(\exists d_2.\, \mathsf{comp}\ (\mathsf{app}_2(\mathsf{lam}(\lambda x.E_0\ x)))\ D\ d_2 \otimes \mathsf{retn}\ V_2\ d_2\ D')$$
$$\multimap \exists y.\, \exists d_0.\, \mathsf{comp\ call}\ D\ d_0 \otimes (\exists d.\, \mathsf{eval}\ (E_0\ y)\ d_0\ d \otimes !\mathsf{bind}\ y\ V_2 \otimes d \doteq D')$$
$$(\exists d_0.\, \mathsf{comp\ call}\ D\ d_0 \otimes \mathsf{retn}\ V_0\ d_0\ D') \multimap \mathsf{retn}\ V_0\ D\ D'$$

Fig. 8 Result of translating the environment semantics from Figure 6.

$$\mathsf{eval}\ X\ D\ D' \otimes !\mathsf{bind}\ X\ V \multimap \mathsf{retn}\ V\ D\ D'$$
$$\mathsf{eval}\ (\mathsf{lam}(\lambda x.E\ x))\ D\ D' \multimap \mathsf{retn}\ (\mathsf{lam}(\lambda x.E\ x))\ D\ D'$$
$$\mathsf{eval}\ (\mathsf{app}\ E_1\ E_2)\ D\ D' \multimap \exists d_1.\, \mathsf{comp}\ (\mathsf{app}_1\ E_2)\ D\ d_1 \otimes \mathsf{eval}\ E_1\ d_1\ D'$$
$$\mathsf{comp}\ (\mathsf{app}_1\ E_2)\ D\ D_1 \otimes \mathsf{retn}\ V_1\ D_1\ D' \multimap \exists d_2.\, \mathsf{comp}\ (\mathsf{app}_2\ V_1)\ D\ d_2 \otimes \mathsf{eval}\ E_2\ d_2\ D'$$
$$\mathsf{comp}\ (\mathsf{app}_2(\mathsf{lam}(\lambda x.E_0\ x)))\ D\ D_2 \otimes \mathsf{retn}\ V_2\ D_2\ D'$$
$$\multimap \exists y.\, \exists d_0.\, \mathsf{comp\ call}\ D\ d_0 \otimes \mathsf{eval}\ (E_0\ y)\ d_0\ D' \otimes !\mathsf{bind}\ y\ V_2$$
$$\mathsf{comp\ call}\ D\ D_0 \otimes \mathsf{retn}\ V_0\ D_0\ D' \multimap \mathsf{retn}\ V_0\ D\ D'$$

Fig. 9 Simplified specification equivalent to the specification in Figure 8.

$$\mathsf{eval}\ X\ D \otimes !\mathsf{bind}\ X\ V \multimap \mathsf{retn}\ V\ D \qquad (\mathsf{d/var})$$
$$\mathsf{eval}\ (\mathsf{lam}(\lambda x.E\ x))\ D \multimap \mathsf{retn}\ (\mathsf{lam}(\lambda x.E\ x))\ D \qquad (\mathsf{d/lam})$$
$$\mathsf{eval}\ (\mathsf{app}\ E_1\ E_2)\ D \multimap \exists d_1.\, \mathsf{comp}\ (\mathsf{app}_1\ E_2)\ D\ d_1 \otimes \mathsf{eval}\ E_1\ d_1 \qquad (\mathsf{d/app})$$
$$\mathsf{comp}\ (\mathsf{app}_1\ E_2)\ D\ D_1 \otimes \mathsf{retn}\ V_1\ D_1 \multimap \exists d_2.\, \mathsf{comp}\ (\mathsf{app}_2\ V_1)\ D\ d_2 \otimes \mathsf{eval}\ E_2\ d_2 \qquad (\mathsf{d/app1})$$
$$\mathsf{comp}\ (\mathsf{app}_2(\mathsf{lam}(\lambda x.E_0\ x)))\ D\ D_2 \otimes \mathsf{retn}\ V_2\ D_2 \qquad (\mathsf{d/app2})$$
$$\multimap \exists y.\, \exists d_0.\, \mathsf{comp\ call}\ D\ d_0 \otimes \mathsf{eval}\ (E_0\ y)\ d_0 \otimes !\mathsf{bind}\ y\ V_2$$
$$\mathsf{comp\ call}\ D\ D_0 \otimes \mathsf{retn}\ V_0\ D_0 \multimap \mathsf{retn}\ V_0\ D \qquad (\mathsf{d/call})$$

Fig. 10 Modification of Figure 9 with the vestigial D' argument removed from eval and retn.

3.2 Linear destination-passing style

We will conclude by returning to the call-by-value lambda calculus example in order to make an observation about the result of passing it (and other SSOS specifications) through the translation we have described. If we translate the environment semantics given in Figure 6, the result is the specification in Figure 8, which is somewhat more complicated than necessary. As it happened before with the translation of the PDA specification, whenever we translate a rule we always want to eliminate equalities like $d \doteq D'$ by replacing d with D'. We also always want to turn variables that are existentially quantified in a premise into variables that are implicitly universally quantified over the whole rule. The resulting specification, which in this case is shown in Figure 9, is always equivalent (again, in the sense of giving rise to the same sequential derivations) to the original specification. In the future, when we translate ordered log-

ical specifications we will just skip the step shown in Figure 8 and go straight to the version shown in Figure 9.

In the case of the specification in Figure 9, one additional simplification can be made. The last argument to $\mathsf{eval}\,E\,D\,D'$ in Figure 9 is not operationally significant – if we start out with a linear atomic proposition $\mathsf{eval}\,E\,d_0\,d_1$, every eval and retn proposition will have that same parameter d_1 as its last argument, as the parameter 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. By removing the vestigial D' parameter, we can rewrite this specification to obtain the specification in Figure 10. This specification 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. (This analogy is why the translation to linear logic we have defined is referred to as a *destination-adding* translation.) Linear destination-passing style was the original form of substructural operational semantics specifications before ordered logic was considered as a framework [11,31].

The fact that linear destination-passing style arises naturally from the destination-adding translation is a new observation, and is interesting in its own right.[4] For the purposes of our current discussion, the translation 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.

4 Approximation as a logic program

In this section, we describe an approximation strategy in which we can approximate ordered and linear logical specifications as persistent logical specifications and then interpret these persistent logical specifications as logic programs. We have already shown how a logical specification can be interpreted as a state transition system; such a transition system can also be naturally given an operational interpretation as a *forward-chaining or "bottom-up" logic programming language.*

For ordered and linear logical specifications, the most obvious forward-chaining logic programming interpretation is based on *committed choice* (at each step, the interpreter arbitrarily picks one transition and does not reconsider that choice) and *quiescence* (the interpreter stops when there are no more transitions possible). This style of giving a forward-chaining operational semantics to substructural logic specifications

[4] One way to interpret this formal relationship between ordered SSOS specifications and linear SSOS specifications using destination-passing style is to think of *linear* specifications as the fundamental specifications and treat the 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 [34]. 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.

has a long history [10, 12, 20, 26, 34, 45], and there is an implementation of a committed-choice logic programming language based on our ordered logical framework.[5] However, committed choice and quiescence are bad foundations for a logic programming interpretation of *persistent* logical specifications. The use of a persistent proposition in the premise of a rule does not remove that persistent proposition from the context when that rule is applied, so if we make a given transition once, we can make it again, deriving a new redundant copy of all the facts in the conclusion. This means that quiescence is a bad criteria for termination, as whenever any transitions are possible an infinite sequence of transitions is possible. It also means that transitions do not imply any sort of commitment, so committed choice is not a meaningful concept.

A forward-chaining semantics that instead makes transitions only to derive *new* persistent atomic propositions until no new persistent atomic propositions can be derived is said to be based on *saturation* as opposed to quiescence. This forward-chaining logic programming interpretation of persistent logic is extremely common; in fact it is what is commonly meant by "forward-chaining logic programming." Just as we introduced the term *persistent logic* to distinguish what is classically referred to as intuitionistic logic from intuitionistic ordered and linear logic, we will introduce the term *saturating logic programming* to distinguish what is classically referred to as forward-chaining logic programming from the forward-chaining logic programming interpretation that makes sense for ordered and linear logical specifications.

We do not wish to consider the details of implementing saturating logic programs here, though we will touch on the topic in the conclusion. However, we will be concerned in this section with the termination of saturating logic programs. We can reason about termination in terms of an idealized interpreter that only allows transitions which derive new facts or equalities ($t \doteq s$) that were not immediately provable before the transition. Any saturating logic program that can derive only finitely many distinct facts from any finite initial state will then necessarily terminate in this idealized interpreter. As discussed in the introduction, a specification with a rule like $(\mathsf{a} \supset \exists x.\, \mathsf{b}(x))$ will not terminate when interpreted as a saturating logic program because we can always productively apply the rule to create a new parameter along with a new fact containing that parameter.

4.1 Approximation and the meta-approximation theorem

Our approximation strategy is simple: a rule $(\forall x_1 \ldots \forall x_n.\, S_1 \twoheadrightarrow S_2)$ in an ordered or linear logical specification can be approximated by making all atomic propositions persistent, removing premises from S_1, and adding conclusions to S_2. Of particular practical importance are added conclusions that equate the parameters introduced by existential quantification with terms: all parameters introduced by existential quantification must be dealt with as a necessary condition for interpreting a persistent specification as a saturating logic program.

First, we define what it means for a specification to be an approximate version of another specification.

Definition 3 A specification Γ_a is an *approximate version* of another specification $\Gamma_\mathcal{P}$ if Γ_a is in the persistent fragment, and if, for every rule $(\forall x_1 \ldots \forall x_n.\, S_1 \twoheadrightarrow S_2)$ in $\Gamma_\mathcal{P}$ there is a corresponding rule in $(\forall x_1 \ldots \forall x_n.\, S_1' \twoheadrightarrow S_2')$ in Γ_a such that:

[5] `http://ollibot.hyperkind.org/`

1. The existential parameters in S_1 and S_2 are identical to the existential parameters in S_1' and S_2' (respectively),
2. For each premise $!Q$ in S_1', there is a premise Q, ¡Q, or $!Q$ in S_1 (and similarly for premises of the form $t \doteq s$), and
3. For each conclusion Q, ¡Q, or $!Q$ in S_2, there is a conclusion $!Q$ in S_2' (and similarly for conclusions of the form $t \doteq s$).

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. We use $S!$ to represent a straightforward operation of turning ordered and linear atomic propositions into persistent ones: if $S = Q_1 \bullet ¡Q_2 \bullet !Q_3$, then $S! = !Q_1 \bullet !Q_2 \bullet !Q_3$.

Definition 4 If Γ_a is an approximate version of $\Gamma_\mathcal{P}$, then the state $(\Gamma_a\Gamma_g; \cdot; \cdot \Rightarrow_{\Sigma_g} S!)$ is a *generalization* of the state $(\Gamma_\mathcal{P}\Gamma; \Delta; \Omega \Rightarrow_\Sigma S)$ if there is a substitution $\Sigma_g \vdash \theta : \Sigma$ such that, for all ordered, linear, and persistent atomic propositions $Q \in \Gamma, \Delta, \Omega$, there exists a persistent proposition $Q_g \in \Gamma_g$ such that $\theta Q = Q_g$.

Since generalizations always have the form $(\Gamma_a\Gamma_g; \cdot; \cdot \Rightarrow_{\Sigma_g} S!)$, we will just write $(\Gamma_a\Gamma_g \Rightarrow_{\Sigma_g} S!)$ for the sake of brevity. One thing we might prove about the relationship between states and their generalizations is that, if $\mathbb{S}_g$ is a generalization of $\mathbb{S}$ and if there is a complete trace $\mathbb{S} \xrightarrow{-+} \ldots \xrightarrow{-+} \mathbb{S}' \rightarrow|$ then there is a complete trace $\mathbb{S}_g \xrightarrow{-+} \ldots \xrightarrow{-+} \mathbb{S}_g' \rightarrow|$. This is a corollary of Lemma 7 (Simulation) in Appendix B. The idea that we actually want to capture is quite a bit stronger, and is expressed by the following definition:

Definition 5 A state $\mathbb{S}_a$ is an *abstraction* of $\mathbb{S}_0$ if, for any trace $\mathbb{S}_0 \xrightarrow{-+} \ldots \xrightarrow{-+} \mathbb{S}'$, $\mathbb{S}_a$ is a generalization of $\mathbb{S}'$.

An abstraction of a state $\mathbb{S}_0$ is therefore a single state that essentially captures *all possible future behaviors* of the state $\mathbb{S}_0$ because, for any atomic proposition Q that may be derived by applying transitions to $\mathbb{S}_0$, there is a substitution θ such that θQ is already present in the abstraction. The meta-approximation theorem relates this definition of abstraction to the concept of approximate versions of programs as specified by Definition 3.

Theorem 3 (Meta-approximation) *If Γ_a is an approximate version of $\Gamma_\mathcal{P}$, there is a state $\mathbb{S}_0 = (\Gamma_\mathcal{P}\Gamma_0; \Delta_0; \Omega_0 \Rightarrow_{\Sigma_0} S)$, and for some $\Sigma_0' \vdash \theta : \Sigma_0$ there is a trace $(\Gamma_a(\theta\Gamma_0)(\theta\Delta_0)(\theta\Omega_0) \Rightarrow_{\Sigma_0'} S!) \xrightarrow{-+} \ldots \xrightarrow{-+} \mathbb{S}_a$ such that $\mathbb{S}_a$ is saturated, then $\mathbb{S}_a$ is an abstraction of $\mathbb{S}_0$.*

The meaning of the meta-approximation theorem is that if (1) we can approximate a specification and an initial state, and (2) we can obtain a saturated state from that approximate specification and approximate initial state, then the saturated state captures all possible future behaviors of the (non-approximate) initial state. The proof is given in Appendix B.

4.2 Termination and Skolemization

The meta-approximation theorem guarantees that we can generate an abstraction of a program if the approximate specification can be interpreted as a terminating logic

program; therefore, we are interested in approximating specifications in such a way that the approximate specifications are terminating when interpreted as saturating logic programs.[6]

Recall the non-terminating approximate PDA specification that we considered in the introduction. The rule push_4 was the one that caused trouble with regards to termination:

$$\mathsf{hd}\,L\,M \;\wedge\; \mathsf{left}\,X\,M\,R \;\supset\; \exists m.\; \mathsf{stack}\,X\,L\,m \;\wedge\; \mathsf{hd}\,m\,R \qquad (\mathsf{push}_4)$$

In the introduction, we considered equating m with both L and M. But if we are going to equate m with *anything*, the most general starting point is to apply Skolemization to the rule. By moving the existential quantifier for m in front of the implicitly quantified X, L, M, and R, we get a resulting Skolem function $(\mathsf{fm}\,X\,L\,M\,R)$ that takes four arguments.

$$\mathsf{hd}\,L\,M \;\wedge\; \mathsf{left}\,X\,M\,R \;\supset\; \mathsf{stack}\,X\,L\,(\mathsf{fm}\,X\,L\,M\,R) \;\wedge\; \mathsf{hd}\,(\mathsf{fm}\,X\,L\,M\,R)\,R \quad (\mathsf{push}_{4\mathsf{sk}})$$

The rule $\mathsf{push}_{4\mathsf{sk}}$ is not actually an approximate version of the rule push_4 according to Definition 3, but the equivalent rule $\mathsf{push}_{4\mathsf{sk}'}$ is:

$$\mathsf{hd}\,L\,M \;\wedge\; \mathsf{left}\,X\,M\,R \;\supset\; \exists m.\; \mathsf{stack}\,X\,L\,m \;\wedge\; \mathsf{hd}\,m\,R \;\wedge\; m \doteq (\mathsf{fm}\,X\,L\,M\,R) \qquad (\mathsf{push}_{4\mathsf{sk}'})$$

Skolem functions therefore provide a natural starting point for approximations, even though the Skolem constant that arises directly from Skolemization is usually more precise than we want. From this starting point, we can define approximations simply by approximating the Skolem function. The two approximations of push_4 that we considered in the introduction can actually be viewed as particular *definitions* of the Skolem function fm. The approximation equating m and M is what results if we define fm to be $(\lambda X.\,\lambda L.\,\lambda M.\,\lambda R.\,M)$, and the approximation equating m and L is what results if we define fm to be $(\lambda X.\,\lambda L.\,\lambda M.\,\lambda R.\,L)$.

Skolemization is a general strategy for equating all parameters introduced by existential quantification with terms; this is a necessary but not a sufficient condition for the termination of a saturating logic program. A sufficient condition, as we discussed above, is a finite bound on the number of derivable propositions. In the PDA approximation from the introduction, when we equated the parameter m in rule push_4 with either L or M, it ensured that we could derive, at most, one fact $(\mathsf{left}\,X\,M\,R)$ for every token X and destination M and R in the initial program, and the same goes for $(\mathsf{right}\,X\,M\,R)$, $(\mathsf{stack}\,X\,M\,R)$, and $(\mathsf{hd}\,M\,R)$. Therefore, if we start with an initial state containing n tokens and m destinations, we can derive no more than $3nm^2 + m^2$ distinct facts. With a little more work we could give a much better bound, but that does not matter for our current purposes. The bound we gave is *finite*, which means that if we only consider transitions that derive new facts, we will reach a state that is saturated in only a finite number of steps.

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

Fig. 11 First-cut approximation of Figure 10; forgetting linearity.

4.3 A control flow analysis from an SSOS specification

Figure 11 is just the linear destination-passing style SSOS specification from Figure 10 converted into a persistent logical specification. In order for us to approximate this program to derive the finite control flow analysis in Figure 12, our first step is to equate the parameter y that we introduced as part of the environment semantics with $\mathsf{var}(\lambda x.\, E_0\ x)$. The constructor var is a greatly simplified Skolem function for y that only mentions the higher-order term $(\lambda x.\, E_0\ x)$ – the most general Skolem function in this setting would have also been dependent on V, D, and D_2. Adding $(y \doteq \mathsf{var}(\lambda x.E_0\ x))$ effectively causes us to associate all parameters ever passed into a function with the *function into which that parameter was passed.*

The pattern above turns out to be a fairly common pattern in the approximation of specifications that use higher-order abstract syntax, because it is a simple way of getting a notion of the "subterms" of a higher-order term. When given a term $(\mathsf{a}\ (\mathsf{b\ c\ c}))$, it is clear that there are three distinct subterms: the entire term, $(\mathsf{b\ c\ c})$, and c. Therefore, it is meaningful to bound the size of a database by some function which depends on the number of subterms of the original term. But what are the subterms of $\mathsf{lam}(\lambda x.\, \mathsf{app}\ x\ x)$? Because we ensure that we *only* substitute terms $\mathsf{var}(\lambda x.\, E\ x)$ into function $\lambda x.\, E\ x$ we can actually answer this question: there are three distinct subterms of $\mathsf{lam}(\lambda x.\, \mathsf{app}\ x\ x)$: the entire term, $(\mathsf{app}\ (\mathsf{var}(\lambda x.\, E\ x))\ (\mathsf{var}(\lambda x.\, E\ x)))$, and $\mathsf{var}(\lambda x.\, E\ x)$. The subterms of any closed term E in our untyped lambda calculus can be enumerated by running this saturating logic program starting with the fact $(\mathsf{subterms}\ E)$:

$$\mathsf{subterms}(\mathsf{lam}(\lambda x.\, E\ x)) \ \supset\ \mathsf{subterms}(E(\mathsf{var}(\lambda x.\, E\ x)))$$
$$\mathsf{subterms}(\mathsf{app}\ E_1\ E_2) \ \supset\ \mathsf{subterms}\ E_1 \wedge \mathsf{subterms}\ E_2$$
$$\mathsf{subterms}(\mathsf{var}(\lambda x.\, E\ x)) \ \supset\ \mathsf{subterms}(E(\mathsf{var}(\lambda x.\, E\ x)))$$

The last rule is redundant: if we ever derive a fact $(\mathsf{subterms}(\mathsf{var}(\lambda x.\, E\ x))$, we know that we previously derived $(\mathsf{subterms}(\mathsf{lam}(\lambda x.\, E\ x)))$ and therefore by the first rule we can already derive $(\mathsf{subterms}(E(\mathsf{var}(\lambda x.\, E\ x))))$.

In order to continue approximating Figure 11 to obtain Figure 12, we need to have in mind the question that we intend to answer with this control flow analysis. The primary question that a flow analysis is intended to answer is, "for any given call site in the

[6] 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.

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

Fig. 12 A control flow analysis derived from Figure 11.

$$\mathsf{eval}\ X \wedge \mathsf{bind}\ X\ V\ \supset\ \mathsf{retn}\ V\ X \qquad (\mathsf{cfa/var})$$
$$\mathsf{eval}\ (\mathsf{lam}(\lambda x.E\,x))\ \supset\ \mathsf{retn}\ (\mathsf{lam}(\lambda x.E\,x))\ (\mathsf{lam}(\lambda x.E\,x)) \qquad (\mathsf{cfa/lam})$$
$$\mathsf{eval}\ (\mathsf{app}\ E_1\ E_2)\ \supset\ \mathsf{comp}\ (\mathsf{app}_1\ E_2)\ (\mathsf{app}\ E_1\ E_2)\ E_1 \wedge \mathsf{eval}\ E_1 \qquad (\mathsf{cfa/app})$$
$$\mathsf{comp}\ (\mathsf{app}_1\ E_2)\ E\ E_1 \wedge \mathsf{retn}\ V_1\ E_1\ \supset\ \mathsf{comp}\ (\mathsf{app}_2\ V_1)\ E\ E_2 \wedge \mathsf{eval}\ E_2 \qquad (\mathsf{cfa/app_1})$$
$$\mathsf{comp}\ (\mathsf{app}_2(\mathsf{lam}(\lambda x.E_0\,x))\ E\ E_2 \wedge \mathsf{retn}\ V_2\ E_2 \qquad (\mathsf{cfa/app_2})$$
$$\supset\ \exists y.\ \mathsf{comp\,call}\ E\ (E_0\ y) \wedge \mathsf{eval}\ (E_0\ y) \wedge \mathsf{bind}\ y\ V_2 \wedge (y \doteq \mathsf{var}(\lambda x.E_0\,x))$$
$$\mathsf{comp\,call}\ E\ E_0 \wedge \mathsf{retn}\ V_0\ E_0\ \supset\ \mathsf{retn}\ V_0\ E \qquad (\mathsf{cfa/call})$$

Fig. 13 A simplified version of Figure 12 that eliminates the now-redundant argument to eval.

source program, what are the functions that may be invoked at that location?"[7] Call sites correspond to expressions of the form $(\mathsf{app}\ E_1\ E_2)$ and functions are expressions of the form $\mathsf{lam}(\lambda x.\ E\,x)$; therefore, our next step is to equate the destinations introduced in the app rules with the expressions we are evaluating at those points. The resulting specification (Figure 12) is terminating because the rules only break expressions E and values V into their "subexpressions" in the sense we have described above. If the expressions in the original state of a program have n subterms, the program can derive no more than n^2 new "eval" facts, n^2 new "retn" facts, and $2n^3+1$ new "comp" facts.

This analysis combined with the meta-approximation theorem ensures that we have derived some sort of program analysis, but to discuss what kind of program analysis it is, it will be helpful to simplify Figure 12 a bit. We have now made the second argument to eval uninteresting – when we derive a new fact of the form $(\mathsf{eval}\,E\,d)$ in Figure 12, we always add a conclusion of the form $(d \doteq E)$, so we might as well drop the second argument to eval because it is the same as the first argument. The result of this simplification is shown in Figure 13. In this figure, we can see that the second argument E to $(\mathsf{comp}\,F\,E\,E')$ is always a term $(\mathsf{app}\,E_1\,E_2)$ – that is, a call site. The rule $\mathsf{cfa/app_2}$ starts evaluating the function $\mathsf{lam}(\lambda x.E_0 x)$ and stores the stack frame $\mathsf{comp\,call}\,E\,(E_0(\mathsf{var}(\lambda x.\,E_0\,x)))$. This means that the function $\mathsf{lam}(\lambda x.\,E_0\,x)$ may be called from call site E only if $(\mathsf{comp\,call}\,E\,(E_0(\mathsf{var}(\lambda x.\,E_0\,x))))$ appears in the saturated database.

[7] This kind of "*may*-" analysis, where the intention is to over-approximate the events that might happen, is the kind of analysis (as opposed to a "*must*-" analysis) that maps easily onto the meta-approximation theorem.

There is one important caveat to this analysis. If for some value V we consider the program $(\mathsf{app}\,(\mathsf{app}\,(\mathsf{lam}(\lambda x.x))\,(\mathsf{lam}(\lambda y.y)))\,V)$, 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, however, $\mathsf{lam}(\lambda x.x)$ and $\mathsf{lam}(\lambda y.y)$ are syntactically identical (names of bound variables do not matter). This is 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 terms. 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 alias analysis presented in Section 5 discusses the use of these labels.

5 Approximating SSOS specifications for alias analysis

So far, we have discussed four stages: ordered logical specifications and substructural operational semantics (Section 2), the destination-adding translation from ordered logical specifications to linear logical specifications (Section 3), the approximation of logical specifications by persistent logical specifications (Section 4.1), and obtaining saturating logic programs which always terminate by equating parameters with their Skolem functions and then approximating the Skolem functions (Section 4.2). We have also worked through these stages to take an SSOS-style environment semantics for the lambda calculus and derive a control flow analysis. In this section, we will take an SSOS specification of a monadic functional language with Lisp-like mutable cons cells and derive an interprocedural alias analysis. 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. 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), pair allocations ($\mathsf{newpair}$), 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

$$\mathsf{eval}\ (\mathsf{let}\ L\ (\mathsf{fun}\ (\lambda x_0.E_0\ x_0))\ (\lambda x.E\ x))\ D \tag{9}$$
$$\supset \exists y.\ \mathsf{eval}\ (E\ y)\ D \wedge \mathsf{bind}\ y\ (\mathsf{lam}(\lambda x_0.E_0\ x_0))) \wedge (y \doteq \mathsf{var}(\lambda x.E\ x))$$
$$\mathsf{eval}\ (\mathsf{let}\ L\ (\mathsf{call}\ F\ X)\ (\lambda x.E\ x))\ D \wedge \mathsf{bind}\ F\ (\mathsf{lam}(\lambda x_0.E_0\ x_0)) \wedge \mathsf{bind}\ X\ V \tag{10}$$
$$\supset \exists y.\ \exists d_0.\ \mathsf{comp}\ (\mathsf{call}_1(\lambda x.E\ x))\ D\ d_0 \wedge \mathsf{eval}\ (E_0\ y)\ d_0 \wedge \mathsf{bind}\ y\ V \wedge (y \doteq d_0 \doteq \mathsf{var}(\lambda x_0.E_0\ x_0))$$
$$\mathsf{comp}\ (\mathsf{call}_1(\lambda x.E\ x))\ D\ D_0 \wedge \mathsf{eval}\ (\mathsf{return}\ L\ X)\ D_0 \wedge \mathsf{bind}\ X\ V \tag{11}$$
$$\supset \exists y.\ \mathsf{eval}\ (E\ y)\ D \wedge \mathsf{bind}\ y\ V \wedge (y \doteq \mathsf{var}(\lambda x.E\ x))$$
$$\mathsf{eval}\ (\mathsf{let}\ L\ \mathsf{newpair}\ (\lambda x.E\ x))\ D \tag{12}$$
$$\supset \exists y.\ \exists d.\ \mathsf{eval}\ (E\ y)\ D \wedge \mathsf{bind}\ y\ (\mathsf{loc}\ d) \wedge \mathsf{cell}\ d\ \mathsf{fst}\ \mathsf{null} \wedge \mathsf{cell}\ d\ \mathsf{snd}\ \mathsf{null} \wedge (y \doteq d \doteq \mathsf{var}(\lambda x.E\ x))$$
$$\mathsf{eval}\ (\mathsf{let}\ L\ (\mathsf{proj}\ X\ C)\ (\lambda x.E\ x))\ D \wedge \mathsf{bind}\ X\ (\mathsf{loc}\ D_X) \wedge \mathsf{cell}\ D_X\ C\ V \tag{13}$$
$$\supset \exists y.\ \mathsf{eval}\ (E\ y)\ D \wedge \mathsf{bind}\ y\ V \wedge \mathsf{cell}\ D_X\ C\ V \wedge (y \doteq \mathsf{var}(\lambda x.E\ x))$$
$$\mathsf{eval}\ (\mathsf{let}\ L\ (\mathsf{set}\ X\ C\ Y)\ (\lambda x.E\ x))\ D \wedge \mathsf{bind}\ X\ (\mathsf{loc}\ D_X) \wedge \mathsf{bind}\ Y\ V \tag{14}$$
$$\supset \exists y.\ \mathsf{eval}\ (E\ y)\ D \wedge \mathsf{bind}\ y\ \mathsf{null} \wedge \mathsf{cell}\ D_X\ C\ V \wedge (y \doteq \mathsf{var}(\lambda x.E\ x))$$

Fig. 14 Approximating the monadic language with mutable references by uniformly equating every parameter introduced existential quantification with a simplified Skolem function dependent only on the higher-order term in the rule.

without retn.[8]

$$\mathsf{eval}(\mathsf{let}\ L\ (\mathsf{fun}\ (\lambda x_0.E_0\ x_0))\ (\lambda x.E\ x)) \twoheadrightarrow \exists y.\ \mathsf{eval}(E\ y) \bullet\ !\mathsf{bind}\ y\ (\mathsf{lam}(\lambda x_0.E_0\ x_0))) \tag{3}$$
$$\mathsf{eval}(\mathsf{let}\ L\ (\mathsf{call}\ F\ X)\ (\lambda x.E\ x)) \bullet\ !\mathsf{bind}\ F\ (\mathsf{lam}(\lambda x_0.E_0\ x_0)) \bullet\ !\mathsf{bind}\ X\ V \tag{4}$$
$$\twoheadrightarrow \exists y.\ \mathsf{comp}(\mathsf{call}_1(\lambda x.E\ x)) \bullet \mathsf{eval}(E_0\ y) \bullet\ !\mathsf{bind}\ y\ V$$
$$\mathsf{comp}(\mathsf{call}_1(\lambda x.E\ x)) \bullet \mathsf{eval}(\mathsf{return}\ L\ X) \bullet\ !\mathsf{bind}\ X\ V \twoheadrightarrow \exists y.\ \mathsf{eval}(E\ y) \bullet\ !\mathsf{bind}\ y\ V \tag{5}$$

The rules for mutable pairs below are our first explicit use of linear atomic propositions in an ordered logical specification (though we hinted at their use in call-by-need specifications in Section 2.2). 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.

$$\mathsf{eval}(\mathsf{let}\ L\ \mathsf{newpair}\ (\lambda x.E\ x)) \tag{6}$$
$$\twoheadrightarrow \exists y.\ \exists d.\ \mathsf{eval}(E\ y) \bullet\ !\mathsf{bind}\ y\ (\mathsf{loc}\ d) \bullet\ ¡\mathsf{cell}\ d\ \mathsf{fst}\ \mathsf{null} \bullet\ ¡\mathsf{cell}\ d\ \mathsf{snd}\ \mathsf{null}$$
$$\mathsf{eval}(\mathsf{let}\ L\ (\mathsf{proj}\ X\ C)\ (\lambda x.E\ x)) \bullet\ !\mathsf{bind}\ X\ (\mathsf{loc}\ D) \bullet\ ¡\mathsf{cell}\ D\ C\ V \tag{7}$$
$$\twoheadrightarrow \exists y.\ \mathsf{eval}(E\ y) \bullet\ !\mathsf{bind}\ y\ V \bullet\ ¡\mathsf{cell}\ D\ C\ V$$
$$\mathsf{eval}(\mathsf{let}\ L\ (\mathsf{set}\ X\ C\ Y)\ (\lambda x.E\ x)) \bullet\ !\mathsf{bind}\ X\ (\mathsf{loc}\ D) \bullet\ !\mathsf{bind}\ Y\ V \bullet\ ¡\mathsf{cell}\ D\ C\ V' \tag{8}$$
$$\twoheadrightarrow \exists y.\ \mathsf{eval}(E\ y) \bullet\ !\mathsf{bind}\ y\ \mathsf{null} \bullet\ ¡\mathsf{cell}\ D\ C\ V$$

When we approximate the specification in rules 3-8, we force our hand almost entirely if we follow the pattern of equating every existential y with a simplified Skolem function that is dependent only on the higher-order term it is being substituted into.

[8] Doing away with retn simplifies our presentation, but it is out of line with previous work [34] where we discuss a classification of predicates in SSOS specifications as *active*, *passive*, or *latent*. Because of rule 5 we cannot classify eval this way. This could be corrected by splitting rule 5 into two rules, one which generates a retn and another which mentions comp.

$$\mathsf{eval}\ (\mathsf{let}\ L\ \mathsf{newpair}\ (\lambda x.E\,x))\ D \tag{15}$$
$$\supset \mathsf{eval}\ (E\ L)\ D \wedge \mathsf{bind}\ L\ (\mathsf{loc}\ L) \wedge \mathsf{cell}\ L\ \mathsf{fst}\ \mathsf{null} \wedge \mathsf{cell}\ L\ \mathsf{snd}\ \mathsf{null}$$
$$\mathsf{eval}\ (\mathsf{let}\ L\ (\mathsf{proj}\ X\ C)\ (\lambda x.E\,x))\ D \wedge \mathsf{bind}\ X\ (\mathsf{loc}\ L_X) \wedge \mathsf{cell}\ L_X\ C\ V \tag{16}$$
$$\supset \mathsf{eval}\ (E\ L)\ D \wedge \mathsf{bind}\ L\ V \wedge \mathsf{cell}\ L_X\ C\ V$$
$$\mathsf{eval}\ (\mathsf{let}\ L\ (\mathsf{set}\ X\ C\ Y)\ (\lambda x.E\,x))\ D \wedge \mathsf{bind}\ X\ (\mathsf{loc}\ L_X) \wedge \mathsf{bind}\ Y\ V \tag{17}$$
$$\supset \mathsf{eval}\ (E\ L)\ D \wedge \mathsf{bind}\ L\ \mathsf{null} \wedge \mathsf{cell}\ L_X\ C\ V$$

Fig. 15 A version of Figure 14 using labels directly instead of simplified Skolem functions, and where equivalent versions of the rules that do not explicitly mention equality are used.

Once we do so, there are only two existentially generated parameters left to consider: the destination generated when we create a new pair in (rule 6), and the destination generated by the translation to linear destination-passing style in the rule handling procedure calls (rule 4). One option is just equate the destinations with the simplified Skolem function we are using in the same rule; the result of this choice is shown in Figure 14. That figure also incorporates one additional simplification: the last premise of rule 7, which served only to consume the linear resource ¡$\mathsf{cell}\ D\,C\,V'$, is removed in the approximate version (as the meta-approximation theorem allows). The resulting specification is terminating for the same reason the control flow analysis was – there are only a finite number of "subterms" of the term that we start with.

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. These labels allow us to consider an alternative to the pattern of equating parameters substituted into higher-order terms $(\lambda x.\,E\,x)$ with simplified Skolem functions $\mathsf{var}(\lambda x.\,E\,x)$. With one exception (the function call in rule 10) every use of a simplified Skolem function of the form $\mathsf{var}(\lambda x.E\,x)$ originates from an expression $(\mathsf{let}\ L\ M\ (\lambda x.E\,x))$. In other words, each $\mathsf{var}(\lambda x.E\,x)$ 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 15 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 answers to some of the primary questions asked of an alias analysis much clearer. For instance:

- *Might the first component of a pair created at label L_1 ever reference a pair created at label L_2?* Only if $\mathsf{cell}\ L_1\ \mathsf{fst}\ (\mathsf{loc}\ L_2)$ appears in the saturated database (and likewise for the second component).
- *Might the first component of a pair created at label L_1 ever reference the same object as the first component of a pair created at label L_2?* Only if there is some L' such that $\mathsf{cell}\ L_1\ \mathsf{fst}\ (\mathsf{loc}\ L')$ and $\mathsf{cell}\ L_2\ \mathsf{fst}\ (\mathsf{loc}\ L')$ both appear in the saturated database.

6 Conclusion

We have defined a framework of ordered, linear, and persistent atomic propositions with higher-order terms and equality assertions; this framework extends the one in previous work [34] with equality between terms. This framework is suitable for specifying interpreters for programming languages in the style of substructural operational

semantics. These specifications in ordered logic can be automatically translated to linear logical specifications by adding destinations, which preserves the adequacy of those specifications while exposing information about control flow that can be manipulated by an eventual approximation. We presented a general criterion for determining when one program was an approximate version of another (Definition 3) and for determining when one state is an abstraction of another (Definition 5) and established a meta-approximation theorem that ensures that approximations compute abstractions when interpreted as saturating logic programs. 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.

6.1 Related work

This article covers topics in programming language specification, proof theory, logical frameworks, logic programming, and program analysis. We conclude by giving a (necessarily incomplete) survey of some of the related work in these fields.

Approximation with equality. In this paper, we essentially presented only one method of deriving a saturating logic program from a persistent logical specification: first, we utilize Skolemization (done by adding equality constraints in order to fit the pattern prescribed by Definition 5), and then we approximate the resulting Skolem function. Another style of approximation, which is considered in the earlier conference version of this paper, is 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). This technique is quite powerful: for instance, a constraint that lists agreeing on their first k elements should be treated as equal is all that is needed to derive a k-CFA analysis from an appropriately constructed exact interpreter [46].

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 [15,40]. Another view, which is closer to the use of equality in the conference version of this article, is based on constraints and has been explored by Virga and Saraswat et al. [17,50]. We did not consider constraint-based or congruence-closure based equality in this paper because it is not currently clear how these two notions of equality should interact in our setting.

Substructural logic programming. Considered as a logic programming language, the ordered logical specification framework we have presented is unlike most previous work in logic programming languages for substructural logics. 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 [28]. Backward chaining operational semantics have been given to the uniform fragment of linear logic [16], ordered logic [36,37], and bunched logic [3].

There is, however, a significant body of work in giving forward-chaining interpretations to linear logic, including Forum [26] and Lollimon [20]. Other languages and programming paradigms, such as multiset rewriting (MSR) and Gamma, are defined independently of linear logic but can be partially or completely described in terms of proof search in linear logic [10,12].

Approximation in linear logic. 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 Miller [27]. Miller's approximations remain linear, which we have ruled out so far in order to obtain a simple meta-approximation theorem.

Implementation of saturating logic programs. We have not considered the implementation of a saturating logic programming language beyond the idealized interpreter which we used to reason about termination. However, there is a great deal of recent work in using saturating logic programming to implement program analyses, such as Whaley and Lam's extremely successful BDDBDDB engine [18] and the Doop framework for Java pointer analysis [9]. These lines of work are both premised on efficient, scalable Datalog implementations. Our programs are not Datalog because of the presence of function symbols, so the approximations we generate cannot directly be fed to these efficient engines. However, the adaptation of standard techniques such as *flattening* [39] to our uses of higher-order abstract syntax could be used to address this shortcoming.

McAllester's work established that it was possible to implement algorithms as saturating logic programs with provable asymptotic bounds on running times, and many of his examples were drawn from program analysis [21]. In previous work we have examined a extension of McAllester's results to forward chaining in linear logic [45]; the translation in Section 3 allows these results to be extended to reasoning about the complexity of certain ordered logical specifications. However, all of this work takes place in a setting with only first-order terms and without equality, and it is not obvious how the meta-complexity theorems should be generalized to the full language of ordered, linear, or persistent logical specifications.

Logical frameworks for specifying abstract machines. Throughout this article, we have relied on an intuition of specifications in ordered logic as rewriting rules. In fact, there is a significant line of work interested in directly specifying programming language semantics as rewriting rules in a rewriting logic [22,38,42]. While rewriting logic lacks a notion of higher-order syntax, the approach we have described in this article could almost certainly be fruitfully applied in rewriting logic as well.

We used an ordered logical framework as the basis for our work instead of rewriting logic to connect it to a larger body of research on the substructural operational semantics of programming languages. Previous work has shown that ordered logical specifications can cleanly specify stateful and concurrent programming languages and systems [34], and ongoing work (still in a preliminary stage) has demonstrated that it is possible to establish properties such as language safety in an extension of the ordered logical framework presented here [44].

Abstracting functional abstract machines. In recent work, Might and Van Horn introduced an approach to program approximation with many parallels to the methodology we have considered in this paper [24, 23, 49]. Their emphasis is on deriving a program approximation by approximating a *functional* abstract interpreter for a programming language's operational semantics. Their methodology is similar to ours in large part because we are doing the same thing in a different setting, deriving a program approximation by approximating an *ordered logical specification* of a programming language's operational semantics.

Many of the steps that they suggest for approximating programs have close analogues in our setting. For instance, their *store-allocated bindings* are analogous to the SSOS environment semantics that we presented, and their *store-allocated continuations* – which they motivate by analogy to implementation techniques for functional languages like SML/NJ – are precisely the destinations that arise naturally from our destination-adding translation. The first approximation step we take is forgetting about linearity to obtain a (nonterminating) persistent logical specification (e.g. the rules push_4 and pop_4 for the PDA approximation). This step is comparable to Might's first approximation step of "throwing hats on everything" (named after the convention in abstract interpretation of denoting the abstract version of a state space Σ as $\hat{\Sigma}$). The "mysterious" introduction of power domains that this entails in Might's setting is, in our setting, a perfectly natural result of relaxing the requirement that there be at most one fact $!\mathsf{bind}\, X\, V$ for every X. As a final point of comparison, the "abstract allocation strategy" discussed in [49] is quite similar to our strategy of introducing and then approximating Skolem functions as a means of deriving a finite approximation. Our current discussion of the approximation of Skolem functions in Section 4.2 is partially inspired by the relationship between our use of Skolemization and the discussion of abstract allocation in [49].

The independent discovery of a similar set of techniques for achieving similar goals in such different settings (though both approaches were to some degree inspired by Van Horn and Mairson's investigations of the complexity of k-CFA [48]) is another indication of the generality of both techniques, and the similarity also suggests that that the wide variety of approximations considered in [49], as well as the approximations of object-oriented programming languages in [24], can be adapted to our setting.

6.2 Acknowledgements

We would like to acknowledge Jonathan Aldrich, Roger Wolff, and the three anonymous reviewers for their helpful comments on an earlier draft of this article.

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.M.: Logic programming with focusing proofs in linear logic. Journal of Logic and Computation **2**(3), 297–347 (1992)
3. Armelín, P., Pym, D.: Bunched logic programming. In: International Joint Conference on Automated Reasoning. Lecture Notes in Computer Science, vol. 2083, pp. 289–304. Springer, New York (2001)
4. Barber, A.: Dual intuitionistic linear logic. Tech. Rep. ECS-LFCS-96-347, Laboratory for Foundations of Computer Sciences, University of Edinburgh (1996)

5. van Benthem, J.: Language in Action: Categories, Lambdas and Dynamic Logic, *Studies in Logic and the Foundations of Mathematics*, vol. 130, chap. 16, pp. 225–250. North-Holland, Amsterdam (1991)
6. Bozzano, M., Delzanno, G.: Automated protocol verification in linear logic. In: International Conference on Principles and Practice of Declarative Programming, pp. 38–49. ACM (2002)
7. Bozzano, M., Delzanno, G., Martelli, M.: An effective fixpoint semantics for linear logic programs. Theory and Practice of Logic Programming **2**(1), 85–122 (2002)
8. Bozzano, M., Delzanno, G., Martelli, M.: Model checking linear logic specifications. Theory and Practice of Logic Programming **4**(5–6), 573–619 (2004)
9. Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. In: Object-Oriented Programming, Systems, Languages, and Applications, pp. 243–261. ACM (2009)
10. Bruscoli, P., Guglielmi, A.: A linear logic view of Gamma style computations as proof searches. In: J.M. Andreoli, C. Hankin, D.L. Métayer (eds.) Coordination programming: mechanisms, models and semantics, pp. 249–273. Imperial College Press, London, UK (1996)
11. 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
12. Cervesato, I., Scedrov, A.: Relating state-based and process-based concurrency through linear logic. Information and Computation **207**, 1044–1077 (2009)
13. Cousout, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Symposium on Principles of Programming Languages, pp. 238–252. ACM (1977)
14. Ganzinger, H., McAllester, D.A.: Logical algorithms. In: International Conference on Logic Programming. Lecture Notes in Computer Science, vol. 2401, pp. 209–223. Springer, New York (2002)
15. Girard, J.Y.: A fixpoint theorem in linear logic (1992). An email posting to the mailing list `linear@cs.stanford.edu`.
16. Hodas, J.S., Miller, D.: Logic programming in a fragment of intuitionistic linear logic. Information and Computation **110**(2), 327–365 (1994)
17. Jagadeesan, R., Nadathur, G., Saraswat, V.A.: Testing concurrent systems: An interpretation of intuitionistic logic. In: Foundations of Software Technology and Theoretical Computer Science. Lecture Notes in Computer Science, vol. 3821, pp. 517–528. Springer, New York (2005)
18. Lam, M.S., Whaley, J., Livshits, V.B., Martin, M.C., Avots, D., Carbin, M., Unkel, C.: Context-sensitive program analysis as database queries. In: Principles of Database Systems, pp. 1–12. ACM (2005)
19. Lambek, J.: The mathematics of sentence structure. American Mathematical Monthly **65**, 363–386 (1958)
20. López, P., Pfenning, F., Polakow, J., Watkins, K.: Monadic concurrent linear logic programming. In: Principles and Practice of Declarative Programming, pp. 35–46. ACM (2005)
21. McAllester, D.A.: On the complexity analysis of static analyses. Journal of the ACM **49**(4), 512–537 (2002)
22. Meseguer, J., Roşu, G.: The rewriting logic semantics project. Theoretical Computer Science **373**, 213–237 (2007)
23. Might, M.: Abstract interpreters for free. In: Static Analysis Symposium. Lecture Notes in Computer Science, vol. 6337, pp. 407–421. Springer, New York (2010)
24. Might, M., Smaragdakis, Y., Van Horn, D.: Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. In: Programming Language Design and Implementation, pp. 305–315. ACM (2010)
25. Miller, D.: A logic programming language with lambda-abstraction, function variables, and simple unification. Journal of Logic and Computation **1**(4), 497–536 (1991)
26. Miller, D.: A multiple-conclusion meta-logic. Theoretical Computer Science **165**(1), 201–232 (1996)
27. 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*, vol. 17, pp. 423–442. College Publications, London, UK (2008)

28. Miller, D., Nadathur, G., Pfenning, F., Scedrov, A.: Uniform proofs as a foundation for logic programming. Annals of Pure and Applied Logic **51**(1-2), 125–157 (1991)
29. Moot, R., Piazza, M.: Linguistic applications of first order intuitionistic linear logic. Journal of Logic, Language and Information **10**(2), 211–232 (2001)
30. Morrill, G.: Higher-order linear logic programming of categorial deduction. In: Proceedings of the Meeting of the European Chapter of the Association for Computational Linguistics, pp. 133–140. Morgan Kaufmann Publishers Inc. (1995)
31. Pfenning, F.: Substructural operational semantics and linear destination-passing style. In: Asian Symposium on Programming Languages and Systems. Lecture Notes in Computer Science, vol. 3302, p. 196. Springer, New York (2004). Abstract of invited talk
32. Pfenning, F., Elliott, C.: Higher-order abstract syntax. In: Programming Language Design and Implementation, pp. 199–208. ACM (1988)
33. Pfenning, F., Schürmann, C.: Algorithms for equality and unification in the presence of notational definitions. In: Types for Proofs and Programs. Lecture Notes in Computer Science, vol. 1657, pp. 179–193. Springer, New York (1998)
34. Pfenning, F., Simmons, R.J.: Substructural operational semantics as ordered logic programming. In: Symposium on Logic in Computer Science, pp. 101–110. IEEE Computer Society Press (2009)
35. 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.
36. Polakow, J.: Linear logic programming with an ordered context. In: International Conference on Principles and Practice of Declarative Programming, pp. 68–79. ACM (2000)
37. Polakow, J.: Ordered linear logic and applications. Ph.D. thesis, Carnegie Mellon University (2001). Available as technical report CMU-CS-01-152
38. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. Journal of Logic and Algebraic Programming **79**(6) (2010)
39. Rouveirol, C.: Flattening and saturation: Two representation changes for generalization. Machine Learning **14**, 219–232 (1994)
40. Schroeder-Heister, P.: Rules of definitional reflection. In: Symposium on Logic and Computer Science, pp. 222–232. IEEE Computer Society Press (1993)
41. Schürmann, C.: Automating the meta theory of deductive systems. Ph.D. thesis, Carnegie Mellon University (2000). Available as Technical Report CMU-CS-00-146
42. Şerbănuţă, T.F., Roşu, G.: K-Maude: A rewriting based tool for semantics of programming languages. In: Rewriting Logic and its Applications. Lecture Notes in Computer Science, vol. 6381, pp. 104–122. Springer, New York (2010)
43. Shieber, S.M., Schabes, Y., Pereira, F.C.N.: Principles and implementation of deductive parsing. Journal of Logic Programming **24**(1–2), 3–36 (1995)
44. Simmons, R.J.: Type safety for substructural specifications: Preliminary results. Tech. Rep. CMU-CS-10-134, School of Computer Science, Carnegie Mellon University (2010)
45. Simmons, R.J., Pfenning, F.: Linear logical algorithms. In: International Colloquium on Automata, Languages and Programming. Lecture Notes in Computer Science, vol. 5126, pp. 336–345. Springer, New York (2008)
46. Simmons, R.J., Pfenning, F.: Linear logical approximations. In: Partial Evaluation and Program Manipulation, pp. 9–20. ACM (2009)
47. Simmons, R.J., Pfenning, F.: Weak focusing for ordered linear logic. Tech. Rep. CMU-CS-10-147, School of Computer Science, Carnegie Mellon University (2011)
48. Van Horn, D., Mairson, H.G.: Relating complexity and precision in control flow analysis. In: International Conference on Functional Programming, pp. 85–96. ACM (2007)
49. Van Horn, D., Might, M.: Abstracting abstract machines. In: International Conference on Functional Programming, pp. 51–62. ACM (2010)
50. Virga, R.: Higher-order rewriting with dependent types. Ph.D. thesis, Carnegie Mellon University (1999)

A Proof of Theorem 2 (Correctness of translation)

In this section, we present the proof of Theorem 2, which (informally) ensures that if the specification $\Gamma_{\mathcal{P}}$ is an adequate representation of some transition system then $[\![\Gamma_{\mathcal{P}}]\!]$,

the result of the destination-adding translation on $\Gamma_{\mathcal{P}}$, is an adequate representation of the transition system as well.

Previously, we defined a sequential derivation from $\mathbb{S}_2$ to $\mathbb{S}_1$, which we also write as $\mathbb{S}_1 \xrightarrow{-+} \mathbb{S}_2$, to be a single focusing phase of the form

$$\begin{array}{c} \mathbb{S} \\ \vdots \\ \mathbb{S}' \end{array}$$

where the invertible left rules are only ever applied to the left-most non-atomic proposition in an ordered context. For the purposes of this section we will generalize this definition somewhat: a sequential derivation always has a state as the unproven leaf at the top, but the bottom of a sequential derivation can be either a left-focused sequent $(\Gamma; \Delta; \Omega_L[A]\Omega_R \Rightarrow_\Sigma S)$ or an unfocused sequent $(\Gamma; \Delta; \Omega \Rightarrow_\Sigma S)$. Note that, in this more general definition, a sequential derivation cannot include the rule $\mathbf{focus}_L$ unless the bottom-most sequent is a state.

A.1 Right focus

The first lemma establishes the "$[\![\mathbb{S}]\!] \rightarrow|$ if and only if $\mathbb{S} \rightarrow|$" portion of Theorem 2. The first part of Lemma 1 is just that $\mathbb{S} \rightarrow|$ implies $[\![\mathbb{S}]\!] \rightarrow|$, and the second part of the lemma generalizes the converse direction; the generalization is important in order to deal with the right-focused sequents that arise from the $\twoheadrightarrow_L$ rule.

Lemma 1 (Correctness of translation for right focus) *For all contexts Γ, Δ, and Ω containing only atomic propositions:*

1. *If there is a derivation of $\Gamma; \Delta; \Omega \Rightarrow_\Sigma [S]$, then there is also a derivation of $[\![\Gamma; \Delta; \Omega \Rightarrow_\Sigma [S]]\!]_{\mathbb{S}}$.*
2. *If there is a derivation of $\Gamma; \Delta; \cdot \Rightarrow_{\Sigma d_0 \ldots d_n} [[\![S]\!]^t_s]$ and $\Delta \subset \Delta_o [\![\Omega_o]\!]^{d_0}_{d_n}$,*
   then $\Delta = \Delta' [\![\Omega']\!]^{d_j}_{d_k}$ 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 $t = s$.
3. *If there is a derivation of $\Gamma; \Delta; \Omega \Rightarrow_\Sigma [S]$ and S contains no ordered atomic propositions, then $\Omega = \cdot$.*

Proof Induction over the structure of right-focused derivations. □

A.2 Weakening and strengthening

The next two lemmas express that we can always add or remove irrelevant parts of a sequential derivation (and that we can do so without changing its structure). Lemma 2 establishes that the only times that the rules (which are closed negative propositions in the persistent context) matter for a sequential derivation is the initial application of $\mathbf{focus}_L$, and Lemma 3 indicating that we can add or remove unmentioned parameters from a (sequential) derivation freely.

Lemma 2 (Weakening/strenghtening rules) *If $\Gamma_{\mathcal{P}}$ and $\Gamma_{\mathcal{Q}}$ contain closed negative propositions (i.e. rules), and there is a sequential derivation from $\Gamma_{\mathcal{P}}\Gamma'; \Delta'; \Omega' \Rightarrow_\Sigma S$ to $\Gamma_{\mathcal{P}}\Gamma; \Delta; \Omega_L[A]\Omega_R \Rightarrow_\Sigma S$, then there is a sequential derivation (of the same size) from $\Gamma_{\mathcal{Q}}\Gamma'; \Delta'; \Omega' \Rightarrow_\Sigma S$ to $\Gamma_{\mathcal{Q}}\Gamma; \Delta; \Omega_L[A]\Omega_R \Rightarrow_\Sigma S$.*

Proof Straightforward induction on derivations; this theorem merely reflects that the only time the rules are relevant is in the rule $\mathbf{focus}_L$, so if we do not use that rule we can remove or replace the specification as it is convenient. □

Lemma 3 (Weakening/strenghtening parameters) *If Γ, Δ, and Ω contain atomic propositions that do not mention the parameters in Σ' or Σ'', then*

1. *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*
2. *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. □

A.3 Correctness of inversion

The correctness of translation for sequential derivations consisting only of invertible left rules can be separated from correctness in general. This part is where we take advantage of the requirement that, in a sequential derivation, an invertible left rule is always applied to the leftmost non-atomic proposition.

Lemma 4 (Completeness of translation for left inversion) *If Γ, Δ, Ω_L, and Ω_R contain only atomic propositions and $\mathbb{S}$ is a state, then*

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

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

Proof Lexicographic induction: either the length of the derivation $\mathcal{D}$ gets smaller or stays the same while the number of propositions in 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 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 the requirement that left rules always be applied to the leftmost non-atomic proposition ensures that the last rule in the derivation must be $\bullet_L$ and that there is a sub-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:

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

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

This completes the case; the five other cases ($S = ¡Q, !Q, \exists x.S, \mathbf{1}, t \doteq s$) are similar. □

Lemma 5 (Soundness of translation for left inversion) *For any contexts Γ', Δ', Ω', Γ, Δ, Ω_L, and Ω_R containing only atomic propositions,*

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

Proof Induction on the length of the derivation $\mathcal{D}$, as in the proof of Lemma 4. □

The use of $d_0 \ldots d_m$ in Lemma 5 and elsewhere is an acknowledgement of the fact that the number of ordered propositions may change throughout a sequential derivation, which effectively forces us to "re-number" the subscripts on the destinations. As an example, if $\Omega_L = Q_1$, $\Omega_R = Q_2$, and $\Omega = \mathbf{1}$ in the statement of Lemma 5, then Ω_L will get translated to $Q_1\, d\, d'$, Ω will get translated to $d' \doteq d''$, and Ω_R will get translated as $Q_2\, d''\, d'''$. Because the $\doteq_L$ rule will force us to unify d' and d'', we have $n = 3$, $\Sigma_\Omega = d\, d'\, d''\, d'''$ and $m = 2$, $\Sigma' = \Sigma''\, d\, d'\, d'''$.

A.4 Main proof of Theorem 2

We now have the facts we need to prove Theorem 2. To recall:

Theorem 2 (Correctness of translation) *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

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

Proof The second point is, as we already discussed, a straightforward consequence of Lemma 1, so we will concentrate on the first case. The bottommost step in any transition $\mathbb{S} \xrightarrow{-+} \mathbb{S}'$ is an application of $\mathbf{focus}_L$ that picks out a rule $A \in \Gamma_{\mathcal{P}}$. Using Lemma 2 to avoid dealing with the the program $\Gamma_{\mathcal{P}}$ everywhere, it suffices to show:

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

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

If A is in the persistent or linear fragment. In this case, $[\![A]\!] = A$, and we can we generalize the above "if and only if" statement by replacing A and $[\![A]\!]$ with γ, where γ is either a left focus $[A]$ where A contains no ordered atomic propositions or else a context Ω containing no ordered atomic propositions. The proof proceeds by straightforward induction on the structure of the sequential derivation.

If A contains ordered atomic propositions. By separation, this means that the premise of A must contain ordered atomic propositions; the conclusion of the rule may or may not contain ordered atomic propositions. First, we will consider the reverse direction (completeness):

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

In the case that $A = \forall x.A'$ we use the induction hypothesis on the sub-derivation 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:

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

By $\mathcal{E}_1$ and the correctness of right focus (Lemma 1) we know there is a derivation $\mathcal{D}_1$ of $\Gamma; \Delta_1 [\![\Omega_1]\!]_{d_k}^{d_j}; \cdot \Rightarrow_{\Sigma d_j \ldots d_k} [[\![S_1]\!]_{d_k}^{d_j}]$. By $\mathcal{E}_2$ and the completeness of left inversion (Lemma 4) there is a sequential derivation $\mathcal{D}_2$:

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

We can weaken and rename the parameters in $\mathcal{D}_1$ and $\mathcal{D}_2$ (Lemma 3) to construct the following:

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

where $\Sigma_\Omega = \Sigma_w d_0 \ldots d_j \ldots d_k \ldots d_n$ 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:

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

In the case that $A = \forall x.A'$ we again use the induction hypothesis on the sub-derivation 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:

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

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 1) 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 do not 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 5), $\mathcal{D}_2$, and strengthening of parameters (Lemma 3), 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 $\mathcal{E}_2$:

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

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. □

B Proof of Theorem 3 (Meta-approximation)

In this section, we present a proof of Theorem 3, which 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 the next two formalize the notion that, in an approximate version of a program, a saturated state captures all the "behaviors" of the state it evolved from. 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 6 (One-step simulation) *Let* $\mathbb{S} = (\Gamma_\mathcal{P}\Gamma; \Delta; \Omega \Rightarrow_\Sigma S)$ *and let* $\mathbb{S}_a = (\Gamma_a\Gamma_g \Rightarrow_{\Sigma_g} S!)$ *be a generalization of* $\mathbb{S}$. *If* $\mathbb{S} \xrightarrow{-+} \mathbb{S}^+$ *by focusing on the rule* $A \in \Gamma_\mathcal{P}$, *then by focusing on a rule* $A_a \in \Gamma_a$, $\mathbb{S}_g \xrightarrow{-+} \mathbb{S}_g^+$ *such that* $\mathbb{S}_g^+$ *is a generalization of* $\mathbb{S}^+$.

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 $(\Gamma_\mathcal{P}\Gamma'; \Delta'; \Omega' \Rightarrow_{\Sigma'} S)$ to $(\Gamma_\mathcal{P}\Gamma; \Delta; \Omega_L[\sigma A]\Omega_R \Rightarrow_\Sigma S)$, and if the state $(\Gamma_a\Gamma_g \Rightarrow_{\Sigma_g} S!)$ is a generalization of $(\Gamma_\mathcal{P}\Gamma; \Delta; \Omega_L\Omega_R \Rightarrow_\Sigma S)$ by way of the substitution $\Sigma_g \vdash \theta : \Sigma$, then there is a sequential derivation from $(\Gamma_a\Gamma'_g \Rightarrow_{\Sigma'_g} S!)$ to $(\Gamma_a\Gamma_g; \cdot; [\theta(\sigma A_a)] \Rightarrow_{\Sigma_g} S!)$ such that the former is a generalization of $(\Gamma_\mathcal{P}\Gamma'; \Delta'; \Omega' \Rightarrow_{\Sigma'} S)$. The substitution σ just tracks what terms have been substituted for the universally quantified variables in A.

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:

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

The approximate version of $S_1 \twoheadrightarrow S_2$ is $S_1' \twoheadrightarrow S_2'$, and we have $\mathcal{E}_1$, a derivation of $(\Gamma_a\Gamma_g \Rightarrow_{\Sigma_g} [\theta(\sigma S_1)])$ by induction over the structure of S_1'. All the components of S_1' also appear in S_1, and so in the base case we need to establish that $(\Gamma_a\Gamma_g; \cdot; \cdot \Rightarrow_{\Sigma_g} [!(\theta(\sigma Q))])$ and 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 $(\Gamma_a\Gamma'_g \Rightarrow_{\Sigma'_g} S!)$ to $(\Gamma_a\Gamma_g; \cdot; \theta(\sigma S_2') \Rightarrow_{\Sigma_g} S!)$ such that the former is a generalization of $(\Gamma_\mathcal{P}\Gamma'; \Delta'; \Omega' \Rightarrow_{\Sigma'} S)$ – from this we can conclude with $\twoheadrightarrow_L$. We generalize the conclusions S_2 and S_2' to contexts Ω_2 and Ω_2'. The remaining proof obligation is that, given a sequential derivation $\mathcal{D}_2$ from $(\Gamma_\mathcal{P}\Gamma'; \Delta'; \Omega' \Rightarrow_{\Sigma'} S)$ to $(\Gamma_\mathcal{P}\Gamma; \Delta_2; \Omega_L(\sigma\Omega_2)\Omega_R \Rightarrow_\Sigma S)$, if $(\Gamma_a\Gamma_g \Rightarrow_{\Sigma_g} S!)$ is a generalization of $(\Gamma_\mathcal{P}\Gamma; \Delta_2; \Omega_L\Omega_R \Rightarrow_\Sigma S)$ then there is a sequential derivation $\mathcal{E}_2$ from $(\Gamma_a\Gamma'_g \Rightarrow_{\Sigma'_g} S!)$ to $(\Gamma_a\Gamma_g; \cdot; \theta(\sigma\Omega_2') \Rightarrow_{\Sigma_g} S)$ where $(\Gamma_a\Gamma'_g \Rightarrow_{\Sigma'_g} S!)$

is a generalization of $(\Gamma_{\mathcal{P}}\Gamma'; \Delta'; \Omega' \Rightarrow_{\Sigma'} S)$. 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 4.

The most interesting case is where Ω_2' is $(t \doteq s)\Omega'$. We assume that we can compute the most general unifier $(\Sigma_g' \vdash \tau : \Sigma_g)$ of the terms $\theta(\sigma t)$ and $\theta(\sigma s)$. If the equality $t \doteq s$ appears only in the approximate version, then because $(\Gamma_a\Gamma_g \Rightarrow_{\Sigma_g} S!)$ is a generalization of $(\Gamma_{\mathcal{P}}\Gamma; \Delta_2; \Omega_L\Omega_R \Rightarrow_{\Sigma} S)$, we have that $(\Gamma_a(\tau\Gamma_g) \Rightarrow_{\Sigma_g'} S!)$ is also a generalization of $(\Gamma_{\mathcal{P}}\Gamma; \Delta; \Omega_L\Omega_R \Rightarrow_{\Sigma} S)$. Therefore, 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 $(\Gamma_a(\tau\Gamma_g); \cdot; \tau(\theta(\sigma\Omega')) \Rightarrow_{\Sigma_g'} \tau S)$, and we conclude by applying $\doteq_L'$. If, on the other hand, the equality $t \doteq s$ 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 $(\Gamma_a(\tau\Gamma_g) \Rightarrow_{\Sigma_g'} S!)$ is a generalization of $(\Gamma_{\mathcal{P}}(\tau'\Gamma); \tau'\Delta; \tau'\Omega \Rightarrow_{\Sigma''} S)$, which amounts to showing that if $A \in (\Gamma \cup \Delta \cup \Omega)$, then $\theta(\tau' A) \in \tau\Gamma_g$. We have that $\theta A \in \Gamma_g$ because of the generalization that we started with, 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 7 (Simulation) *If* $(\Gamma_{\mathcal{P}}\Gamma_0; \Delta_0; \Omega_0 \Rightarrow_{\Sigma_0} S) \xrightarrow{-+} \ldots \xrightarrow{-+} \mathbb{S}_n$ *is a trace, if* $\Sigma_0' \vdash \theta : \Sigma_0$*, and if* Γ_a *is an approximate version of* $\Gamma_{\mathcal{P}}$*, then there is a generalization* $\mathbb{S}_n'$ *of* $\mathbb{S}_n$ *such that* $(\Gamma_{\mathcal{P}}(\theta\Gamma_0)(\theta\Delta_0)(\theta\Omega_0) \Rightarrow_{\Sigma_0'} S!) \xrightarrow{-+} \ldots \xrightarrow{-+} \mathbb{S}_n'$.

Proof By induction on the length of the program trace. The base case is immediate and the inductive case follows from Lemma 6. □

Lemma 8 (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 9 (Saturation) *If the program* $\Gamma_{\mathcal{P}}$ *contains no linear or ordered predicates and there is a trace* $\mathbb{S}_0 \xrightarrow{-+} \ldots \xrightarrow{-+} \mathbb{S}_a$ *where* $\mathbb{S}_a$ *is saturated, then if* $\mathbb{S}_0 \xrightarrow{-+} \ldots \xrightarrow{-+} \mathbb{S}_n$*, 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 8 – $\mathbb{S}_a$ 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 $\mathbb{S}_n$ evolves to $\mathbb{S}_{n+1}$. We need to show $\mathbb{S}_a$ is a generalization of $\mathbb{S}_{n+1}$. By Lemma 6, $\mathbb{S}_a$ 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 $\mathbb{S}_a$. □

B.1 Main theorem

To recall:

Theorem 3 (Meta-approximation)
If Γ_a is an approximate version of $\Gamma_{\mathcal{P}}$, there is a state $\mathbb{S}_0 = (\Gamma_{\mathcal{P}}\Gamma_0; \Delta_0; \Omega_0 \Rightarrow_{\Sigma_0} S)$, and for some $\Sigma_0' \vdash \theta : \Sigma_0$ there is a trace $(\Gamma_a(\theta\Gamma_0)(\theta\Delta_0)(\theta\Omega_0) \Rightarrow_{\Sigma_0'} S!) \xrightarrow{-+} \ldots \xrightarrow{-+} \mathbb{S}_a$ such that $\mathbb{S}_a$ is saturated, then $\mathbb{S}_a$ is an abstraction of $\mathbb{S}_0$.

Proof Consider a trace $(\Gamma_0; \Delta_0; \Omega_0 \Rightarrow_{\Sigma_0} S) \xrightarrow{-+} \ldots \xrightarrow{-+} \mathbb{S}_n$ of the original program. By Lemma 7 (Simulation) there is a trace $(\Gamma_a\theta(\Gamma_0\Delta_0\Omega_0) \Rightarrow_{\Sigma_0} S!) \xrightarrow{-+} \ldots \xrightarrow{-+} \mathbb{S}_n'$ of the approximate program such that $\mathbb{S}_n'$ generalizes $\mathbb{S}_n$. Then, by Lemma 9 (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. □