Type safety for substructural specifications: preliminary results

Robert J. Simmons

July 15, 2010
CMU-CS-10-134

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Substructural logics, such as linear logic and ordered logic, have an inherent notion of state and state change. This makes them a natural choice for developing logical frameworks that specify evolving stateful systems. Our previous work has shown that the so-called *forward reasoning* fragment of ordered linear logic can be used to give clear, concise, and modular specifications of stateful and concurrent features of programming languages in a style called Substructural Operational Semantics (SSOS).

This report is discusses two new ideas. First, I discuss a re-orientation of existing logics and logical frameworks based on a presentation of logic as a state transition system. Second, utilizing this transition-based interpretation of logic, I present a safety proof for the SSOS specification of a simple sequential programming language, and discuss how that proof can be updated as the language is extended with parallel evaluation and exception handling.

This report is derived from my thesis proposal [35], which was certified on July 7, 2010.

This material is based upon work supported under a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Keywords: logical frameworks, ordered logic, preservation, progress, safety, substructural logic, specification, thesis proposal

1 Introduction

Robust and flexible frameworks for specifying and reasoning about programming languages form the bedrock of programming languages research. In part, this is because they provide a common convention: historically, the most successful specification framework, Structural Operational Semantics (i.e. SOS), is the *lingua franca* of PL research. Even more critical is the ability of a framework to permit reasoning about properties of programming languages; in the case of SOS, the traditionally interrelated notion of "Safety = Progress + Preservation" has essentially come to define what it means for a programming language to make sense.

One problem with SOS specifications of programming languages is that they are *non-modular*: especially when dealing with imperative and concurrent programming language features, the natural specification of language *parts* cannot, in general, be combined to give a natural specification of the language *whole*. This makes it difficult to straightforwardly extend a pure "toy" programming language with a feature like imperative references without a complete overhaul of the language's definition and type safety proof; a similar overhaul is necessary to extend the definition of a large sequential language to handle concurrency. This has troubling practical consequences: it means that it is more difficult to scale from a small language formalization to a large one, and it means that existing language specifications are difficult to augment with features that were not anticipated in advance.

In this preliminary report, which is based on the content of my thesis proposal, I discuss a solution that involves two interconnected ideas. The first idea is a *specification style*, Substructural Operational Semantics (i.e. SSOS). SSOS specifications generalize abstract machine specifications and permit clear and modular specification of programming language features that involve state and concurrency — the specification style that I consider is a generalization of the ordered SSOS specifications as presented in existing work [28].

The second idea is a *logical framework* based on forward-chaining in ordered linear logic. The specifics of this framework are broadly similar to the ordered logical framework discussed in the aforementioned paper on SSOS specifications, and the framework also shares broad similarities with an existing family of logical frameworks such as LF [11], LLF [1], HLF [30], OLF [29], and (in particular) CLF [40]. The interesting difference between existing work and this framework is that this framework is based on an interpretation of logic that has a first-class notion of *state transitions*; as a result, a "step" of an SSOS specification can be represented by a state transition in the logical framework.

The ability to talk about individual transitions and potentially non-terminating computations is essential for giving a full account of type safety, since type safety must also account for the behavior of non-terminating programs. The primary novel contribution in this report is a proof of type safety, by way of progress and preservation lemmas, for the SSOS specification of a simple programming language. I then discuss how that proof can be straightforwardly extended to handle program extensions which introduce concurrency, non-local transfer of control, and state.

In Section 2 I motivate substructural operational semantics specifications by discussing modular and non-modular extensions to programming language specifications, and in Section 3 I motivate the treatment of logic as a state transition system. Section 4 presents a logical framework based on the ideas in Section 3; this section is rather technical and can probably be skipped on a first reading. In Section 5 I present a variety of substructural operational semantics specifications and proofs of their type safety. Both the dynamic and static semantics of the programming languages are presented as ordered logical specifications.

2 Modular and non-modular specification

In the introduction, the phrase *modular specification* was used but not defined. Modularity means different things in different settings; in this section, we will motivate the problem by considering modular and non-modular extensions to a very simple programming language. We start by assuming that we have natural numbers n and some operations (such as addition) on those natural numbers; expressions e can then be defined by giving a BNF specification:

$$e ::= \mathsf{num}(n) \mid e_1 + e_2$$

Call this language of numbers and addition $\mathcal{L}_0$; we next give a small-step structural operational semantics for $\mathcal{L}_0$ that specifies a left-to-right evaluation order. We also adopt the standard convention of writing expressions that are known to be values as v, not e:

$$\frac{}{\mathsf{num}(n_1)\ \mathsf{value}} \quad \frac{e_1 \mapsto e_1'}{e_1 + e_2 \mapsto e_1' + e_2} \quad \frac{v_1\ \mathsf{value} \quad e_2 \mapsto e_2'}{v_1 + e_2 \mapsto v_1 + e_2'} \quad \frac{n_1 + n_2 = n_3}{\mathsf{num}(n_1) + \mathsf{num}(n_2) \mapsto \mathsf{num}(n_3)}$$

The extension of $\mathcal{L}_0$ with eager pairs ($e ::= \ldots \mid \pi_1 e \mid \pi_2 e \mid \langle e_1, e_2 \rangle$) is an example of a modular extension. We can take the following natural small-step SOS specification of eager pairs and concatenate it together with the $\mathcal{L}_0$ specification to form a coherent (and, with a suitable type system, provably type-safe) specification of a language that has both features.

$$\frac{v_1\ \mathsf{value} \quad v_2\ \mathsf{value}}{\langle v_1, v_2 \rangle\ \mathsf{value}} \quad \frac{e_1 \mapsto e_1'}{\langle e_1, e_2 \rangle \mapsto \langle e_1', e_2 \rangle} \quad \frac{v_1\ \mathsf{value} \quad e_2 \mapsto e_2'}{\langle v_1, e_2 \rangle \mapsto \langle v_1, e_2' \rangle}$$

$$\frac{e \mapsto e'}{\pi_1 e \mapsto \pi_1 e'} \quad \frac{\langle v_1, v_2 \rangle\ \mathsf{value}}{\pi_1 \langle v_1, v_2 \rangle \mapsto v_1} \quad \frac{e \mapsto e'}{\pi_2 e \mapsto \pi_2 e'} \quad \frac{\langle v_1, v_2 \rangle\ \mathsf{value}}{\pi_2 \langle v_1, v_2 \rangle \mapsto v_2}$$

In contrast, the extension of $\mathcal{L}_0$ with ML-style imperative references is an obvious example of a non-modular extension. The extended syntax is ($e ::= \ldots \mid \mathsf{ref}\ e \mid \mathsf{loc}[l] \mid\ !e \mid e_1\ \mathsf{gets}\ e_2$) — the expression $\mathsf{ref}\ e$ creates a reference, $!e$ dereferences a reference, $e_1\ \mathsf{gets}\ e_2$ assigns the value of e_2 to the location represented by e_1, and $\mathsf{loc}[l]$ is a value, which only exists at runtime, referencing the abstract heap location l where some value is stored. In order to give a small-step SOS specification for the extended language, we must mention a store σ in every rule:

$$\frac{}{\mathsf{loc}[l]\ \mathsf{value}} \quad \frac{(\sigma, e) \mapsto (\sigma', e')}{(\sigma, \mathsf{ref}\ e) \mapsto (\sigma', \mathsf{ref}\ e')} \quad \frac{v\ \mathsf{value} \quad l \notin \mathrm{dom}(\sigma)}{(\sigma, \mathsf{ref}\ v) \mapsto (\sigma[l := v], \mathsf{loc}[l])}$$

$$\frac{(\sigma, e) \mapsto (\sigma', e')}{(\sigma, !e) \mapsto (\sigma', !e')} \quad \frac{\sigma(l) = v}{(\sigma, !\,\mathsf{loc}[l]) \mapsto (\sigma, v)}$$

$$\frac{(\sigma, e_1) \mapsto (\sigma', e_1')}{(\sigma, e_1\ \mathsf{gets}\ e_2) \mapsto (\sigma', e_1'\ \mathsf{gets}\ e_2)} \quad \frac{v_1\ \mathsf{value} \quad (\sigma, e_2) \mapsto (\sigma', e_2')}{(\sigma, v_1\ \mathsf{gets}\ e_2) \mapsto (\sigma', v_1\ \mathsf{gets}\ e_2')}$$

$$\frac{v_2\ \mathsf{value}}{(\sigma, \mathsf{loc}[l]\ \mathsf{gets}\ v_2) \mapsto (\sigma[l := v_2], v_2)}$$

The non-modularity of this extension comes from the fact that it forces us to rewrite the rules describing addition as well:

$$\frac{(\sigma, e_1) \mapsto (\sigma', e_1')}{(\sigma, e_1 + e_2) \mapsto (\sigma', e_1' + e_2)}$$

$$\frac{v_1\ \mathsf{value} \quad (\sigma, e_2) \mapsto (\sigma, e_2')}{(\sigma, v_1 + e_2) \mapsto (\sigma', v_1 + e_2')} \quad \frac{n_1 + n_2 = n_3}{(\sigma, \mathsf{num}(n_1) + \mathsf{num}(n_2)) \mapsto (\sigma, \mathsf{num}(n_3))}$$

2.1 Modularity and specification styles

Another interesting loss of modularity in SOS specifications appears when we introduce exceptions and exception handling ($e ::= \ldots \mid \mathsf{error} \mid \mathsf{try}\, e_1 \,\mathsf{ow}\, e_2$).

$$\frac{e_1 \mapsto e_2}{\mathsf{try}\, e_1 \,\mathsf{ow}\, e_2 \mapsto \mathsf{try}\, e_1' \,\mathsf{ow}\, e_2} \qquad \frac{v_1 \;\mathsf{value}}{\mathsf{try}\, v_1 \,\mathsf{ow}\, e_2 \mapsto v_1} \qquad \frac{}{\mathsf{try}\;\mathsf{error}\;\mathsf{ow}\, e_2 \mapsto e_2}$$

Our rules for addition do not need to be revised to be compatible with this extension, but in order to preserve type safety, we must provide a way for errors to "bubble up" through additions.

$$\frac{}{\mathsf{error} + e_2 \mapsto \mathsf{error}} \qquad \frac{v_1 \;\mathsf{value}}{v_1 + \mathsf{error} \mapsto \mathsf{error}}$$

We can avoid this particular form of non-modularity while technically staying within the SOS framework. This is possible if we use an *abstract machine* style of specification. Abstract machine-style specifications have an explicit control stack (or continuation), usually denoted k, that represents unfinished parts of the computation; a control stack is a list of *continuation frames*; we introduce two continuation frames f in order to specify $\mathcal{L}_0$:

- $(\Box + e_2)$ is a frame waiting on the first argument to be evaluated to a value so that the evaluation of e_2 can begin, and
- $(v_1 + \Box)$ is a frame holding the already-evaluated value v_1 waiting on the second argument to be evaluated to a value.

Basic abstract machine specifications have two kinds of states: an expression being evaluated on a stack ($k \rhd e$) and a value being returned to the stack ($k \lhd v$). The abstract machine specification of the operational semantics of $\mathcal{L}_0$ has four rules:

$$\begin{aligned}
k \rhd \mathsf{num}(n) &\mapsto k \lhd \mathsf{num}(n) \\
k \rhd (e_1 + e_2) &\mapsto (k, \Box + e_2) \rhd e_1 \\
(k, \Box + e_2) \lhd v_1 &\mapsto (k, v_1 + \Box) \rhd e_2 \\
(k, \mathsf{num}(n_1) + \Box) \lhd \mathsf{num}(n_2) &\mapsto k \lhd \mathsf{num}(n_3) \qquad\qquad (\text{if } n_1 + n_2 = n_3)
\end{aligned}$$

Given this specification, we can define the behavior of exceptions by adding a new frame ($\mathsf{try}\ \Box \,\mathsf{ow}\, e_2$) and a new kind of state — (k ◀), a stack dealing with an error.

$$\begin{aligned}
k \rhd \mathsf{try}\, e_1 \,\mathsf{ow}\, e_2 &\mapsto (k, \mathsf{try}\ \Box \,\mathsf{ow}\, e_2) \rhd e_1 \\
(k, \mathsf{try}\ \Box \,\mathsf{ow}\, e_2) \lhd v_1 &\mapsto k \lhd v_1 \\
k \rhd \mathsf{error} &\mapsto k \blacktriangleleft \\
(k, \mathsf{try}\ \Box \,\mathsf{ow}\, e_2) \blacktriangleleft &\mapsto k \rhd e_2 \\
(k, f) \blacktriangleleft &\mapsto k \blacktriangleleft \qquad (\text{if } f \neq \mathsf{try}\ \Box \,\mathsf{ow}\, e_2)
\end{aligned}$$

When it comes to control features like exceptions, abstract machine specifications are more modular. We did not have to specifically consider or modify the stack frames for addition (or for multiplication, or function application...) in order to introduce exceptions and exception handling; one rule (the last one above) interacts modularly with essentially any "pure" language features. Both the SOS specification and the abstract machine specification were reasonable ways of specifying pure features like addition, but the abstract machine specification better anticipates the addition of control features.

Observation 1. *Different styles of specification can allow different varieties of programming language features to be specified in a modular style.*

Another way of stating this observation is that we must *precisely* anticipate the language features dealing with state and control if SOS-style specifications are our only option. When we pull back and look at a choice between these two specification styles, we only need to *generally* anticipate whether we might need control features like exceptions or first-class continuations; if so, we should use an abstract machine specification.

2.2 Modularity and ambient state

Of course, an abstract-machine-style presentation does not solve the problem with mutable references that we discussed at first! Various attempts have been made to address this problem. The Definition of Standard ML used an ad-hoc convention that, applied to our example, would allow us to write the original $\mathcal{L}_0$ rules while actually meaning that we were writing the state-annotated version of the rules [21, p. 40]. In this case, the convention is less about modular specification and more about not writing hundreds and hundreds of basically-obvious symbols in the language definition.

Mosses's Modular Structural Operational Semantics (MSOS) attempts to formalize and standardize this sort of convention [22]. A transition in MSOS is written as $(e_1 \xrightarrow{X} e_2)$, where X is an open-ended description of the ambient state. By annotating congruence rules with $X = \{...\}$, the MSOS specification of $\mathcal{L}_0$ specifies that the congruence rules simply pass on whatever effects happen in subexpressions. The reduction rule is annotated with $X = \{-\}$ to indicate that it has no effect on the ambient state.

$$\frac{v_1\ \mathsf{value} \quad e_2 \xrightarrow{\{...\}} e_2'}{v_1 + e_2 \xrightarrow{\{...\}} v_1 + e_2'} \qquad \frac{n_1 + n_2 = n_3}{\mathsf{num}(n_1) + \mathsf{num}(n_2) \xrightarrow{\{-\}} \mathsf{num}(n_3)}$$

Reduction rules that actually access some part of the ambient state, such as reduction rules for referencing the store, can mention and manipulate the store as necessary:

$$\frac{\sigma(l) = v}{!\,\mathsf{loc}(l) \xrightarrow{\{\sigma, -\}} v} \qquad \frac{v_2\ \mathsf{value}}{\mathsf{loc}(l)\ \mathsf{gets}\ v_2 \xrightarrow{\{\sigma,\, \sigma'=\sigma[l\,:=\,v_2],\, -\}} v_2}$$

There is a good bit of related work along the lines of MSOS; for example, Implicitly-Modular Structural Operational Semantics (I-MSOS) is an extension of MSOS that partially removes the need for the explicit annotations X [23]. In all cases, however, the goal is to provide a regular notion of ambient state that allows each part of a language specification to leave irrelevant parts of the state implicit and open-ended.

Observation 2. *A framework that provides an open-ended notion of ambient state assists in writing modular programming language specifications.*

2.3 Modularity and the LF context

Logical frameworks in the LF family (such as LF [11], LLF [1], OLF [29], CLF [40], and HLF [30]) have an inherent notion of ambient information provided by the framework's *context*. As an example, consider the following LF specification of the static semantics of $\mathcal{L}_0$ (we use the usual convention in which capital letters are understood to be universally quantified):

$\mathsf{of} : \mathsf{exp} \to \mathsf{tp} \to \mathsf{type}.$
$\mathsf{of/num} : \mathsf{of}\ (\mathsf{num}\ \mathsf{N})\ \mathsf{int}.$
$\mathsf{of/plus} : \mathsf{of}\ \mathsf{E}_1\ \mathsf{int} \to \mathsf{of}\ \mathsf{E}_2\ \mathsf{int} \to \mathsf{of}\ (\mathsf{plus}\ \mathsf{E}_1\ \mathsf{E}_2)\ \mathsf{int}.$

The contents of the LF context are specified independently from the specification of the inference rules. If we declare the LF context to be empty, our specification corresponds to an "on-paper" specification of the language's static semantics that looks like this:

$$\frac{}{\mathsf{num}(n) : \mathsf{int}} \qquad \frac{e_1 : \mathsf{int} \quad e_2 : \mathsf{int}}{e_1 + e_2 : \mathsf{int}}$$

This assumption that there is nothing interesting in the context, frequently called a *closed world assumption*, is not the only option. Another option is to declare that the LF context may include free expression variables x in tandem with an assumption that $x{:}\tau$ for some type τ. In this case, the LF specification written above corresponds to the following on-paper specification:

$$\frac{}{\Gamma, x{:}\tau \vdash x : \tau} \qquad \frac{}{\Gamma \vdash \mathsf{num}(n) : \mathsf{int}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 + e_2 : \mathsf{int}}$$

Language features that incorporate binding require a variable context in the specification of their static semantics; as a result, our original on-paper static semantics could not be composed with an on-paper static semantics of language features like let-expressions and first-class functions. But because the contents of the LF context are open-ended, we *can* combine the LF specifications of the static semantics. The specification we wrote down in LF is open-ended in precisely the way we need it to be, even though the on-paper version is not.

Observation 3. *The LF context provides an open-ended notion of ambient information that enhances the modularity of certain specifications.*

Unfortunately, the open-endedness we describe here is useful for capturing ambient *information* in a static semantics; it is not useful for capturing ambient *state* in a dynamic semantics. The LF context only supports the addition of new assumptions that are persistent and unchanging, but capturing the dynamic semantics of stateful features like mutable references requires adding, modifying, and removing information about the state of the specified system. The solution is, in principle, well understood: logical frameworks based on substructural logics (especially linear logic) can capture stateful change — it is possible to remove facts from a linear logic context. Therefore, assumptions in a linear logic context can be thought of as representing *ephemeral* facts about the state of a system, facts which might change, as opposed to persistent assumptions which can never be removed or modified. Examples of logical frameworks that provide an open-ended notion of ambient state through the use of a linear context include Forum [18], Lolli [14], LLF [1], and CLF [40]. Furthermore, all of these frameworks have been given an operational interpretation as logic programming languages, so specifications are, at least in theory, executable.

2.4 Modularity in substructural operational semantics

These observations motivate the design of substructural operational semantics and, in turn, the design of the logical framework. We can give a initial picture of SSOS specifications at this point by thinking of them as collections of string rewriting rules where, for instance, the rewrite rule $\mathsf{a} \cdot \mathsf{b} \twoheadrightarrow \mathsf{c}$ allows the string $\mathsf{a} \cdot \mathsf{a} \cdot \mathsf{b} \cdot \mathsf{b}$ to transition to the string $\mathsf{a} \cdot \mathsf{c} \cdot \mathsf{b}$.

SSOS specifications resemble the abstract machine-style specifications above, but instead of capturing the entire control stack k as a single piece of syntax, each <u>cont</u>inuation frame f is captured by a token $\mathsf{cont}(f)$ in the string. The token $\mathsf{eval}(e)$ represents an evaluating expression e, and is analogous to the state $(k \rhd e)$ in the abstract machine specification, whereas the token $\mathsf{retn}(v)$ represents a returned value and is analogous to the state $(k \lhd v)$ in the abstract machine specification. The resulting rules are quite similar to the abstract machine rules:

$$\begin{aligned}
\mathsf{eval}(\mathsf{num}(n)) &\twoheadrightarrow \mathsf{retn}(\mathsf{num}(n)) \\
\mathsf{eval}(e_1 + e_2) &\twoheadrightarrow \mathsf{cont}(\Box + e_2) \cdot \mathsf{eval}(e_1) \\
\mathsf{cont}(\Box + e_2) \cdot \mathsf{retn}(v_1) &\twoheadrightarrow \mathsf{cont}(v_1 + \Box) \cdot \mathsf{eval}(e_2) \\
\mathsf{cont}(\mathsf{num}(n_1) + \Box) \cdot \mathsf{retn}(\mathsf{num}(n_2)) &\twoheadrightarrow \mathsf{retn}(\mathsf{num}(n_3)) \qquad \text{(if } n_1 + n_2 = n_3\text{)}
\end{aligned}$$

We can then give modular specifications of exceptions and exception handling much as we did in the abstract machine for control, and the non-local nature of the specifications means that we can add state. To give a very simple example that does not generalize well, we can modularly extend the language with a single counter; evaluating the expression get gets the current value of the counter, and evaluating the expression inc gets the current value of the counter and then increments it by one. The value of the counter is stored in a token $\mathsf{counter}(n)$ that appears to the right of the $\mathsf{eval}(e)$ or $\mathsf{retn}(v)$ tokens (whereas the stack grows off to the left).

$$\begin{aligned}
\mathsf{eval}(\mathsf{get}) \cdot \mathsf{counter}(n) &\twoheadrightarrow \mathsf{retn}(\mathsf{num}(n)) \cdot \mathsf{counter}(n) \\
\mathsf{eval}(\mathsf{inc}) \cdot \mathsf{counter}(n) &\twoheadrightarrow \mathsf{retn}(\mathsf{num}(n)) \cdot \mathsf{counter}(n+1)
\end{aligned}$$

Just as the rules need only reference the parts of the control stack that they modify, the rules only need to mention the counter if they access or modify that counter in some way. Again, this is a terrifically non-generalizable way of extending the system with state, because the use of ambient state is not "open-ended" — where would we put the *second* counter? Nevertheless, it is a specification of a stateful feature that can be composed modularly with the specification of addition.

After we motivate and present the logical framework we plan to use in the next two sections, we will return to this style of specification and see how it can be used to give modular specifications of exceptions and parallel evaluation.

3 Logics of deduction and transition

The aforementioned logical frameworks that provide an open-ended notion of ambient state through the use of substructural logics — Forum, CLF, etc. — can be difficult to use for certain types of reasoning about properties of systems specified in the framework. One source of difficulty is that many properties of specified systems are naturally described as properties of the *partial* proofs that arise during proof search, but in all of these frameworks it is more natural to reason about properties of *complete* proofs — put another way, SSOS-like specifications are properly understood *small-step* specifications, but the logical frameworks only make it easy to reason about *big-step* properties of those specifications. In this section, we discuss a solution to this mismatch: we define linear logic as a state transition system, at which point it is natural to talk about sequences of transitions (which do, in fact, correspond to partial proofs in a sequent calculus) as first-class entities. The only other solutions that I am aware of are based on Ludics, a presentation of logic that treats proofs with missing "leaves" as objects of consideration [10].

Underlying logical frameworks like LF and LLF is a particular justification of logic, dating back to Martin-Löf's 1983 Siena Lectures [15], which holds that the meaning of a proposition is captured by what counts as a verification of that proposition. In Section 3.1 I briefly revisit Martin-Löf's verificationist meaning-theory and how it relates to the use of canonical forms to adequately capture deductive systems.

There is another presentation of logic — first outlined by Pfenning [25] and developed further in Section 3.2 — which underlies a line of work on using forward-chaining in substructural logics for logic programming and logical specification [36, 37, 28]. Substructural contexts are treated as descriptions of the ephemeral state of a system, and the meaning of a proposition is captured by the state transitions it generates.

The integration of these two ways of understanding logic requires care, as a naïve approach for combining the two paradigms will destroy desirable properties of both the canonical-forms-based framework and the state-transition-based framework. In Section 3.3, I explain my proposed approach, which is based on the observations of adjoint logic [31].

3.1 A canonical-forms-based view of logic

Existing work on the proof theory of logical frameworks, including the LF family of logical frameworks [11], is based in part on a *verificationist* philosophy described in Martin-Löf's Siena Lectures [15]. A verificationist meaning-theory provides that the meaning of a proposition is precisely captured by its introduction rules — that is, by the ways in which it may be verified. A proposition $A \wedge B$ is true, then, precisely if A is true and B is true, and a proposition $A \supset B$ is true precisely if we can verify B given an additional assumption that A is true.

The elimination rules, on the other hand, must be *justified* with respect to the introduction rules. We justify the elimination rules for a connective in part by checking their *local soundness* and *local completeness* [26]. Local soundness ensures that the elimination rules are not too strong: if an introduction rule is immediately followed by an elimination rule, the premise gives us all the necessary evidence for the conclusion.

$$\dfrac{\dfrac{\begin{matrix}\mathcal{D}_1\\ A\ \mathit{true}\end{matrix}\quad \begin{matrix}\mathcal{D}_2\\ B\ \mathit{true}\end{matrix}}{A \wedge B\ \mathit{true}}\wedge I}{A\ \mathit{true}}\wedge E_1 \qquad \Rightarrow_R \qquad \begin{matrix}\mathcal{D}_1\\ A\ \mathit{true}\end{matrix}$$

Local completeness, on the other hand, ensures that the elimination rules are not too weak: given a derivation of the compound connective, we can use elimination rules to get all the evidence necessary to reapply the introduction rule:

$$\begin{matrix}\mathcal{D}\\ A \wedge B\ \mathit{true}\end{matrix} \qquad \Rightarrow_R \qquad \dfrac{\dfrac{\begin{matrix}\mathcal{D}\\ A \wedge B\ \mathit{true}\end{matrix}}{A\ \mathit{true}}\wedge E_1 \quad \dfrac{\begin{matrix}\mathcal{D}\\ A \wedge B\ \mathit{true}\end{matrix}}{B\ \mathit{true}}\wedge E_2}{A \wedge B\ \mathit{true}}\wedge I$$

3.1.1 Verifications and canonical forms

An examination of the requirement that introduction rules *precisely* define the connectives leads to a restricted set of proofs called *verifications*. If there is a proof of the truth of $A \wedge B$, then we don't know a great deal about its proof. Maybe it looks like this!

$$\dfrac{\dfrac{\dfrac{\begin{matrix}\vdots\\ D \wedge (A \wedge B)\ \mathit{true}\end{matrix}}{A \wedge B\ \mathit{true}}\wedge E_2 \quad \begin{matrix}\vdots\\ C\ \mathit{true}\end{matrix}}{(A \wedge B) \wedge C\ \mathit{true}}\wedge I}{A \wedge B\ \mathit{true}}\wedge E_1$$

On the other hand, if we have a *verification* of $A \wedge B$ then we know that the last step in its proof combines two (smaller) verifications, one of which is a verification of A and the other of which is a verification of B. We consider the verifications to be the canonical proofs of a proposition — when we introduce proof terms, verifications correspond to a restricted set of proofs called the *canonical forms*.

3.1.2 Definitions, atomic propositions, and adequacy

The canonical forms are interesting from the point of view of logical frameworks because they allow us to capture a representative of a derivation as a proof term within the logic. To use a standard example, we can introduce a new atomic proposition ($\mathsf{add\,N\,M\,P}$) that is defined by the two constants $\mathsf{add/z}$ and $\mathsf{add/s}$. We continue to use the convention that identifiers beginning with capital letters represent variables that are implicitly universally quantified.

```
z : nat.
s : nat → nat.

add : nat → nat → nat → type.
add/z : add z N N.
add/s : add N M P → add (s N) M (s P).
```

Given this signature, we would like to say that we also know what counts as a verification of the proposition $\mathsf{add\,(s\,z)\,(s\,z)\,(s(s\,z))}$ — in particular, that only one thing will do: a use of $\mathsf{add/s}$ and a verification of $\mathsf{add\,z\,(s\,z)\,(s\,z)}$. But this is not the case! We can also verify a proposition $\mathsf{add\,(s\,z)\,(s\,z)\,(s(s\,z))}$ by using an *assumption* of the same proposition, or by using an assumption of the form $\mathsf{add\,N\,M\,P} \to \mathsf{add\,N\,M\,P}$ and a verification of $\mathsf{add\,(s\,z)\,(s\,z)\,(s(s\,z))}$, or by using an assumption of the form $\mathsf{add\,N\,(s\,M)\,P} \to \mathsf{add\,N\,M\,P}$ and a verification of $\mathsf{add\,(s\,z)\,(s(s\,z))\,(s(s\,z))}$... The point is that the very open-endedness of the LF context that we pointed out in Observation 3 is preventing us from making the claim that we know something specific about what counts as a verification of the proposition $\mathsf{add\,(s\,z)\,(s\,z)\,(s(s\,z))}$.

However, it is intuitively reasonable enough to require that our current assumptions include neither assumptions of the form $\mathsf{add\,N\,M\,P}$ nor assumptions that can be immediately used to prove $\mathsf{add\,N\,M\,P}$ (which is effectively a *closed world assumption*). Under this requirement, our intuition for what counts as a verification of the proposition $\mathsf{add\,(s\,z)\,(s\,z)\,(s(s\,z))}$ is correct. Considerations such as these lead to a formal notion of *adequate encodings*, which in this case means that there is a bijection between terms of the type $\mathsf{add\,N\,M\,P}$ and derivations of $\mathsf{N + M = P}$ as defined by the following deductive system:

$$\frac{\mathsf{N + M = P}}{\mathsf{s\,N + M = s\,P}}\ \mathsf{add/s} \qquad \frac{}{\mathsf{z + N = N}}\ \mathsf{add/z}$$

There is, as the example of a static semantics from Section 2 illustrates, tension between considerations of modularity (which dictate that the form of the context should be as open-ended as possible) and adequacy (which dictate that the form of the context should be as fully specified as possible). However, experience shows that, by incorporating concepts such as subordination [39], this tension is unproblematic in practice.

The above discussion is informal, and adequacy is given a much more careful treatment elsewhere [11, 12]. The point of the preceding discussion is to introduce the use of canonical forms for specifying inductively defined systems and to underscore the necessity of imposing some sort of regularity constraint on the context.

3.2 A state-transition-based view of logic

The presentation of logic as verifications and canonical forms treats atomic propositions and their proofs as the primary objects of consideration. We are interested in treating the *state of a system* and *transitions between states* as the primary objects of consideration. The result has strong similarities to sequent calculus presentations of linear logic, which is unsurprising: previous work has universally started with a sequent presentation of logic and then "read off" a state transition system. Cervesato and Scedrov's multiset rewriting language ω is a prime example of this process; an introduction to ω and a review of related work in the area can be found in their article [3].

We will treat the state of a specified system as being defined by a set of *ephemeral facts* about that state; we think of these ephemeral facts as *resources*. Therefore, states are multisets of ephemeral facts $(A\ res)$; the empty multiset is written as $(\cdot)$ and multiset union is written as (Δ_1, Δ_2). We define the meaning of propositions by the effect they have on resources. The proposition $A \,\&\, B$ represents a resource that can provide *either* a resource A or a resource B, and $\top$ is an unusable resource, so the resource $(\top\ res)$ will always stick around uselessly. The proposition $\forall x{:}\tau.A(x)$ represents a resource that can provide the resource $A(t)$ for any term t with type τ (we assume there is some signature Σ defining base types and constants).

$(\&_{T1})$	$A \,\&\, B\ res \rightsquigarrow A\ res$	
$(\&_{T2})$	$A \,\&\, B\ res \rightsquigarrow B\ res$	
(*no rule for* $\top$)		
$(\forall_T)$	$\forall x{:}\tau.A(x)\ res \rightsquigarrow A(t)\ res$	if $\quad \Sigma \vdash t : \tau$

Linear implication represents a sort of test. It simplifies matters if, initially, we only consider linear implication $(Q \multimap A)$ where Q is an atomic proposition; it means that we can consume Q to produce A.

$$(\multimap_{T1}) \qquad Q\ res,\ Q \multimap A\ res \rightsquigarrow A\ res$$

The transition relation is, in general, $\Delta_1 \rightsquigarrow \Delta_2$, and there is a congruence or frame property that says a transition can happen to any part of the state.

$$\frac{\Delta_1 \rightsquigarrow \Delta_1'}{\Delta_1, \Delta_2 \rightsquigarrow \Delta_1', \Delta_2}\ \textit{frame}$$

We also may allow the signature Σ to mention certain resources as being valid or unrestrictedly available; in this case we can spontaneously generate new copies of the resource:

$$(\textit{copy}) \qquad \cdot \rightsquigarrow A\ res \qquad \text{if} \qquad (A\ valid) \in \Sigma$$

This is a very brief version of this story, and in its current telling it can be understood as a subset of Cervesato and Scedrov's ω multiset rewriting language [3]. However, that language and other similar languages take the notion of derivability in a linear logic sequent calculus as primary and derive a transition semantics from open proofs. The story we have told here starts with transitions and generalizes to all of first-order linear logic save for additive disjunction $(A \oplus B)$ and its unit $\mathbf{0}$.[1] I believe there is an inherent value in reducing the gap between the specification of linear logic and the specification of systems within that logic, and having transitions as a first-class notion within the logical framework appears to clarify many issues related to reasoning about specifications.

[1]Furthermore, I have made substantial, though unfinished, progress on incorporating these connectives into a transition-based presentations as well.

3.2.1 Canonical transitions

Our motivation for describing logic as a transition system is to make it more natural to use logic for the specification of transition systems. An example of a very simple transition system that we would hope to represent is the following bare-bones asynchronous pi-calculus represented by the usual structural congruences and the following reduction rule:

$$c(x).p \parallel c\langle v\rangle \mapsto p[v/x]$$

To represent this system in logic, we can show that the above transition system corresponds exactly to the following signature where there are two sorts of terms, channels chan and processes $\mathsf{process}$. A process can be either asynchronous send along a channel ($\mathsf{send}\,\mathsf{C}\,\mathsf{V}$, i.e. $c\langle v\rangle$) or a receive along a channel ($\mathsf{recv}\,\mathsf{C}\,(\lambda\mathsf{x}.\,\mathsf{P}\,\mathsf{x})$, i.e. $c(x).p$). The resource ($\mathsf{proc}\,\mathsf{P}$) represents the active process P.

$\mathsf{send} : \mathsf{chan} \to \mathsf{chan} \to \mathsf{process}.$
$\mathsf{recv} : \mathsf{chan} \to (\mathsf{chan} \to \mathsf{process}) \to \mathsf{process}.$

$\mathsf{proc} : \mathsf{process} \to \mathsf{type}.$

$\mathsf{synch} : \mathsf{proc}(\mathsf{recv}\,\mathsf{C}\,(\lambda\mathsf{x}.\,\mathsf{P}\,\mathsf{x})) \multimap \mathsf{proc}(\mathsf{send}\,\mathsf{C}\,\mathsf{V}) \multimap \mathsf{proc}(\mathsf{P}\,\mathsf{V}).$

What do we mean by *corresponds exactly*? Perhaps something like weak bisimulation or strong bisimulation would do, but we really want something akin to the adequacy properties discussed earlier. Adequacy in this setting means that there should be a bijective correspondence between states in linear logic and states in the simple process calculus; furthermore, transitions in both systems must respect this correspondence — any single transition that is made in the logic can be mapped onto a single transition in the process calculus, and vice-versa.

This, certainly, is not the case in system we have described so far. Take the synchronization transition $(\mathsf{a}(x).x\langle x\rangle \parallel \mathsf{a}\langle\mathsf{b}\rangle) \mapsto \mathsf{b}\langle\mathsf{b}\rangle$ in the simple process calculus. We can say that $\mathsf{a}(x).x\langle x\rangle$ is represented as the atomic proposition $\mathsf{proc}(\mathsf{recv}\,\mathsf{a}\,\lambda\mathsf{x}.\,\mathsf{send}\,\mathsf{x}\,\mathsf{x})$ and that $\mathsf{a}\langle\mathsf{b}\rangle$ is represented as the atomic proposition $\mathsf{proc}(\mathsf{send}\,\mathsf{a}\,\mathsf{b})$. However, because the synch rule has two premises, fully applying it will take at least two transitions in linear logic (five if we count the three transitions necessary due to the implicit quantification of C, P, and V in the synch rule).

The analogue to canonical forms and verifications in this setting addresses this problem. A *focused* presentation of our linear logic allows us to restrict attention to transitions that atomically copy a proposition from the signature, instantiate all of its variables, and satisfy all of its premises. Then we can say that the synch rule defined in the signature corresponds to the following *synthetic transition*:

$$(\mathsf{synch}) \qquad \mathsf{proc}(\mathsf{send}\,\mathsf{C}\,\mathsf{V})\ \mathit{res},\ \mathsf{proc}(\mathsf{recv}\,\mathsf{C}\,\lambda x.\,\mathsf{P}\,x)\ \mathit{res} \rightsquigarrow \mathsf{proc}(\mathsf{P}\,\mathsf{V})\ \mathit{res}$$

The synthetic transition associated with synch precisely captures the transition $c(x).p \parallel c\langle v\rangle \mapsto p[v/x]$ in the process calculus, so transitions in the encoded system are adequately represented by transitions in the logical framework. In order to ensure that states are adequately represented in the framework, we again must impose a constraint on the regular structure of the context, namely that it only consists of atomic resources of the form ($\mathsf{proc}\,\mathsf{P}$).

3.3 Combining transitions and canonical forms

In Section 3.1, we saw that deductive systems, such as the addition of unary natural numbers, can be adequately represented in a logical framework based on verifications and canonical forms. In Section 3.2 we saw that transition systems can be characterized by an interpretation of linear logic that treats the logic as a transition system. However, a transition system frequently needs to refer to some notion of inductive definition as (essentially) a side condition. As a slightly contrived example, consider a transition system where each resource contains a natural number and where two distinct numbers can be added together:

$$\mathsf{num}\;\mathsf{N}_1\;\mathit{res}, \mathsf{num}\;\mathsf{N}_2\;\mathit{res} \leadsto \mathsf{num}(\mathsf{N}_3)\;\mathit{res} \qquad \textit{(if } \mathsf{add}\;\mathsf{N}_1\;\mathsf{N}_2\;\mathsf{N}_3 \textit{ can be derived)}$$

Recall that, in Section 2 when we presented the psedo-SSOS specification of a language with addition, we also used a side condition of this form, but now we have made it explicit that the *content* of the side condition is an inductively defined judgment.

3.3.1 Introduction to adjoint logic

In the next section, we will take a fragment of LF — a canonical forms-based logic for representing syntax and derivations (which are used to express side conditions) — with a transition-based presentation of ordered linear logic that is used to represent evolving systems. The basis for this combination is adopted from Reed's adjoint logic [31], in which two syntactically distinct logics can be connected through unary connectives U and F. In the logical framework presented in the next section, these two logics will be a fragment of LF and a transition-based ordered linear logic, but in this section, we will consider a logic that combines persistent and linear logic:

Persistent propositions	$P, Q, R ::= p \mid UA \mid P \supset Q$
Linear/ephemeral propositions	$A, B, C ::= a \mid FP \mid A \multimap B$

Following Reed (and in order to match up with the naturally sequent-calculus-oriented state transition presentation of logic) we will consider adjoint logic as a sequent calculus. Persistent contexts Γ hold persistent facts and linear contexts Δ hold ephemeral resources; the sequent $\Gamma \vdash P$ characterizes proofs of persistent propositions and the sequent $\Gamma; \Delta \vdash A$ characterizes proofs of linear (ephemeral) propositions. Reading the rules in a bottom-up fashion, the linear goal FP is only true if there are no linear hypotheses and the persistent proposition P is provable; the linear hypothesis FP can be replaced by the persistent hypothesis P.

$$\frac{}{\Gamma; a \vdash a} \quad \frac{\Gamma \vdash P}{\Gamma; \cdot \vdash FP} \quad \frac{\Gamma, P; \Delta \vdash C}{\Gamma; \Delta, FP \vdash C} \quad \frac{\Gamma; \Delta, A \vdash B}{\Gamma; \Delta \vdash A \multimap B} \quad \frac{\Gamma; \Delta_A \vdash A \quad \Gamma; \Delta, B \vdash C}{\Gamma; \Delta, \Delta_A, A \multimap B \vdash C}$$

The description of persistent connectives below has one quirk: there are two left rules for $(P \supset Q)$, one where the ultimate goal is to prove a persistent proposition and one where the ultimate goal is to prove a linear proposition. This is not a quirk that is unique to adjoint logic; it is the same issue that requires Pfenning and Davies' judgmental S4 to have two elimination rules for $\Box A$ [26], for instance.[2]

$$\frac{p \in \Gamma}{\Gamma \vdash p} \quad \frac{\Gamma; \cdot \vdash A}{\Gamma \vdash UA} \quad \frac{(UA) \in \Gamma \quad \Gamma; \Delta, A \vdash C}{\Gamma; \Delta \vdash C} \quad \frac{\Gamma, P \vdash Q}{\Gamma \vdash P \supset Q}$$

$$\frac{(P \supset Q) \in \Gamma \quad \Gamma \vdash P \quad \Gamma, Q \vdash R}{\Gamma \vdash R} \quad \frac{(P \supset Q) \in \Gamma \quad \Gamma \vdash P \quad \Gamma, Q; \Delta \vdash C}{\Gamma; \Delta \vdash C}$$

As Reed notes, if we erase every persistent proposition save for UA, the resulting logic is equivalent to linear logic with exponentials, where $!A \equiv FUA$. Similarly, if we erase every linear proposition save for FP, the resulting logic is equivalent to the Pfenning-Davies reconstruction of lax logic where $\bigcirc P \equiv UFP$.

[2]Another way to look at this "quirk" is that the *real* rule is parametric in the conclusion, able to prove either $A\ \mathit{res}$ or $P\ \mathit{valid}$, and we just give the two instances of the real rule here.

3.3.2 Combining transitions and canonical forms with adjoint logic

My approach in the following section is to use adjoint logic to present a framework that connects a persistent logical framework of canonical forms and a substructural logical framework of transitions. The result, as we have just seen, combines elements of substructural logic with elements of lax logic, so it is not surprising that the end result turns out to be very similar to the CLF logical framework that (in a rather different way) also segregates transitions and deductions using a combination of lax and substructural logic.[3] It is even closer to the *semantic effects* fragment of CLF, which requires the canonical forms-based fragment of the language to forgo any reference to the state-transition-based fragment [5].

The framework I present in the next section has no way of incorporating transitions within canonical forms; syntactically, this is achieved by removing the proposition UA from the framework. This exclusion decreases the expressiveness of the framework relative to CLF, but brings it closer in line to the semantic effects fragment. The connective F is then the only connection between the two systems, and since it retains much of the character of the linear logic exponential, it is written as $!P$ even though it actually acts only as the "first half" of the exponential.

4 A logical framework for evolving systems

In the previous section, we considered the use of logic to describe both deductive systems and state-transition systems, and our immediate goal is to use a logical framework based on these principles to specify the operational semantics of programming languages with state and concurrency. In this section we will present that logical framework, which based on a substructural logic that includes both ordered and linear propositions.

In addition to CLF, the design synthesizes ideas from Polakow et al.'s ordered logic framework [29], Reed's adjoint logic [31], and Zeilberger's polarized higher-order focusing [41]. There is no fundamental reason why the canonical forms fragment of the framework described here cannot be the full dependently typed logical framework LF [11, 12]; however, it is convenient for our current purposes to just consider a restricted fragment of LF.

4.1 Representing terms

We have already been using LF-like representations of terms: both natural numbers and process-calculus terms can be adequately represented as canonical forms in a simply-typed lambda calculus under the following signature.

```
nat : type.              process : type.
z : nat.                 chan : type.
s : nat → nat.           send : chan → chan → process.
                         recv : chan → (chan → process) → process.
```

A term type τ is either an atomic term type a defined by the signature (like nat, process, and chan) or an implication $\tau \to \tau$. The signature also defines term constants (like z, s, send, and recv) which are defined to have some type τ. Canonical terms are the η-long, β-normal terms of the simply-typed lambda calculus

[3] I believe that adjoint logic provides a better explanation than lax logic for how CLF is actually used and thought about, but I should re-emphasize that the critical difference between CLF and the framework presented in the next section — the reason I cannot use CLF directly — is that CLF does not treat individual state transitions and sequences of state transitions as objects of consideration.

using constants drawn from the signature: the natural number 3 is represented as $\mathsf{s}(\mathsf{s}(\mathsf{s}\,\mathsf{z}))$ and the process $\mathsf{a}(x).x\langle x\rangle$ is represented as the canonical term $(\mathsf{recv}\,\mathsf{a}\,\lambda\mathsf{x}.\,\mathsf{send}\,\mathsf{x}\,\mathsf{x})$.

4.2 Judgments as types

An organizing principle of LF-like frameworks is the representation of *judgments as types* — in the case of the aforementioned example of addition, the three-place judgment $n_1 + n_2 = n_3$ is represented by the type family add of kind $\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat} \to \mathsf{type}$, and for any specific n_1, n_2, and n_3 we represent derivations of $\mathsf{n}_1 + \mathsf{n}_2 = \mathsf{n}_3$ as canonical proof terms of type $(\mathsf{add}\,\mathsf{n}_1\,\mathsf{n}_2\,\mathsf{n}_3)$ in the following signature:

$\mathsf{add} : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat} \to \mathsf{type}.$
$\mathsf{add/z} : \Pi\mathsf{N{:}nat}.\,\mathsf{addzNN}.$
$\mathsf{add/s} : \Pi\mathsf{N{:}nat}.\,\Pi\mathsf{M{:}nat}.\,\Pi\mathsf{P{:}nat}.\,\mathsf{add\,N\,M\,P} \to \mathsf{add\,(s\,N)\,M\,(s\,P)}.$

In this signature, the proof term $(\mathsf{add/s}\,\mathsf{z}\,(\mathsf{s}\,\mathsf{z})\,(\mathsf{s}\,\mathsf{z})\,(\mathsf{add/z}\,(\mathsf{s}\,\mathsf{z})))$ acts as a representative of this derivation:

$$\dfrac{\dfrac{}{\mathsf{z} + \mathsf{s}\,\mathsf{z} = \mathsf{s}\,\mathsf{z}}\ \mathsf{add/z}}{\mathsf{s}\,\mathsf{z} + \mathsf{s}\,\mathsf{z} = \mathsf{s}(\mathsf{s}\,\mathsf{z})}\ \mathsf{add/s}$$

More generally, type families are declared as $(a : \tau_1 \to \ldots \to \tau_n \to \mathsf{type})$ in the signature; atomic types in the type family a are written as $a\,t_1 \ldots t_n$, where each of the arguments t_i has type τ_i. Types A are either atomic types, implications $A \to B$, or universal quantifications over terms $\Pi x{:}\tau.A$. We usually leave universal quantification implicit in the signature, which allows us to write rules as we did earlier:

$\mathsf{add/z} : \mathsf{add\,z\,N\,N}.$
$\mathsf{add/s} : \mathsf{add\,N\,M\,P} \to \mathsf{add\,(s\,N)\,M\,(s\,P)}.$

Furthermore, we leave the instantiation of implicit quantifiers implicit as well, which allows us to write the proof term corresponding to the above derivation more concisely as $(\mathsf{add/s}\,\mathsf{add/z})$.

4.3 States as contexts

In the transition-based fragment of the logic, the states of the systems we intend to specify are represented as collections of atomic propositions. In sequent calculus-based logical frameworks, these collections correspond to *contexts*; we adopt this terminology.

Ephemeral contexts Linear logic and other substructural logics are useful for describing systems that change over time; part of the reason for this is that a linear resource is effectively an *ephemeral* fact about the state of a system — it describes something that is true in the current state of the system but which may not be true in a future state of the system. Our framework considers two kinds of ephemeral facts: *linear* facts (insensitive to their position relative to other facts) and *ordered* facts (sensitive to their position relative to other ordered facts).

We incorporate ephemeral facts into our framework by introducing two different kinds of type family: linear type families (where we write type_l — l for linear — instead of type in the type family declaration) and ordered type families (where we write type_o — o for ordered — instead of type in the type family declaration). This effectively enforces a syntactic separation between those persistent, linear, and ordered atomic propositions; the need for this syntactic separation has been noted in previous work [36, 37] but was only given a logical interpretation in [28].

An ephemeral context (written Δ) is syntactically a list of linear and ordered types, but we consider contexts to be equivalent if they can be made syntactically equal by reordering linear propositions. As an example, if a_l and b_l are linear atomic propositions and c_o and d_o are ordered atomic propositions, then the contexts (a_l, b_l, c_o, d_o), (c_o, d_o, b_l, a_l), and (c_o, a_l, d_o, b_l) are all equivalent, but they are not equivalent to (a_l, b_l, d_o, c_o) because the ordered propositions c_o and d_o appear in a different order. We write this equivalence on contexts as $\Delta \approx \Delta'$.

We can use ordered resources to represent the *concrete* syntax of the language $\mathcal{L}_0$ from Section 2 by defining an ordered atomic proposition for each syntactic token:

```
+ : type_o.
num : nat → type_o.
```

The proposition $\mathsf{+}$ represents the addition token, and $\mathsf{num}\,\mathsf{N}$ (where N is a term of type nat) represents a token for the natural number N. While we cannot yet actually describe parsing, we can intuitively say that the ephemeral context $(\mathsf{num}(\mathsf{s}\,\mathsf{z})), \mathsf{+}, (\mathsf{num}\,\mathsf{z})$ corresponds to the $\mathcal{L}_0$ program $1+0$, the ephemeral context $(\mathsf{num}(\mathsf{s}\,\mathsf{z})), \mathsf{+}, (\mathsf{num}\,\mathsf{z}), \mathsf{+}, (\mathsf{num}(\mathsf{s}(\mathsf{s}\,\mathsf{z})))$ is ambiguous and corresponds to either the $\mathcal{L}_0$ program $(1+0)+2$ or the $\mathcal{L}_0$ program $1+(0+2)$, and the ephemeral context $\mathsf{+}, \mathsf{+}, (\mathsf{num}(\mathsf{s}\,\mathsf{z}))$ is not syntactically well-formed and does not correspond to any $\mathcal{L}_0$ program.

As in sequent calculus presentations of a logic, we annotate the propositions in a context with unique variable names. Therefore, the context corresponding to the first $\mathcal{L}_0$ program above could also have been been written as $(x_1 : \mathsf{num}(\mathsf{s}\,\mathsf{z}), x_2 : \mathsf{+}, x_3 : \mathsf{num}\,\mathsf{z})$. We further follow the convention of sequent calculus presentations of logic by often not mentioning these variable names when it is possible to reconstruct them.

Persistent contexts In addition to ephemeral state, is important to allow systems to have *persistent* state, which includes both dynamically introduced parameters and facts that, once true, necessarily stay true.[4] We write free term parameters as $x : \tau$ and persistent assumptions as $x : A$, where A is a persistent type. A collection of these assumptions is a persistent context, written as Γ, and we treat all contexts containing the same terms and assumptions as equivalent, writing this equivalence as $\Gamma \approx \Gamma'$.

4.4 Substitutions

The connection point between contexts and terms is the definition of *substitutions*. A substitution σ is a term which gives evidence that, given the persistent facts Γ and ephemeral resources Δ, we can model or represent the state described by the persistent facts Γ' and ephemeral facts Δ'. Alternatively, we can think of Γ' and Δ' as "context-shaped holes," in which case a substitution is evidence that Γ and Δ "fit in the holes."

The way a substitution gives this evidence is by providing a resource (designated by its label y) for every for every resource in Δ', providing a proof term $\mathcal{D}$ that is defined in the context Γ for every fact $x : A$ in Γ', and providing a term t that is defined in the context Γ for every $x : \tau$ in Γ'.[5] Syntactically, a substitution is described by the following grammar:

$$\sigma ::= [] \mid \sigma, t/x \mid \sigma, \mathcal{D}/x \mid \sigma, y/x$$

[4]Parameters can, for instance, be used to model the channels in the process calculus from the previous section, or to model abstract heap locations in a specification of ML-style references. We need to be able to dynamically introduce new parameters in order to represent producing a new channel or allocating a new location on the heap.

[5]In an appropriate dependently-typed setting, term types τ and types A are members of the same syntactic class, as are terms t and proof terms $\mathcal{D}$.

The definition of substitution typing — $\Gamma; \Delta \vdash_\Sigma \sigma : \Gamma'; \Delta'$ — captures the judgment that σ is a witness to the fact that the state $\Gamma; \Delta$ can model or represent the state Γ', Δ' (under the signature Σ). The definition relies on two other judgments, $\Gamma \vdash_\Sigma t : \tau$ and $\Gamma \vdash_\Sigma \mathcal{D} : A$, which describe well-typed terms and well-typed proof terms, respectively. It also relies on the operation $A[\sigma]$, the application of a substitution to a type. Because we are using only a restricted fragment of LF, only the term components (t/x) of substitutions actually matter for this substitution.

$$\frac{}{\Gamma; \cdot \vdash_\Sigma [] : \cdot; \cdot} \qquad \frac{\Gamma; \cdot \vdash_\Sigma \sigma : \Gamma'; \cdot \qquad \Gamma \vdash_\Sigma t : \tau}{\Gamma; \cdot \vdash_\Sigma (\sigma, t/x) : (\Gamma', x : \tau); \cdot} \qquad \frac{\Gamma; \cdot \vdash_\Sigma \sigma : \Gamma'; \cdot \qquad \Gamma \vdash_\Sigma \mathcal{D} : A[\sigma]}{\Gamma; \cdot \vdash_\Sigma (\sigma, \mathcal{D}/x) : (\Gamma', x : A); \cdot}$$

$$\frac{\Delta \approx (\Delta'', y : Q') \qquad Q' = Q[\sigma] \qquad \Gamma; \Delta'' \vdash_\Sigma \sigma : \Gamma'; \Delta'}{\Gamma; \Delta \vdash_\Sigma (\sigma, y/x) : \Gamma'; (\Delta', x : Q)}$$

4.5 Positive types as patterns

In the discussion of a transition-based view of logic in Section 3.2, the only connectives we discussed were those that were defined in terms of their effect on the context; these are the connectives that make up the so-called *negative* propositions. Another family of connectives, which make up the so-called *positive* connectives, are more naturally defined by the contexts they describe. The pattern judgment — $\Gamma; \Delta \Vdash p :: A^+$ — expresses that p is the evidence that A^+ describes the context $\Gamma; \Delta$. The most natural way to think about Γ and Δ, which should appropriately be seen as an *output* of the judgment, is that it is describing a context-shaped hole that will be filled by a substitution in order to prove A^+.

Ordered conjunction $(A^+ \cdot B^+)$ describes a context containing, to the left, the context described by A^+ and then, to the right, the context described by B^+.

$$\frac{\Gamma_1; \Delta_1 \Vdash p_1 :: A^+ \quad \Gamma_2; \Delta_2 \Vdash p_2 :: B^+}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 \Vdash (p_1 \cdot p_2) :: (A^+ \cdot B^+)}$$

Existential quantification $(\exists x{:}\tau.A^+)$ describes a context described by A^+, plus a variable $x : \tau$. If we are thinking of the output contexts as context-shaped holes, then x is a hole that must be filled by a term of type τ. The rule below should make it clear that pattern judgments are fundamentally different from typing judgments, where we would expect the x to appear in the premise, not in the conclusion.

$$\frac{\Gamma; \Delta \Vdash p :: A^+}{x : \tau, \Gamma; \Delta \Vdash (x.p) :: (\exists x{:}\tau.A^+)}$$

Finally, $\mathbf{1}$ describes a context containing no ephemeral propositions, and we treat atomic propositions as positive as well: Q (where Q is an ordered atomic proposition) describes a context with just one ordered atomic proposition Q, $¡Q$ (where Q is a linear atomic proposition) describes a context with just one linear atomic proposition Q, and $!A$ (where A is a persistent proposition) describes a context with no ephemeral propositions where A is true.

$$\frac{}{\cdot; \cdot \Vdash () :: \mathbf{1}} \qquad \frac{}{x : A; \cdot \Vdash x :: !A} \qquad \frac{}{\cdot; x : Q \Vdash x :: ¡Q} \qquad \frac{}{\cdot; x : Q \Vdash x :: Q}$$

Example. Because we may think of the pattern judgment $\Gamma; \Delta \Vdash p :: A^+$ as producing context-shape holes Γ and Δ, looking at a pattern (which produces context-shape holes) together with a substitution (which provides evidence that those holes can be filled) is illustrative. If we want to show that the ephemeral context $(x_1 : \mathsf{num}(\mathsf{s}\,\mathsf{z}), x_2 : \mathsf{+}, x_3 : \mathsf{num}\,\mathsf{z})$ is described by the positive proposition $\exists n.\,(\mathsf{num}\,n \cdot \mathsf{+}) \cdot (\mathsf{num}\,\mathsf{z} \cdot {!\mathsf{add}}\,n\,n\,(\mathsf{s}(\mathsf{s}\,\mathsf{z})))$, then we first find the pattern associated with that proposition (we write n instead of num for brevity):

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\cdot; y_1 : \mathsf{n}\,n \Vdash y_1 :: \mathsf{n}\,n} \quad \cfrac{}{\cdot; y_2 : \mathsf{+} \Vdash y_2 :: \mathsf{+}}}{\cdot; y_1 : \mathsf{n}\,n, y_2 : \mathsf{+} \Vdash y_1 \cdot y_2 :: (\mathsf{n}\,n \cdot \mathsf{+})} \quad \cfrac{\cfrac{}{\cdot; y_3 : \mathsf{n}\,\mathsf{z} \Vdash y_3 :: \mathsf{n}\,\mathsf{z}} \quad \cfrac{}{y_4 : \mathsf{add}\,n\,n\,(\mathsf{s}(\mathsf{s}\,\mathsf{z})); \cdot \Vdash y_4 :: {!\mathsf{add}}\,n\,n\,(\mathsf{s}(\mathsf{s}\,\mathsf{z}))}}{y_4 : \mathsf{add}\,n\,n\,(\mathsf{s}(\mathsf{s}\,\mathsf{z})); y_3 : \mathsf{n}\,\mathsf{z} \Vdash (y_3 \cdot y_4) :: (\mathsf{n}\,\mathsf{z} \cdot {!\mathsf{add}}\,n\,n\,(\mathsf{s}(\mathsf{s}\,\mathsf{z})))}}{y_4 : \mathsf{add}\,n\,n\,(\mathsf{s}(\mathsf{s}\,\mathsf{z})); y_1 : \mathsf{n}\,n, y_2 : \mathsf{+}, y_3 : \mathsf{n}\,\mathsf{z} \Vdash (y_1 \cdot y_2) \cdot (y_3 \cdot y_4) :: (\mathsf{n}\,n \cdot \mathsf{+}) \cdot (\mathsf{n}\,\mathsf{z} \cdot {!\mathsf{add}}\,n\,n\,(\mathsf{s}(\mathsf{s}\,\mathsf{z})))}}{n : \mathsf{nat}, y_4 : \mathsf{add}\,n\,n\,(\mathsf{s}(\mathsf{s}\,\mathsf{z})); y_1 : \mathsf{n}\,n, y_2 : \mathsf{+}, y_3 : \mathsf{n}\,\mathsf{z} \Vdash (n.(y_1 \cdot y_2) \cdot (y_3 \cdot y_4)) :: \exists n.\,(\mathsf{n}\,n \cdot \mathsf{+}) \cdot (\mathsf{n}\,\mathsf{z} \cdot {!\mathsf{add}}\,n\,n\,(\mathsf{s}(\mathsf{s}\,\mathsf{z})))}$$

Then, we find a substitution σ such that we can derive the following:

$$\cdot; (x_1 : \mathsf{num}(\mathsf{s}\,\mathsf{z}), x_2 : \mathsf{+}, x_3 : \mathsf{num}\,\mathsf{z}) \vdash \sigma : (n : \mathsf{nat}, y_4 : \mathsf{add}\,n\,n\,(\mathsf{s}(\mathsf{s}\,\mathsf{z}))); (y_1 : \mathsf{num}\,n, y_2 : \mathsf{+}, y_3 : \mathsf{num}\,\mathsf{z})$$

The substitution $\sigma = [], (\mathsf{s}\,\mathsf{z})/n, (\mathsf{add}/\mathsf{s}\,\mathsf{add}/\mathsf{z})/y_4, x_1/y_1, x_2/y_2, x_3/y_3$ works in this case.

In future examples, we will use a derived notation of a *filled pattern*, a substitution applied to a pattern. For example, rather than first deriving a pattern and then a substitution in the preceding example, we will simply say that the filled pattern $((\mathsf{s}\,\mathsf{z}).\,(x_1 \cdot x_2) \cdot (x_3 \cdot (\mathsf{add}/\mathsf{s}\,\mathsf{add}/\mathsf{z})))$ is a proof term showing that the ephemeral context $(x_1 : \mathsf{num}(\mathsf{s}\,\mathsf{z}), x_2 : \mathsf{+}, x_3 : \mathsf{num}\,\mathsf{z})$ is described by the positive proposition $\exists n.\,(\mathsf{num}\,n \cdot \mathsf{+}) \cdot (\mathsf{num}\,\mathsf{z} \cdot {!\mathsf{add}}\,n\,n\,(\mathsf{s}(\mathsf{s}\,\mathsf{z})))$.

4.6 Negative types as transition rules

Negative types were introduced in Section 3.2 as describing ways in which contexts can change. The proposition $\mathsf{b} \cdot \mathsf{c} \twoheadrightarrow {\uparrow}\mathsf{e}$, for example, is essentially a rewriting rule that allows an ephemeral context such as $(\mathsf{a}\,\mathsf{b}\,\mathsf{c}\,\mathsf{d})$ to be rewritten as an ephemeral context $(\mathsf{a}\,\mathsf{e}\,\mathsf{d})$.

The judgment associated with negative propositions is $(\Gamma; \Delta \Vdash g :: A^- \gg \Gamma'; \Delta' \Vdash p')$. This judgment expresses that, if the premises of A^- — the context-shaped holes Γ and Δ — can be filled by consuming some number of ephemeral resources, then the consumed ephemeral resources can be replaced with the conclusion of A^- — the resources described by Δ' and the new facts described by Γ'. The *goal* g captures the premises, and the pattern p describes the conclusion; together, a goal g and a pattern p are the proof term associated with a negative proposition.

The simplest negative proposition is ${\uparrow}A^+$, which consumes no resources and produces the resources described by A^+:

$$\frac{\Gamma'; \Delta' \Vdash p' :: A^+}{\cdot; \cdot \Vdash () :: {\uparrow}A^+ \gg \Gamma'; \Delta' \Vdash p'}$$

The proposition $A^- \mathbin{\&} B^-$ can represent either the transition described by A^- or the transition described by B^-:

$$\frac{\Gamma; \Delta \Vdash p :: A^- \gg \Gamma'; \Delta' \Vdash p'}{\Gamma; \Delta \Vdash (\pi_1 p) :: (A^- \mathbin{\&} B^-) \gg \Gamma'; \Delta' \Vdash p'} \qquad \frac{\Gamma; \Delta \Vdash p :: B^- \gg \Gamma'; \Delta' \Vdash p'}{\Gamma; \Delta \Vdash (\pi_2 p) :: (A^- \mathbin{\&} B^-) \gg \Gamma'; \Delta' \Vdash p'}$$

Transitions have to be located at a particular point in the ephemeral context; ordered implication $(A^+ \twoheadrightarrow B^-)$ is a transition that consumes a part of the context described by A^+ to the right of its initial position and then proceeds with the transition described by B^-.[6]

$$\frac{\Gamma_1; \Delta_1 \Vdash p :: A^+ \quad \Gamma_2; \Delta_2 \Vdash g :: B^- \gg \Gamma'; \Delta' \Vdash p'}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 \Vdash (p; g) :: (A^+ \twoheadrightarrow B^-) \gg \Gamma'; \Delta' \Vdash p'}$$

Finally, universal quantification $\forall x{:}\tau.A^-$ can behave like $A^-[t/x]$ for any $t : \tau$; therefore, its pattern introduces a new

$$\frac{\Gamma; \Delta \Vdash g :: A^- \gg \Gamma'; \Delta' \Vdash p'}{x : \tau, \Gamma; \Delta \Vdash (x.g) :: (\forall x{:}\tau.A^-) \gg \Gamma'; \Delta' \Vdash p'}$$

When the placement of the shift operator $\uparrow$ can be inferred from the context, we will leave it implicit, writing $A^+ \twoheadrightarrow B^+$ instead of $A \twoheadrightarrow \uparrow B^+$.

4.7 Transitions and expressions

We now can actually describe parsing the concrete syntax of $\mathcal{L}_0$ using transition rules. First, we describe the abstract syntax of $\mathcal{L}_0$ as canonical terms of type exp; then we describe an ordered type family parsed E and two rewriting rules describing how the concrete syntax can be parsed.

```
exp : type.
n : nat → exp.
plus : exp → exp → exp.
parsed : exp → type_o.
parse/num : num N ↠ parsed(n N).
parse/plus : parsed E1 · + · parsed E2 ↠ parsed(plus E1 E2).
```

Intuitively, parse/num is a rewriting rule that allows us to transition from the ephemeral context $(\mathsf{num}(\mathsf{s}\,\mathsf{z})), +, (\mathsf{num}\,\mathsf{z})$ to the ephemeral context $(\mathsf{parsed}(\mathsf{n}(\mathsf{s}\,\mathsf{z}))), +, (\mathsf{num}\,\mathsf{z})$. To describe the proof term capturing this transition, we first discover the goal and pattern associated with the type of parse/num (note that the implicit quantification and the implicit shift operator $\uparrow$ are both made explicit):

$$\dfrac{\dfrac{\dfrac{}{(\cdot; y : \mathsf{num}\, n) \Vdash y :: \mathsf{num}\, n} \quad \dfrac{\dfrac{}{(\cdot; y_1 : \mathsf{parsed}(\mathsf{n}\, n)) \Vdash y_1 :: \mathsf{parsed}(\mathsf{n}\, n)}}{(\cdot; \cdot) \Vdash () :: \uparrow \mathsf{parsed}(\mathsf{n}\, n) \gg (\cdot; y_1 : \mathsf{parsed}(\mathsf{n}\, n)) \Vdash y_1}}{(\cdot; y : \mathsf{num}\, n) \Vdash (y; ()) :: (\mathsf{num}\, n \twoheadrightarrow \uparrow \mathsf{parsed}(\mathsf{n}\, n)) \gg (\cdot; y_1 : \mathsf{parsed}(\mathsf{n}\, n)) \Vdash y_1}}{(n : \mathsf{nat}; y : \mathsf{num}\, n) \Vdash (n.y; ()) :: (\forall n.\, \mathsf{num}\, n \twoheadrightarrow \uparrow \mathsf{parsed}(\mathsf{n}\, n)) \gg (\cdot; y_1 : \mathsf{parsed}(\mathsf{n}\, n)) \Vdash y_1}$$

If x_1 is the variable marking the ordered resource $(\mathsf{parsed}(\mathsf{n}(\mathsf{s}\,\mathsf{z})))$, we can fill this goal with the substitution $([], \mathsf{s}\,\mathsf{z}/n, x_1/y)$; the resulting filled pattern is $(\mathsf{s}\,\mathsf{z}.\, x_1; ())$. The proof term corresponding to the transition can be written in a CLF-like notation as the let-expression $(\mathsf{let}\ (y_1) = \mathsf{parse/num}(\mathsf{s}\,\mathsf{z}.\, x_1; ())\ \mathsf{in})$. However, I prefer a more "directional" notation for transitions that looks like this: $(\mathsf{parse/num}(\mathsf{s}\,\mathsf{z}.\, x_1; ()) \rhd (y_1))$. In addition, we will leave universal quantification implicit and omit the trailing $()$ in a goal; the final result is a proof term for the aforementioned transition that looks like this:

[6]In ordered logic there are actually two distinct implications, right ordered implication $A^+ \twoheadrightarrow B^-$ and left ordered implication $A^+ \rightarrowtail B^-$. Either one or the other is usually sufficient for the logical fragment we are considering, and I conjecture that as long as the ephemeral context only contains atomic propositions, the addition of left ordered implication does not actually add any expressiveness.

$\textsf{parse/num}(x_1) \rhd (y_1)$
$\quad : (x_1 : \textsf{num}(\textsf{s}\,\textsf{z}), x_2 : \textsf{+}, x_3 : \textsf{num}\,\textsf{z}) \leadsto_{\Sigma\textsf{parse}} (y_1 : \textsf{parsed}(\textsf{n}(\textsf{s}\,\textsf{z})), x_2 : \textsf{+}, x_3 : \textsf{num}\,\textsf{z})$

We'll return to the Σparse annotation in the next section; in the meantime, the typing rule for a transition $\mathcal{T} : (\Gamma; \Delta \leadsto_\Sigma \Gamma'; \Delta')$ is as follows:

$$\frac{c : A^- \in \Sigma \qquad \Gamma_{in}; \Delta_{in} \Vdash g :: A^- \gg \Gamma_{out}; \Delta_{out} \Vdash p \qquad \Delta \approx \Delta_L, \Delta', \Delta_R \qquad \Gamma; \Delta' \vdash_\Sigma \sigma : \Gamma_{in}; \Delta_{in}}{c(g[\sigma]) \rhd p : \Gamma; \Delta \leadsto_\Sigma \Gamma, \Gamma_{out}; \Delta_L, \Delta_{out}, \Delta_R}$$

This transition can be read like this:

- Starting from the state $\Gamma; \Delta$,
- Pick a rule A^- from the signature Σ (this is the constant c),
- Determine the input context $\Gamma_{in}; \Delta_{in}$ (this is the goal g) and the output context $\Gamma_{out}; \Delta_{out}$ (this is the pattern p) associated with A^-,
- Split the ephemeral context Δ into three parts, Δ_L, Δ', and Δ_R,
- Show that, using the persistent facts Γ and the ephemeral resources Δ' you can fulfill the demands represented by the input pattern (this is the substitution σ), and
- Extend the persistent context with Γ_{out} and replace Δ' in the context with $\Delta_{\textsf{out}}$ to get the result of the transition.

4.8 Expressions

A sequence of one or more transitions $\mathcal{T}$ is an expression $\mathcal{E} : (\Gamma; \Delta \leadsto_\Sigma^* \Gamma'; \Delta')$. An expression is either a single transition $\mathcal{T}$, the sequential composition of two expressions $\mathcal{E}_1; \mathcal{E}_2$, or the empty expression $\diamond$. We treat sequential composition as an associative operation with unit $\diamond$.

The following expression represents a complete parse of an $\mathcal{L}_0$ program:

$\textsf{parse/num}(x_1) \rhd (y_1);\ \textsf{parse/num}(x_3) \rhd (y_3);\ \textsf{parse/plus}(y_1 \cdot x_2 \cdot y_3) \rhd (y)$
$\quad : (x_1 : \textsf{num}(\textsf{s}\,\textsf{z}), x_2 : \textsf{+}, x_3 : \textsf{num}\,\textsf{z}) \leadsto^*_{\Sigma\textsf{parse}} (y : \textsf{parsed}(\textsf{plus}\,(\textsf{n}(\textsf{s}\,\textsf{z}))\,(\textsf{n}\,\textsf{z})))$

The annotation of a signature Σparse on the transition $\mathcal{T} : (\Gamma; \Delta \leadsto_\Sigma \Gamma'; \Delta')$ will be critical in practice. We frequently want to define multiple families of transitions that can act on a single proposition. For instance, consider the following rules which describe the direct calculation of a number from the concrete syntax of $\mathcal{L}_0$ (note, in particular, the use of !add $\textsf{N}_1\,\textsf{N}_2\,\textsf{N}_3$ to capture the "side condition" in calc/plus):

$\textsf{calc} : \textsf{nat} \to \textsf{type}_o.$
$\textsf{calc/num} : \textsf{num}\,\textsf{N} \twoheadrightarrow \textsf{calc}\,\textsf{N}.$
$\textsf{calc/plus} : \textsf{calc}\,\textsf{N}_1 \cdot \textsf{+} \cdot \textsf{calc}\,\textsf{N}_2 \cdot \textsf{!add}\,\textsf{N}_1\,\textsf{N}_2\,\textsf{N}_3 \twoheadrightarrow \textsf{calc}\,\textsf{N}_3.$

Then we can talk about the transitions and expressions where we refer only to the transition rules beginning with calc.

$\textsf{calc/num}(x_1) \rhd (y_1) : (x_1 : \textsf{num}(\textsf{s}\,\textsf{z}), x_2 : \textsf{+}, x_3 : \textsf{num}\,\textsf{z}) \leadsto_{\Sigma\textsf{calc}} (y_1 : \textsf{calc}(\textsf{s}\,\textsf{z}), x_2 : \textsf{+}, x_3 : \textsf{num}\,\textsf{z})$

$$\begin{array}{l}\mathsf{calc/num}(x_1) \rhd (y_1);\ \ \mathsf{calc/num}(x_3) \rhd (y_3);\ \ \mathsf{calc/plus}(y_1 \cdot x_2 \cdot y_3 \cdot (\mathsf{add/s\,add/z})) \rhd (y) \\ \qquad\qquad\qquad\qquad\qquad\qquad : \ (x_1 : \mathsf{num(s\,z)}, x_2 : \mathsf{+}, x_3 : \mathsf{num\,z}) \ \leadsto^*_{\Sigma\mathsf{calc}} \ (y : \mathsf{calc(s\,z)}))\end{array}$$

We use these annotations to specify theorems about the relationship between two different transitions or expressions. For instance, if $\Sigma\mathsf{e}$ describes the (not-yet-specified) SSOS specification of $\mathcal{L}_0$, then we can write the following conjecture about the relationship between the rules starting with calc/... and the rules starting with parse/...

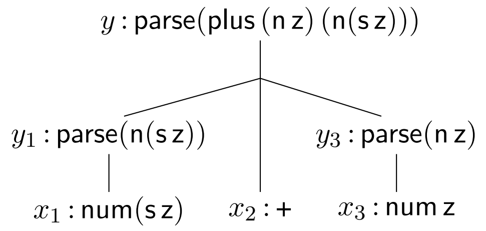
Conjecture 1. *If* $(\Delta \leadsto^*_{\Sigma\mathsf{calc}} y : \mathsf{calc\,N})$ *and such that* $(\Delta \leadsto^*_{\Sigma\mathsf{parse}} y : \mathsf{parsed\,E})$*, then* $(x : \mathsf{eval\,E} \leadsto^*_{\Sigma\mathsf{e}} y : \mathsf{retn(n\,N)})$.

4.9 Concurrent equivalence

The two expressions below both represent a complete parse of the $\mathcal{L}_0$ program that we have been using in our example:

$$\mathsf{parse/num}(x_3) \rhd (y_3);\ \ \mathsf{parse/num}(x_1) \rhd (y_1);\ \ \mathsf{parse/plus}(y_1 \cdot x_2 \cdot y_3) \rhd (y)$$
$$\mathsf{parse/num}(x_1) \rhd (y_1);\ \ \mathsf{parse/num}(x_3) \rhd (y_3);\ \ \mathsf{parse/plus}(y_1 \cdot x_2 \cdot y_3) \rhd (y)$$

In fact, these two expressions represent the *same* parse if we look at the parse not as a list of transitions but as a tree of string rewrites:



The notion of concurrent (or permutative) equivalence $\mathcal{E}_1 \approx \mathcal{E}_2$ captures the notion that, while we represent expressions as sequences, they are perhaps more appropriately thought of as directed acyclic graphs. Concurrent equivalence is the least equivalence relation such that

$$(\mathcal{E}_1;\ c(g[\sigma]) \rhd p;\ c'(g'[\sigma']) \rhd p';\ \mathcal{E}_2) \ \approx \ (\mathcal{E}_1;\ c'(g'[\sigma']) \rhd p';\ c(g[\sigma]) \rhd p;\ \mathcal{E}_2)$$

whenever σ does not include any variables bound by p' and σ' likewise does not mention any variables bound by p.

4.10 Notational conventions

We have discussed a number of notational conventions, such as omitting the shift operator $\uparrow$ and simplifying the way that goals are written down. In discussions of SSOS in Section 5, we will use a more concise shorthand notation for describing transitions and expressions; this section will introduce this shorthand notation.

Consider the last transition in the parse of our example $\mathcal{L}_0$ program:

$$\begin{array}{l}\mathsf{parse/plus}(y_1 \cdot x_2 \cdot y_3) \rhd (y) \\ \qquad\qquad : \ (y_1 : \mathsf{parsed(n(s\,z))}, x_2 : \mathsf{+}, y_3 : \mathsf{parsed(n\,z)}) \ \leadsto_{\Sigma\mathsf{parse}} \ (y : \mathsf{parsed(plus\,(n(s\,z))\,(n\,z))})\end{array}$$

The variables that describe what parts of the context changed are not strictly necessary here; we can convey the exact same information in a more concise form by omitting the mention of any variables:

$$\mathsf{parse/plus} \;:\; \mathsf{parsed}(\mathsf{n}(\mathsf{s}\,\mathsf{z})), \mathsf{+}, \mathsf{parsed}(\mathsf{n}\,\mathsf{z}) \;\leadsto_{\Sigma\mathsf{parse}}\; \mathsf{parsed}(\mathsf{plus}\,(\mathsf{n}(\mathsf{s}\,\mathsf{z}))\,(\mathsf{n}\,\mathsf{z}))$$

This notation must be used carefully: for instance, if we say that the complete parse is represented by the expression $(\mathsf{parse/num}; \mathsf{parse/num}; \mathsf{parse/plus})$, it is unclear whether the first token or the last token was rewritten first (although in this case, both of the traces are concurrently equivalent!)

It is not always the case that the name of a rule alone is sufficient. The exception is when one of the premises to a rule is a canonical form, such as the last step in the calculation of our example $\mathcal{L}_0$ program:

$$\begin{array}{l}\mathsf{calc/plus}(y_1 \cdot x_2 \cdot y_3 \cdot (\mathsf{add/s}\,\mathsf{add/z})) \rhd (y) \\ \qquad\qquad\qquad\qquad : \;\; (y_1 : \mathsf{calc}(\mathsf{s}\,\mathsf{z}), x_2 : \mathsf{+}, y_3 : \mathsf{calc}\,\mathsf{z}) \;\leadsto_{\Sigma\mathsf{calc}}\; (y : \mathsf{calc}(\mathsf{s}\,\mathsf{z}))\end{array}$$

The shorthand version of this rule only mentions the canonical form $(\mathsf{add/s}\,\mathsf{add/z} : \mathsf{add}\,(\mathsf{s}\,\mathsf{z})\,\mathsf{z}\,(\mathsf{s}\,\mathsf{z}))$ — unlike variable names, that part of the original transition conveys more information than just a position.

$$\mathsf{calc/plus}(\mathsf{add/s}\,\mathsf{add/z}) \;:\; \mathsf{calc}(\mathsf{s}\,\mathsf{z}), \mathsf{+}, \mathsf{calc}\,\mathsf{z} \;\leadsto_{\Sigma\mathsf{calc}}\; \mathsf{calc}(\mathsf{s}\,\mathsf{z})$$

Finally, in the following section I will refer to states $\Gamma; \Delta$ generally as $\mathcal{S}$, and will use the notation $\mathcal{S}[\Delta]$ to describe a state that includes somewhere inside of it the ephemeral context Δ — a "one-hole context" over contexts, in other words. For instance, we can write the following:

$$\begin{array}{rcl}\mathsf{parse/num} & : & \mathcal{S}[\mathsf{num}(\mathsf{s}(\mathsf{s}\,\mathsf{z}))] \;\leadsto_{\Sigma\mathsf{parse}}\; \mathcal{S}[\mathsf{parsed}(\mathsf{s}(\mathsf{s}\,\mathsf{z}))] \\ \mathsf{calc/num} & : & \mathcal{S}[\mathsf{num}(\mathsf{s}(\mathsf{s}\,\mathsf{z}))] \;\leadsto_{\Sigma\mathsf{calc}}\; \mathcal{S}[\mathsf{calc}(\mathsf{s}(\mathsf{s}\,\mathsf{z}))] \\ \mathsf{calc/plus}(\mathsf{plus/z}) & : & \mathcal{S}[\mathsf{calc}\,\mathsf{z}, \mathsf{+}, \mathsf{calc}(\mathsf{s}(\mathsf{s}\,\mathsf{z}))] \;\leadsto_{\Sigma\mathsf{calc}}\; \mathcal{S}[\mathsf{calc}(\mathsf{s}(\mathsf{s}\,\mathsf{z}))]\end{array}$$

5 Substructural operational semantics

Now that we have discussed the outlines of our logical framework, we can finally return to the discussion of substructural operational semantics specifications that we motivated in the introduction and in Section 2. In this section, we consider a series of relatively simple substructural operational semantics specifications, mostly taken from [28]. Substructural operational semantics (or SSOS) is a style of specifying the operational semantics of programming languages as transitions in substructural logics. Its origin lies in a set of examples illustrating the expressiveness of the logical framework CLF [2], though elements were anticipated by Chirimar's Ph.D. thesis [4] and by the continuation-based specification of ML with references in LLF [1]. A methodology of SSOS specifications was refined and generalized to include other substructural logics in subsequent work [24, 28].

In this section, we will develop a base language $\mathcal{L}_1$ that extends $\mathcal{L}_0$ with functions and function application (Section 5.1) and then extend that language with parallel evaluation of pairs (Section 5.3) and with exceptions and exception handling (Section 5.4). The novel content of this section is a presentation of the *static semantics* of this language and proof of language safety via progress and preservation lemmas that it enables. We give a proof of safety for $\mathcal{L}_1$ in Section 5.2, and then briefly discuss how that safety proof can be extended when we consider the static semantics of parallel evaluation and exceptions.

Fundamental components of SSOS specifications Many of the elements of substructural operational semantics specifications are immediately familiar to anyone who has written or read traditional SOS-style operational semantics formalized in Twelf. The syntax, for instance, consists of expressions E, types T, stack frames F, and a judgment (value V) capturing the subsort of expressions that are also values (we continue to write V for those expressions which are known to be values).

```
exp : type.
tp : type.
frame : type.
value : exp → type.
```

Particular to SSOS specifications are three groups of propositions. The *active* propositions are those that can always eagerly participate in a transition; the prototypical active proposition is (eval E), an ordered atomic proposition containing an expression E that we are trying to evaluate to a value.

eval : exp → type$_o$.

The *latent* propositions represent suspended computations. The primary latent proposition we will consider is (cont F), which stores a part of a continuation (in the form of a stack frame) waiting for a value to be returned.[7]

cont : frame → type$_o$.

Finally, *passive* propositions are those that do not drive transitions on their own, but which may take part in transitions when combined with latent propositions. The only passive proposition in the SSOS specification of an effect-free languages is (retn V), which holds a value being returned from the evaluation of an expression. On its own, it is passive, representing a completed computation. If there is a latent stack frame to its left, a transition should occur by returning the value to the stack frame.

retn : exp → type$_o$.

5.1 Specifying $\mathcal{L}_1$

We will introduce SSOS specifications by encoding the operational semantics of the very simple language $\mathcal{L}_0$ that has been used as running example; we extend $\mathcal{L}_0$ with call-by-value functions to make the language less boring, and call the result $\mathcal{L}_1$.

5.1.1 Syntax

The syntax of expressions can be given by the following BNF specification:

$$e ::= x \mid \lambda x.e \mid e_1(e_2) \mid n \mid e_1 + e_2$$

The encoding of that BNF is standard, including the use of higher-order abstract syntax to represent binding. The judgments v/n and v/lam indicate that functions and numbers are values.

[7]Latent propositions can also be seen as artifacts of kind of defunctionalization that is performed on "higher-order" SSOS specifications to make them more amenable to extension and analysis. (See Section **??** for a further discussion.)

$\mathsf{lam} : (\mathsf{exp} \to \mathsf{exp}) \to \mathsf{exp}.$
$\mathsf{app} : \mathsf{exp} \to \mathsf{exp} \to \mathsf{exp}.$
$\mathsf{n} : \mathsf{nat} \to \mathsf{exp}.$
$\mathsf{plus} : \mathsf{exp} \to \mathsf{exp} \to \mathsf{exp}.$
$\mathsf{v/lam} : \mathsf{value}(\mathsf{lam}\,\lambda \mathsf{x}.\,\mathsf{E}\,\mathsf{x}).$
$\mathsf{v/n} : \mathsf{value}(\mathsf{n}\,\mathsf{N}).$

We also need to define the syntax of frames.

$$f ::= \Box(e_2) \mid v_1(\Box) \mid \Box + e \mid v + \Box$$

app_1	$: \mathsf{exp} \to \mathsf{frame}.$	— *this is* $\Box(e_2)$
app_2	$: \mathsf{exp} \to \mathsf{frame}.$	— *this is* $v_1(\Box)$
plus_1	$: \mathsf{exp} \to \mathsf{frame}.$	— *this is* $\Box + e_2$
plus_2	$: \mathsf{exp} \to \mathsf{frame}.$	— *this is* $v_1 + \Box$

5.1.2 Dynamic semantics

The dynamic semantics for $\mathcal{L}_1$ is straightforward by analogy with an abstract machine for control. A function is a value, and therefore it is always returned immediately, and a function application first evaluates the function part to a value, then evaluates the argument part to a value, and then finally substitutes the argument into the function. In the usual style of higher-order abstract syntax, substitution is performed by application in the rule $\mathsf{e/app}_2$.

$\mathsf{e/lam} : \ \mathsf{eval}(\mathsf{lam}\,\lambda \mathsf{x}.\,\mathsf{E}\,\mathsf{x}) \twoheadrightarrow \mathsf{retn}(\mathsf{lam}\,\lambda \mathsf{x}.\,\mathsf{E}\,\mathsf{x}).$
$\mathsf{e/app} : \ \mathsf{eval}(\mathsf{app}\,\mathsf{E}_1\,\mathsf{E}_2) \twoheadrightarrow \mathsf{cont}(\mathsf{app}_1\,\mathsf{E}_2) \cdot \mathsf{eval}\,\mathsf{E}_1.$
$\mathsf{e/app}_1 : \ \mathsf{cont}(\mathsf{app}_1\,\mathsf{E}_2) \cdot \mathsf{retn}\,\mathsf{V}_1 \twoheadrightarrow \mathsf{cont}(\mathsf{app}_2\,\mathsf{V}_1) \cdot \mathsf{eval}\,\mathsf{E}_2.$
$\mathsf{e/app}_2 : \ \mathsf{cont}(\mathsf{app}_2(\mathsf{lam}\,\lambda \mathsf{x}.\,\mathsf{E}_0\,\mathsf{x})) \cdot \mathsf{retn}\,\mathsf{V}_2 \twoheadrightarrow \mathsf{eval}(\mathsf{E}_0\,\mathsf{V}_2).$

With one exception, the dynamic semantics of numbers and addition are just as straightforward. As mentioned before, when both subterms have been evaluated to numerical values, there must be a primitive operation that actually adds the two numbers together. As discussed in the context of the calc/... rules in the previous section, the solution is to rely on the persistent type family (add N M P) that adequately encodes the inductive definition of addition for natural numbers. Given this definition, the dynamic semantics of addition are also straightforward: a number is a value, and to evaluate $\mathsf{plus}\,\mathsf{E}_1\,\mathsf{E}_2$ we evaluate E_1, then evaluate E_2, and then add the resulting numbers.

$\mathsf{e/n} : \ \mathsf{eval}(\mathsf{nat}\,\mathsf{N}) \twoheadrightarrow \mathsf{retn}(\mathsf{nat}\,\mathsf{N}).$
$\mathsf{e/plus} : \ \mathsf{eval}(\mathsf{plus}\,\mathsf{E}_1\,\mathsf{E}_2) \twoheadrightarrow \mathsf{cont}(\mathsf{plus}_1\,\mathsf{E}_2) \cdot \mathsf{eval}\,\mathsf{E}_1.$
$\mathsf{e/plus}_1 : \ \mathsf{cont}(\mathsf{plus}_1\,\mathsf{E}_2) \cdot \mathsf{retn}\,\mathsf{V}_1 \twoheadrightarrow \mathsf{cont}(\mathsf{plus}_2\,\mathsf{V}_1) \cdot \mathsf{eval}\,\mathsf{E}_2.$
$\mathsf{e/plus}_2 : \ \mathsf{cont}(\mathsf{plus}_2(\mathsf{n}\,\mathsf{N}_1)) \cdot \mathsf{retn}(\mathsf{n}\,\mathsf{N}_2) \cdot !\mathsf{add}\,\mathsf{N}_1\,\mathsf{N}_2\,\mathsf{N}_3 \twoheadrightarrow \mathsf{retn}(\mathsf{n}\,\mathsf{N}_3).$

5.1.3 Example trace

As an example, Figure 1 shows a complete evaluation of an $\mathcal{L}_1$ expression that we would write on paper as $((\lambda x.\, 2 + x)(1 + 5))$. The expression represented by the series of transitions on the right has the following type:

$$\begin{array}{l}\mathsf{eval}(\mathsf{app}\,(\mathsf{lam}(\lambda x.\mathsf{plus}\,(\mathsf{n}(\mathsf{s}(\mathsf{s}\,\mathsf{z})))\,x))\,(\mathsf{plus}\,(\mathsf{n}(\mathsf{s}\,\mathsf{z}))\,(\mathsf{n}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}\,\mathsf{z}))))))))\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \leadsto^{*}_{\Sigma\mathsf{e}} \;\; \mathsf{retn}(\mathsf{n}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}\,\mathsf{z})))))))))\end{array}$$

For brevity and clarity, the on-paper notation is used in the figure.

$$\begin{array}{lll}
 & \mathsf{eval}((\lambda x.\,2+x)(1+5)) & \\
\leadsto_{\Sigma\mathsf{e}} & \mathsf{cont}(\Box(1+5)),\;\; \mathsf{eval}(\lambda x.\,2+x) & \text{by } \mathsf{e/app} \\
\leadsto_{\Sigma\mathsf{e}} & \mathsf{cont}(\Box(1+5)),\;\; \mathsf{retn}(\lambda x.\,2+x) & \text{by } \mathsf{e/lam} \\
\leadsto_{\Sigma\mathsf{e}} & \mathsf{cont}((\lambda x.\,2+x)(\Box)),\;\; \mathsf{eval}(1+5) & \text{by } \mathsf{e/app_1} \\
\leadsto_{\Sigma\mathsf{e}} & \mathsf{cont}((\lambda x.\,2+x)(\Box)),\;\; \mathsf{cont}(\Box+5),\;\; \mathsf{eval}(1) & \text{by } \mathsf{e/plus} \\
\leadsto_{\Sigma\mathsf{e}} & \mathsf{cont}((\lambda x.\,2+x)(\Box)),\;\; \mathsf{cont}(\Box+5),\;\; \mathsf{retn}(1) & \text{by } \mathsf{e/n} \\
\leadsto_{\Sigma\mathsf{e}} & \mathsf{cont}((\lambda x.\,2+x)(\Box)),\;\; \mathsf{cont}(1+\Box),\;\; \mathsf{eval}(5) & \text{by } \mathsf{e/plus_1} \\
\leadsto_{\Sigma\mathsf{e}} & \mathsf{cont}((\lambda x.\,2+x)(\Box)),\;\; \mathsf{cont}(1+\Box),\;\; \mathsf{retn}(5) & \text{by } \mathsf{e/n} \\
\leadsto_{\Sigma\mathsf{e}} & \mathsf{cont}((\lambda x.\,2+x)(\Box)),\;\; \mathsf{retn}(6) & \text{by } \mathsf{e/plus_2(add/s\,add/z)} \\
\leadsto_{\Sigma\mathsf{e}} & \mathsf{eval}(2+6) & \text{by } \mathsf{e/app_2} \\
\leadsto_{\Sigma\mathsf{e}} & \mathsf{cont}(\Box+6),\;\; \mathsf{eval}(2) & \text{by } \mathsf{e/plus} \\
\leadsto_{\Sigma\mathsf{e}} & \mathsf{cont}(\Box+6),\;\; \mathsf{retn}(2) & \text{by } \mathsf{e/n} \\
\leadsto_{\Sigma\mathsf{e}} & \mathsf{cont}(2+\Box),\;\; \mathsf{eval}(6) & \text{by } \mathsf{e/plus_1} \\
\leadsto_{\Sigma\mathsf{e}} & \mathsf{cont}(2+\Box),\;\; \mathsf{retn}(6) & \text{by } \mathsf{e/n} \\
\leadsto_{\Sigma\mathsf{e}} & \mathsf{retn}(8) & \text{by } \mathsf{e/plus_2(add/s(add/s\,add/z))}
\end{array}$$

Figure 1: Example trace in $\mathcal{L}_1$.

5.2 Static semantics, progress, and preservation

The description of the dynamic semantics of $\mathcal{L}_1$ is not enough on its own; there's nothing we have said so far that prevents us from attempting to evaluate $5+\lambda x.x$ or $3(\lambda x.x+2)$, both of which would get *stuck*. A state is stuck if it has not evaluated all the way to the single atomic proposition ($\mathsf{retn}\,\mathsf{E}$) but which nevertheless cannot take a step. A safe expression is one that cannot lead to a stuck state:

Definition 1 (Safety). *The expression* E *is* **safe** *if* ($\mathsf{eval}\,\mathsf{E} \leadsto^{*}_{\Sigma\mathsf{e}} \mathcal{S}$) *implies that either* $\mathcal{S} \leadsto_{\Sigma\mathsf{e}} \mathcal{S}'$ *for some* $\mathcal{S}'$ *or else* $\mathcal{S} = \mathsf{retn}\,\mathsf{V}$ *for some* V.

While this definition of safety is appropriate for our current purposes, it is not necessarily the only definition of safety we might want to consider. For instance, a system that might deadlock is usually considered to be safe, but would not be safe according to this definition.

Integral to establishing safety will be the static semantics, a new set of rewriting rules annotated with t/... (so we use $\Sigma\mathsf{t}$ to refer to them collectively). The goal of these rewriting rules is to rewrite the entire state $\mathcal{S}$ to a single atomic proposition $\mathsf{abs}\,\mathsf{T}$. The atomic proposition $\mathsf{abs}\,\mathsf{T}$ is intended to represent an <u>abs</u>traction of states that will produce a value of type T if they produce any value at all.[8] These rewriting rules, which

[8]This characterization of $\mathsf{abs}\,\mathsf{T}$ is actually a corollary of the preservation lemma.

we will refer to as the static semantics, rely on the not-yet-defined judgments of $\mathsf{E}\,\mathsf{T}$ (the expression E has type T) and off $\mathsf{F}\,\mathsf{T}\,\mathsf{T}'$ (the frame F takes values of type T to states of type T'). The rules themselves are straightforward:

$$\begin{aligned}
&\mathsf{abs} : \mathsf{tp} \to \mathsf{type}_o.\\
&\mathsf{t/eval} : \quad \mathsf{eval}\,\mathsf{E} \cdot \mathsf{!of}\,\mathsf{E}\,\mathsf{T} \twoheadrightarrow \mathsf{abs}\,\mathsf{T}.\\
&\mathsf{t/retn} : \quad \mathsf{retn}\,\mathsf{E} \cdot \mathsf{!value}\,\mathsf{E} \cdot \mathsf{!of}\,\mathsf{E}\,\mathsf{T} \twoheadrightarrow \mathsf{abs}\,\mathsf{T}.\\
&\mathsf{t/cont} : \quad \mathsf{cont}\,\mathsf{F} \cdot \mathsf{abs}\,\mathsf{T} \cdot \mathsf{!off}\,\mathsf{F}\,\mathsf{T}\,\mathsf{T}' \twoheadrightarrow \mathsf{abs}\,\mathsf{T}'.
\end{aligned}$$

These static semantics (together with the not-yet-specified typing rules) allow us to prove progress and preservation theorems, which in turn allow us to establish type safety for the language.

Progress If $\mathcal{S} \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{abs}\,\mathsf{T}$ and $\mathcal{S}$ contains no resources $\mathsf{abs}\,\mathsf{T}$, then either $\mathcal{S} \rightsquigarrow_{\Sigma\mathsf{e}} \mathcal{S}'$ for some $\mathcal{S}'$ or else $\mathcal{S} = \mathsf{retn}\,\mathsf{V}$.

$$\begin{tikzcd}
\mathcal{S} \arrow[r, rightsquigarrow, dotted, "\Sigma\mathsf{e}"] \arrow[dr, equal] \arrow[d, rightsquigarrow, "\Sigma\mathsf{t}", "*"'] & \mathcal{S}' \arrow[d, phantom, "\text{or}"] \\
\mathsf{abs}\,\mathsf{T} & \mathsf{retn}\,\mathsf{V}
\end{tikzcd}$$

Preservation If $\mathcal{S} \rightsquigarrow_{\Sigma\mathsf{e}} \mathcal{S}'$ and $\mathcal{S} \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{abs}\,\mathsf{T}$, then $\mathcal{S}' \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{abs}\,\mathsf{T}$.

$$\begin{tikzcd}
\mathcal{S} \arrow[r, rightsquigarrow, "\Sigma\mathsf{e}"] \arrow[d, rightsquigarrow, "\Sigma\mathsf{t}", "*"'] & \mathcal{S}' \arrow[d, rightsquigarrow, dotted, "\Sigma\mathsf{t}", "*"'] \\
\mathsf{abs}\,\mathsf{T} & \mathsf{abs}\,\mathsf{T}
\end{tikzcd}$$

Given progress and preservation as stated above, we can prove the desired type safety theorem:

Theorem 1. *If* $\vdash \mathsf{of}\,\mathsf{E}\,\mathsf{T}$, *then* E *is safe.*

Proof. The overall picture for safety is this one:

$$\begin{tikzcd}
\mathsf{eval}\,\mathsf{E} \arrow[r, rightsquigarrow, "\Sigma\mathsf{e}"] \arrow[d, rightsquigarrow, "\Sigma\mathsf{t}", "*"'] & \mathcal{S}_1 \arrow[d, rightsquigarrow, dotted, "\Sigma\mathsf{t}", "*"'] \arrow[r, phantom, "\cdots"] & \mathcal{S}_n \arrow[r, rightsquigarrow, "\Sigma\mathsf{e}"] \arrow[d, rightsquigarrow, dotted, "\Sigma\mathsf{t}", "*"'] & \mathcal{S} \arrow[r, rightsquigarrow, dotted, "\Sigma\mathsf{e}"] \arrow[dr, equal] \arrow[d, rightsquigarrow, dotted, "\Sigma\mathsf{t}", "*"'] & \mathcal{S}' \arrow[d, phantom, "\text{or}"] \\
\mathsf{abs}\,\mathsf{T} & \mathsf{abs}\,\mathsf{T} & \mathsf{abs}\,\mathsf{T} & \mathsf{abs}\,\mathsf{T} & \mathsf{retn}\,\mathsf{V}
\end{tikzcd}$$

We are given an expression ($\mathsf{eval}\,\mathsf{E} \rightsquigarrow^*_{\Sigma\mathsf{e}} \mathcal{S}$), and we must show that either $\mathcal{S} \rightsquigarrow_{\Sigma\mathsf{e}} \mathcal{S}'$ for some $\mathcal{S}'$ or else $\mathcal{S} = \mathsf{retn}\,\mathsf{V}$ for some V. By rule $\mathsf{t/eval}$ and the premise, ($\mathsf{eval}\,\mathsf{E} \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{abs}\,\mathsf{T}$), and by induction on the length of the trace and progress lemma, ($\mathcal{S} \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{abs}\,\mathsf{T}$). By a similar induction on the trace and the observation that $\mathsf{abs}\,\mathsf{T}$ appears nowhere in any of the rules of the form $\mathsf{e}/\ldots$, $\mathcal{S}$ contains no resources $\mathsf{abs}\,\mathsf{T}$. The result then follows by the progress lemma. □

Note that both the proof of type safety and the statements of the progress and preservation theorems talk about individual transition steps and about taking a series of transitions and extending them with an additional transition. Both of these notions are naturally represented by the transitions and expressions of our logical framework, but neither of these are easy to represent in the CLF framework. The CLF framework has proof terms that express a completed derivations that has ended by proving something, but a series of transitions in our framework translates to a *partial* derivation in CLF, not a complete derivation. It is for this reason that I previously described the focus on a state-transition-based presentation of logic as the most critical difference between CLF and the framework presented here.

In the remainder of this section, we will present the typing rules and static semantics for $\mathcal{L}_1$, and then give proofs of the progress and preservation lemmas above.

5.2.1 Static semantics

Typing rules are completely conventional. The syntax of types can be written as a BNF specification as $\tau ::= \mathsf{nat} \mid \tau \rightarrow \tau$, and the signature for types is the following:

$\mathsf{tnat} : \mathsf{tp}.$
$\mathsf{arrow} : \mathsf{tp} \rightarrow \mathsf{tp} \rightarrow \mathsf{tp}.$

The typing rules are the familiar ones:

$\mathsf{of} : \mathsf{exp} \rightarrow \mathsf{tp} \rightarrow \mathsf{type}.$
$\mathsf{of/lam} : \; (\Pi x.\, \mathsf{of}\, x\, \mathsf{T} \rightarrow \mathsf{of}\, (\mathsf{E}\, x)\, \mathsf{T}') \rightarrow \mathsf{of}\, (\mathsf{lam}\, \lambda x.\, \mathsf{E}\, x)(\mathsf{arrow}\, \mathsf{T}\, \mathsf{T}').$
$\mathsf{of/app} : \; \mathsf{of}\, \mathsf{E}_1\, (\mathsf{arrow}\, \mathsf{T}'\, \mathsf{T}) \rightarrow \mathsf{of}\, \mathsf{E}_2\, \mathsf{T}' \rightarrow \mathsf{of}\, (\mathsf{app}\, \mathsf{E}_1\, \mathsf{E}_2)\, \mathsf{T}.$
$\mathsf{of/n} : \; \mathsf{of}\, (\mathsf{nat}\, \mathsf{N})\, \mathsf{tnat}.$
$\mathsf{of/plus} : \; \mathsf{of}\, \mathsf{E}_1\, \mathsf{tnat} \rightarrow \mathsf{of}\, \mathsf{E}_2\, \mathsf{tnat} \rightarrow \mathsf{of}\, (\mathsf{plus}\, \mathsf{E}_1\, \mathsf{E}_2)\, \mathsf{tnat}.$

Frame typing rules are also straightforward:

$\mathsf{off/app}_1 : \; \mathsf{of}\, \mathsf{E}_2\, \mathsf{T}' \rightarrow \mathsf{off}\, (\mathsf{app}_1\, \mathsf{E}_2)\, (\mathsf{arrow}\, \mathsf{T}'\, \mathsf{T})\, \mathsf{T}.$
$\mathsf{off/app}_2 : \; \mathsf{value}\, \mathsf{E}_1 \rightarrow \mathsf{of}\, \mathsf{E}_1\, (\mathsf{arrow}\, \mathsf{T}'\, \mathsf{T}) \rightarrow \mathsf{off}\, (\mathsf{app}_2\, \mathsf{E}_2)\, \mathsf{T}'\, \mathsf{T}.$
$\mathsf{off/plus}_1 : \; \mathsf{of}\, \mathsf{E}_2\, \mathsf{tnat} \rightarrow \mathsf{off}\, (\mathsf{plus}_1\, \mathsf{E}_2)\, \mathsf{tnat}\, \mathsf{tnat}.$
$\mathsf{off/plus}_2 : \; \mathsf{value}\, \mathsf{E}_1 \rightarrow \mathsf{of}\, \mathsf{E}_1\, \mathsf{tnat} \rightarrow \mathsf{off}\, (\mathsf{plus}_2\, \mathsf{E}_1)\, \mathsf{tnat}\, \mathsf{tnat}.$

5.2.2 Preservation

The preservation lemma establishes that our purported invariant is actually an invariant. The form of the preservation lemma should be very familiar: if the invariant described by the static semantics holds of a state, the invariant still holds of the state after any transition made under the dynamic semantics.

Lemma 1 (Preservation). *If* $\mathcal{S} \leadsto_{\Sigma_e} \mathcal{S}'$ *and* $\mathcal{S} \leadsto^*_{\Sigma_t} \mathsf{abs}\, \mathsf{T_f}$*, then* $\mathcal{S}' \leadsto^*_{\Sigma_t} \mathsf{abs}\, \mathsf{T_f}$.

Preservation proofs tend to rely on typing inversion lemmas, and we use traditional typing inversion lemmas as needed without proof. However, there is another lemma that is critical for preservation proofs for SSOS specifications. It captures the intuition that if $\mathcal{S} \leadsto^*_{\Sigma_t} \mathsf{abs}\, \mathsf{T}$, either $\mathcal{S}$ is already a substructural context containing a single resource ($\mathsf{abs}\, \mathsf{T}$) or else everything in $\mathcal{S}$ must eventually be consumed in the service of producing a single resource ($\mathsf{abs}\, \mathsf{T}$). We call this the *consumption lemma* because it represents the obligation that the static semantics consume everything in the context. Recall that $\mathcal{S}[\Delta]$ is notation representing a state $\mathcal{S}$ containing the ephemeral state Δ inside of it.

Lemma 2 (Consumption)**.**

- *If* $\mathcal{E} : \mathcal{S}[\mathsf{eval\,E}] \leadsto^*_{\Sigma\mathsf{t}} \mathsf{abs\,T_f}$,
  then $\mathcal{E} \approx (\mathsf{t/eval}(\mathcal{D}_t);\ \mathcal{E}')$*, where* $\mathcal{D}_t : \mathsf{of\,E\,T}$ *and* $\mathcal{E}' : \mathcal{S}[\mathsf{abs\,T}] \leadsto^*_{\Sigma\mathsf{t}} \mathsf{abs\,T_f}$ *for some* T.

- *If* $\mathcal{E} : \mathcal{S}[\mathsf{retn\,V}] \leadsto^*_{\Sigma\mathsf{t}} \mathsf{abs\,T_f}$,
  then $\mathcal{E} \approx (\mathsf{t/retn}(\mathcal{D}_v \cdot \mathcal{D}_t);\ \mathcal{E}')$*, where* $\mathcal{D}_v : \mathsf{value\,V}$, $\mathcal{D}_t : \mathsf{of\,V\,T}$*, and* $\mathcal{E}' : \mathcal{S}[\mathsf{abs\,T}] \leadsto^*_{\Sigma\mathsf{t}} \mathsf{abs\,T_f}$ *for some* T.

- *If* $\mathcal{E} : \mathcal{S}[\mathsf{cont\,F}, \mathsf{abs\,T}] \leadsto^*_{\Sigma\mathsf{t}} \mathsf{abs\,T_f}$,
  then $\mathcal{E} \approx (\mathsf{t/cont}(\mathcal{D}_f);\ \mathcal{E}')$*, where* $\mathcal{D}_f : \mathsf{off\,F\,T\,T'}$ *and* $\mathcal{E}' : \mathcal{S}[\mathsf{abs\,T'}] \leadsto^*_{\Sigma\mathsf{t}} \mathsf{abs\,T_f}$ *for some* $\mathsf{T'}$.

Proof. Each of the statements can be proved independently by induction on the structure of $\mathcal{E}$. The eval and retn cases are quite similar to each other and are both simpler versions of the cont case, which we give below.

Case: $\mathcal{E} = \diamond$

This case cannot occur, as there is no way for $\mathcal{S}[\mathsf{cont\,F}, \mathsf{abs\,T}]$ to be equivalent to $(\mathsf{abs\,T_f})$.

Case: $\mathcal{E} = \mathsf{t/eval}(\mathcal{D}'_t);\ \mathcal{E}'$
$\mathcal{D}'_t : \mathsf{of\,E\,T_e}$,
$\mathcal{E}' : (\mathcal{S}'[\mathsf{abs\,T_e}] \leadsto^*_{\Sigma\mathsf{t}} \mathsf{abs\,T_f})$, and
$\mathcal{S}[\mathsf{cont\,F}, \mathsf{abs\,T}] \approx \mathcal{S}'[\mathsf{eval\,E}]$

Because $\mathcal{S}[\mathsf{cont\,F}, \mathsf{abs\,T}] \approx \mathcal{S}'[\mathsf{eval\,E}]$, it must be the case that both of these are also equivalent to $\mathcal{S}''[\mathsf{cont\,F}, \mathsf{abs\,T}][\mathsf{eval\,E}]$ for some $\mathcal{S}'$ and $\mathcal{E}' : (\mathcal{S}''[\mathsf{cont\,F}, \mathsf{abs\,T}][\mathsf{abs\,T_e}] \leadsto^*_{\Sigma_t} \mathsf{abs\,T_f})$.

$\mathcal{E}' \approx (\mathsf{t/cont}(\mathcal{D}_f);\ \mathcal{E}'')$	By i.h.
$\mathcal{D}_f : \mathsf{off\,E\,T\,T'}$	”
$\mathcal{E}'' : (\mathcal{S}''[\mathsf{abs\,T'}][\mathsf{abs\,T_e}] \leadsto^*_{\Sigma\mathsf{t}} \mathsf{abs\,T_f})$	”
$\mathcal{E} \approx (\mathsf{t/eval}(\mathcal{D}'_t);\ \mathsf{t/cont}(\mathcal{D}_f);\ \mathcal{E}'')$	By construction
$\mathcal{E} \approx (\mathsf{t/cont}(\mathcal{D}_f);\ \mathsf{t/eval}(\mathcal{D}'_t);\ \mathcal{E}'')$	By concurrent equivalence

Case: $\mathcal{E} = \mathsf{t/retn}(\mathcal{D}'_v \cdot \mathcal{D}'_t);\ \mathcal{E}'$

Similar to previous case.

Case: $\mathcal{E} = \mathsf{t/cont}(\mathcal{D}'_f);\ \mathcal{E}'$
$\mathcal{D}'_f : \mathsf{off\,F_1\,T_1\,T'_1}$,
$\mathcal{E}' : (\mathcal{S}'[\mathsf{abs\,T'_1}] \leadsto^*_{\Sigma\mathsf{t}} \mathsf{abs\,T_f})$, and
$\mathcal{S}[\mathsf{cont\,F}, \mathsf{abs\,T}] \approx \mathcal{S}'[\mathsf{cont\,F_1}, \mathsf{abs\,T_1}]$

Because $\mathcal{S}[\mathsf{cont\,F}, \mathsf{abs\,T}] \approx \mathcal{S}'[\mathsf{cont\,F_1}, \mathsf{abs\,T_1}]$, there are two possibilities. The first possibility is that $\mathcal{S}[\mathsf{abs\,T'}] \approx \mathcal{S}'[\mathsf{abs\,T'}]$, $\mathsf{F} = \mathsf{F_1}$, and $\mathsf{T} = \mathsf{T_1}$, in which case we are done.

The other possibility is $\mathcal{S}[\mathsf{cont\,F}, \mathsf{abs\,T}] \approx \mathcal{S}'[\mathsf{cont\,F_1}, \mathsf{abs\,T_1}] \approx \mathcal{S}''[\mathsf{cont\,F}, \mathsf{abs\,T}][\mathsf{cont\,F_1}, \mathsf{abs\,T_1}]$ for some $\mathcal{S}''$. In that case, $\mathcal{E}' : (\mathcal{S}''[\mathsf{cont\,F}, \mathsf{abs\,T}][\mathsf{abs\,T'_1}])$ and we proceed with the following reasoning:

$\mathcal{E}' \approx (\mathsf{t/cont}(\mathcal{D}_f); \mathcal{E}'')$ By i.h.
$\mathcal{D}_f : \mathsf{off\ E\ T\ T'}$ ”
$\mathcal{E}'' \approx (\mathcal{S}''[\mathsf{abs\ T'}][\mathsf{abs\ T'_1}])$ ”
$\mathcal{E} \approx (\mathsf{t/cont}(\mathcal{D}'_f);\ \mathsf{t/cont}(\mathcal{D}_f);\ \mathcal{E}'')$ By construction
$\mathcal{E} \approx (\mathsf{t/cont}(\mathcal{D}_f);\ \mathsf{t/cont}(\mathcal{D}'_f);\ \mathcal{E}'')$ By concurrent equivalence

This completes the proof. □

It's worth emphasizing what may be obvious: the proof of the consumption lemma is a bit more general than necessary. In the static semantics we have defined so far, $\mathcal{S} \leadsto^*_{\Sigma\mathsf{t}} \mathsf{abs\ T}$ means that $\mathcal{S}$ contains *precisely one* atomic proposition of the form $(\mathsf{eval\ E})$, $(\mathsf{retn\ E})$, or $(\mathsf{abs\ T})$, which means that most of the cases we consider are actually impossible. However, this impossibility is *also* a fact that must be proven by induction on the structure of expressions. The proof we have given is both straightforward in its own right and allows the proof of preservation to "scale" to specifications with parallel evaluation (see Section 5.3). Furthermore, the theorem *statement* is not at all complicated as a result of handling the general case. If we wanted to write a version of the consumption lemma for the special case of sequential SSOS specifications, the only difference is that we would write "If $\mathcal{E} : (\mathcal{S}, \mathsf{eval\ E} \leadsto^*_{\Sigma\mathsf{t}} \mathsf{abs\ T_f})$, then..." instead of "If $\mathcal{E} : (\mathcal{S}[\mathsf{eval\ E}] \leadsto^*_{\Sigma\mathsf{t}} \mathsf{abs\ T_f})$, then..." because we know that $\mathsf{eval\ E}$ appears only on the right-hand edge of the ephemeral context.

Now we can consider the actual proof of preservation:

Proof of the Preservation Lemma (Lemma 1). The proof proceeds by case analysis on the transition $\mathcal{T} : (\mathcal{S} \leadsto_{\Sigma\mathsf{e}} \mathcal{S}')$, followed in each case by applying the consumption lemma to the expression $\mathcal{E} : (\mathcal{S} \leadsto^*_{\Sigma\mathsf{t}} \mathsf{abs\ T_f})$.

Case: $\mathcal{T} = \mathsf{e/lam} : \mathcal{S}[\mathsf{eval}(\mathsf{lam}\ \lambda x.\, \mathsf{E}\, x)] \leadsto_{\Sigma\mathsf{e}} \mathcal{S}[\mathsf{retn}(\mathsf{lam}\ \lambda x.\, \mathsf{E}\, x)]$

$\mathcal{E} \approx (\mathsf{t/eval}(\mathcal{D}_t);\ \mathcal{E}')$ By consumption lemma
$\mathcal{D}_t : \mathsf{of}\ (\mathsf{lam}\ \lambda x.\, \mathsf{E}\, x)\ \mathsf{T}$ ”
$\mathcal{E}' : (\mathcal{S}[\mathsf{abs\ T}] \leadsto^*_{\Sigma\mathsf{t}} \mathsf{abs\ T_f})$ ”
$(\mathsf{t/retn}(\mathsf{v/lam} \cdot \mathcal{D}_t);\ \mathcal{E}') : (\mathcal{S}[\mathsf{retn}(\mathsf{lam}\ \lambda x.\, \mathsf{E}\, x)] \leadsto^*_{\Sigma\mathsf{t}} \mathsf{abs\ T_f})$ By construction

Case: $\mathcal{T} = \mathsf{e/app} : \mathcal{S}[\mathsf{eval}(\mathsf{app\ E_1\ E_2})] \leadsto_{\Sigma\mathsf{e}} \mathcal{S}[\mathsf{cont}(\mathsf{app_1\ E_2}), \mathsf{eval\ E_1}]$

$\mathcal{E} \approx (\mathsf{t/eval}(\mathcal{D}_t);\ \mathcal{E}')$ By consumption lemma
$\mathcal{D}_t : \mathsf{of}\ (\mathsf{app\ E_1\ E_2})\ \mathsf{T}$ ”
$\mathcal{E}' : \mathcal{S}[\mathsf{abs\ T}] \leadsto^*_{\Sigma\mathsf{t}} \mathsf{abs\ T_f}$ ”
$\mathcal{D}_t = \mathsf{of/app}\ \mathcal{D}_1\ \mathcal{D}_2$ By inversion on $\mathcal{D}_t$
$\mathcal{D}_1 : \mathsf{of\ E_1\ (arrow\ T'\ T)}$ ”
$\mathcal{D}_2 : \mathsf{of\ E_2\ T'}$ ”
$(\mathsf{t/cont}(\mathsf{off/app_1}\ \mathcal{D}_2);\ \mathcal{E}') : (\mathcal{S}[\mathsf{cont}(\mathsf{app_1\ E_2}), \mathsf{abs}(\mathsf{arrow\ T'\ T})] \leadsto^*_{\Sigma\mathsf{t}} \mathsf{abs\ T_f})$ By construction
$(\mathsf{t/eval}(\mathcal{D}_1);\ \mathsf{t/cont}(\mathsf{off/app_1}\ \mathcal{D}_2);\ \mathcal{E}') : (\mathcal{S}[\mathsf{cont}(\mathsf{app_1\ E_2}), \mathsf{eval\ E_1}] \leadsto^*_{\Sigma\mathsf{t}} \mathsf{abs\ T_f})$ By construction

Case: $\mathcal{T} = \mathsf{e/app_1} : \mathcal{S}[\mathsf{cont}(\mathsf{app_1\ E_2}), \mathsf{retn\ V_1}] \leadsto_{\Sigma\mathsf{e}} \mathcal{S}[\mathsf{cont}(\mathsf{app_2\ V_1}), \mathsf{eval\ E_2}]$

$\mathcal{E} \approx (\mathsf{t/retn}(\mathcal{D}_v \cdot \mathcal{D}_t);\ \mathcal{E}')$ By consumption lemma
$\mathcal{D}_v : \mathsf{value\,V_1}$ ”
$\mathcal{D}_t : \mathsf{of\,V_1\,T}$ ”
$\mathcal{E}' : (\mathcal{S}[\mathsf{cont(app_1\,E_2), abs\,T}] \leadsto^*_{\Sigma\mathsf{t}} \mathsf{abs\,T_f})$ ”
$\mathcal{E}' \approx \mathsf{t/cont}(\mathcal{D}_f);\ \mathcal{E}''$ By consumption lemma
$\mathcal{D}_f : \mathsf{off\,(app_1\,E_2)\,T\,T'}$ ”
$\mathcal{E}'' : (\mathcal{S}[\mathsf{abs\,T'}] \leadsto^*_{\Sigma\mathsf{t}} \mathsf{abs\,T_f})$ ”
$\mathcal{D}_f = \mathsf{off/app_1}(\mathcal{D}'_t)$ By inversion on $\mathcal{D}_f$
$\mathsf{T} = \mathsf{arrow\,T_0\,T'}$ ”
$\mathcal{D}'_t : \mathsf{of\,E_2\,T_0}$ ”
$(\mathsf{t/cont(off/app_2}\,\mathcal{D}_v\,\mathcal{D}_t);\ \mathcal{E}'') : (\mathcal{S}[\mathsf{cont(app_2\,V_1), abs\,T_0}] \leadsto^*_{\Sigma\mathsf{t}} \mathsf{abs\,T_f})$ By construction
$(\mathsf{t/eval}(\mathcal{D}'_t);\ \mathsf{t/cont(off/app_2}\,\mathcal{D}_v\,\mathcal{D}_t);\ \mathcal{E}'') : (\mathcal{S}[\mathsf{cont(app_2\,V_1), eval\,E_2}] \leadsto^*_{\Sigma\mathsf{t}} \mathsf{abs\,T_f})$
By construction

Case: $\mathcal{T} = \mathsf{e/app_2} : \mathcal{S}[\mathsf{cont(app_2(lam\,\lambda x.\,E\,x)), retn\,V_2}] \leadsto_{\Sigma\mathsf{e}} \mathcal{S}[\mathsf{eval(E\,V_2)}]$

$\mathcal{E} \approx \mathsf{t/retn}(\mathcal{D}_v \cdot \mathcal{D}_t);\ \mathcal{E}'$ By consumption lemma
$\mathcal{D}_t : \mathsf{of\,V_2\,T}$ ”
$\mathcal{E}' : (\mathcal{S}[\mathsf{cont(app_2(lam\,\lambda x.\,E\,x)), abs\,T}] \leadsto^*_{\Sigma\mathsf{t}} \mathsf{abs\,T_f})$ ”
$\mathcal{E}' \approx \mathsf{t/cont}(\mathcal{D}_f);\ \mathcal{E}''$ By consumption lemma
$\mathcal{D}_f : \mathsf{off\,(app_2(lam\,\lambda x.\,E\,x))\,T\,T'}$ ”
$\mathcal{E}'' : (\mathcal{S}[\mathsf{abs\,T'}] \leadsto^*_{\Sigma\mathsf{t}} \mathsf{abs\,T_f})$ ”
$\mathcal{D}_f = \mathsf{off/app_2}\,\mathcal{D}'_v\,(\lambda x.\,\lambda d.\,\mathcal{D}'_t\,x\,d)$ By inversion on $\mathcal{D}_f$
$(\lambda x.\,\lambda d.\,\mathcal{D}'_t\,x\,d) : \Pi x.\,\mathsf{of}\,x\,\mathsf{T} \to \mathsf{of}\,(\mathsf{E}\,x)\,\mathsf{T'}$ ”
$\mathcal{D}'_t\,\mathsf{V_2}\,\mathcal{D}_t : \mathsf{of\,(E\,V_2)\,T'}$ By substitution
$(\mathsf{t/eval}(\mathcal{D}'_t\,\mathsf{V_2}\,\mathcal{D}_t);\ \mathcal{E}'') : (\mathcal{S}[\mathsf{eval(E\,V_2)}] \leadsto^*_{\Sigma\mathsf{t}} \mathsf{abs\,T_f})$ By construction

The remaining cases for $\mathsf{e/nat}$, $\mathsf{e/plus}$, $\mathsf{e/plus_1}$, and $\mathsf{e/plus_2}$ are similar. □

5.2.3 Progress

Preservation theorems, and in particular big-step preservation theorems, have traditionally been the primary correctness criteria for language specifications in substructural logics [1]. However, preservation alone is insufficient to ensure language safety as specified by Definition 1. The preservation lemma ensures that the invariant is maintained; the progress lemma ensures that the invariant actually establishes safety.

The critical lemma for progress, the analogue to the consumption lemma in the preservation lemma, is actually quite a bit more general. It relies on a general property of the static semantics, namely that it is *contractive*, because each rule has at most one conclusion.

Definition 2 (Contractive signature)**.** *If every rule in Σ has at most one atomic proposition in the conclusion, and that conclusion is an ordered or linear atomic proposition, then Σ is contractive.*

When a signature is contractive, then different pieces of the context at the end of an expression can be traced backwards to different pieces at the beginning of the expression.

Lemma 3 (Splitting)**.** *If Σ is contractive and $\mathcal{E} : (\mathcal{S} \leadsto^*_{\Sigma} \mathcal{S}'_1, \mathcal{S}'_2)$, then $\mathcal{S} \approx \mathcal{S}_1, \mathcal{S}_2$, and there exist expressions $\mathcal{E}_1$ and $\mathcal{E}_2$ such that $\mathcal{E}_1 : (\mathcal{S}_1 \leadsto^*_{\Sigma} \mathcal{S}'_1)$ and $\mathcal{E}_2 : (\mathcal{S}_2 \leadsto^*_{\Sigma} \mathcal{S}'_2)$, where $\mathcal{E}$ has the same number of transitions as $\mathcal{E}_1$ and $\mathcal{E}_2$ combined.*

Proof. By induction on the structure of $\mathcal{E}$. If $\mathcal{E} = \diamond$, the result is immediate, and if $\mathcal{E} = (\mathcal{E}';\ \mathcal{T})$ then we apply the induction hypothesis on $\mathcal{E}'$ and add $\mathcal{T}$ to the end of the appropriate smaller expression: $\mathcal{E}_1$ if the conclusion of $\mathcal{T}$ was bound in $\mathcal{S}_1$, $\mathcal{E}_2$ otherwise. $\square$

A stronger version of the splitting lemma would also establish that $\mathcal{E} \approx (\mathcal{E}_1;\ \mathcal{E}_2) \approx (\mathcal{E}_2;\ \mathcal{E}_1)$, but this lemma is sufficient to allow us to prove the progress lemma.

Lemma 4 (Progress)**.** *If* $\mathcal{S} \leadsto^*_{\Sigma t} \mathsf{abs\,T}$ *and* $\mathcal{S}$ *contains no resources* $\mathsf{abs\,T}$*, then either* $\mathcal{S} \leadsto_{\Sigma e} \mathcal{S}'$ *for some* $\mathcal{S}'$ *or else* $\mathcal{S} = \mathsf{retn\,V}$ *for some value* V *with type* T.

At a high level, this proof works by showing that any state $\mathcal{S}$ that satisfies the typing invariant must be one of the following:

- $\mathcal{S} = \mathsf{retn\,V}$, a safe state that takes no step,
- $\mathcal{S} = \mathsf{eval\,E}$, in which case we can always take a step because $\mathsf{eval\,E}$ is an active proposition,
- $\mathcal{S} = (\mathsf{comp\,F}, \mathcal{S}')$ where $\mathcal{S}' \leadsto_{\Sigma e} \mathcal{S}''$; in this case, the larger state takes a step as well, or
- $\mathcal{S} = (\mathsf{comp\,F}, \mathsf{retn\,V})$, in which case we use canonical forms lemmas (which are standard, and which we therefore use without proof) to ensure that we can perform a reduction.

Proof. By induction on the number of transitions in $\mathcal{E}$. We note that $\mathcal{E} \neq \diamond$, because that would mean that $\mathcal{S} = \mathsf{abs\,T}$, contradicting the premise that $\mathcal{S}$ does not contain any atomic propositions of the form $\mathsf{abs\,T}$. So we can assume $\mathcal{E} = (\mathcal{E}';\ \mathcal{T})$ and do case analysis on the form of the transition $\mathcal{T}$.

Case: $\mathcal{T} = \mathsf{t/retn}(\mathcal{D}_v \cdot \mathcal{D}_t)$ and $\mathcal{E}' : \mathcal{S} \leadsto^*_{\Sigma t} \mathsf{retn\,V}$
where $\mathcal{D}_v : \mathsf{value\,V}$ and $\mathcal{D}_t : \mathsf{of\,E\,T}$.

There are no transitions in Σt that can appear as the last transition in $\mathcal{E}'$, so $\mathcal{E}' = \diamond$ and $\mathcal{S} = \mathsf{retn\,V}$; therefore, we are done.

Case: $\mathcal{T} = \mathsf{t/eval}(\mathcal{D}_t)$ and $\mathcal{E}' : \mathcal{S} \leadsto^*_{\Sigma t} \mathsf{eval\,E}$.

There are no transitions in Σt that can appear as the last transition in $\mathcal{E}'$, so $\mathcal{E}' = \diamond$ and $\mathcal{S} = \mathsf{eval\,E}$. We proceed by case analysis on the structure of E.

Subcase: $\mathsf{E} = \mathsf{lam}\ \lambda x.\, \mathsf{E}\,x$	—	$\mathsf{e/lam}$:	$\mathsf{eval}(\mathsf{lam}\ \lambda x.\, \mathsf{E}\,x) \leadsto_{\Sigma e} \mathsf{retn}(\mathsf{lam}\ \lambda x.\, \mathsf{E}\,x)$
Subcase: $\mathsf{E} = \mathsf{app}\,\mathsf{E}_1\,\mathsf{E}_2$	—	$\mathsf{e/app}$:	$\mathsf{eval}(\mathsf{app}\,\mathsf{E}_1\,\mathsf{E}_2) \leadsto_{\Sigma e} (\mathsf{cont}(\mathsf{app}_1\,\mathsf{E}_2), \mathsf{eval}\,\mathsf{E}_1)$
Subcase: $\mathsf{E} = \mathsf{n\,N}$	—	$\mathsf{e/n}$:	$\mathsf{eval}(\mathsf{n\,N}) \leadsto_{\Sigma e} \mathsf{retn}\,(\mathsf{n\,N})$
Subcase: $\mathsf{E} = \mathsf{plus}\,\mathsf{E}_1\,\mathsf{E}_2$	—	$\mathsf{e/plus}$:	$\mathsf{eval}(\mathsf{plus}\,\mathsf{E}_1\,\mathsf{E}_2) \leadsto_{\Sigma e} (\mathsf{cont}(\mathsf{plus}_1\,\mathsf{E}_2), y_2 : \mathsf{eval}\,\mathsf{E}_1)$

Case: $\mathcal{T} = \mathsf{t/cont}(\mathcal{D}_f)$ and $\mathcal{E}' : (\mathcal{S} \leadsto^*_{\Sigma t} \mathsf{cont\,F}, \mathsf{abs\,T}')$, where $\mathcal{D}_f : \mathsf{off\,F\,T'\,T}$.

By the splitting lemma, $\mathcal{S} \approx \mathcal{S}_1, \mathcal{S}_2$ and there exist two expressions $\mathcal{E}_1 : (\mathcal{S}_1 \leadsto^*_{\Sigma t} \mathsf{cont\,F})$ and $\mathcal{E}_2 : (\mathcal{S}_2 \leadsto^*_{\Sigma t} \mathsf{abs\,T}')$. There are no transitions in Σt that can appear as the last transition in $\mathcal{E}_1$, so $\mathcal{E}_1 = \diamond$ and $\mathcal{S}_1 = \mathsf{cont\,F}$.

We then apply the induction hypothesis on $\mathcal{E}_2$ (which has one transition less than $\mathcal{E}$, justifying the call to the induction hypothesis). If $\mathcal{T} : \mathcal{S}_2 \leadsto_{\Sigma e} \mathcal{S}_2'$, then $\mathcal{T} : (\mathsf{cont\,F},\ \mathcal{S}_2 \leadsto_{\Sigma e} \mathsf{cont\,F},\ \mathcal{S}_2')$ by the rules for typing transitions,[9] and we are done.

[9]Generically we could call this a *frame property* of the framework.

Otherwise, we have $\mathcal{S}_2 = \mathsf{retn\,V}$, $\mathcal{D}_v : \mathsf{value\,V}$, and $\mathcal{D}_t : \mathsf{of\,V\,T'}$. We proceed by case analysis on the proof term $\mathcal{D}_f$ establishing that the frame F is well typed.

Subcase: $\mathcal{D}_f = \mathsf{off/app_1}\,\mathcal{D}_{t2}$, so $\mathsf{F} = \mathsf{app_1\,E_2}$.
$\mathsf{e/app_1} \;:\; (\mathsf{cont(app_1\,E_2)},\ \mathsf{retn\,V}) \rightsquigarrow_{\Sigma\mathsf{e}} (\mathsf{cont(app_2\,V)},\ \mathsf{eval\,E_2})$.

Subcase: $\mathcal{D}_f = \mathsf{off/app_2}\,\mathcal{D}_{v1}\,\mathcal{D}_{t1}$, so $\mathsf{F} = \mathsf{app_2\,V_1}$,
$\mathcal{D}_{v1} : \mathsf{value\,V_1}$, and $\mathcal{D}_{t1} : \mathsf{of\,E_1\,(arrow\,T'\,T)}$.
By the canonical forms lemma on $\mathcal{D}_{v1}$ and $\mathcal{D}_{t1}$, $\mathsf{V_1} = \mathsf{lam}\,\lambda\mathsf{x}.\,\mathsf{E_0\,x}$.
$\mathsf{e/app_2} \;:\; (\mathsf{cont(app_2(lam}\,\lambda\mathsf{x}.\,\mathsf{E_0\,x})),\ \mathsf{retn\,V}) \rightsquigarrow_{\Sigma\mathsf{e}} \mathsf{eval(E_0\,V)}$.

Subcase: $\mathcal{D}_f = \mathsf{off/plus_1}\,\mathcal{D}_{t2}$, so $\mathsf{F} = \mathsf{plus_1\,E_2}$.
$\mathsf{e/plus_1} \;:\; (\mathsf{cont(plus_1\,E_2)},\ \mathsf{retn\,V})) \rightsquigarrow_{\Sigma\mathsf{e}} (\mathsf{cont(plus_2\,V)},\ \mathsf{eval\,E_2})$.

Subcase: $\mathcal{D}_f = \mathsf{off/plus_2}\,\mathcal{D}_{v1}\,\mathcal{D}_{t1}$, so $\mathsf{F} = \mathsf{plus_2\,V_1}$ and $\mathsf{T} = \mathsf{T'} = \mathsf{tnat}$
$\mathcal{D}_{v1} : \mathsf{value\,E_1}$, and $\mathcal{D}_{t1} : \mathsf{of\,E_1\,tnat}$.
By the canonical forms lemma on $\mathcal{D}_{v1}$ and $\mathcal{D}_{t1}$, $\mathsf{V_1} = \mathsf{n\,N_1}$.
By the canonical forms lemma on $\mathcal{D}_v$ and $\mathcal{D}_t$, $\mathsf{V} = \mathsf{n\,N_2}$.
By the effectiveness of addition on $\mathsf{N_1}$ and $\mathsf{N_2}$, there exists a natural number $\mathsf{N_3}$ and a proof term $\mathcal{D}_a : \mathsf{add\,N_1\,N_2\,N_3}$.
$\mathsf{e/plus_2}(\mathcal{D}_a) \;:\; (\mathsf{cont(app_2(n\,N_1))},\ \mathsf{retn(n\,N_2)}) \rightsquigarrow_{\Sigma\mathsf{e}} \mathsf{retn(n\,N_3)}$.

This completes the proof; we have assumed standard canonical forms lemmas and the effectiveness of addition, provable by induction on the structure of $\mathsf{N_1}$. □

5.3 Parallel evaluation

One way in which parallel evaluation has been incorporated into functional programming languages is by allowing pairs (or, more generally, tuples) to evaluate in parallel [9]. In this section we will consider $\mathcal{L}_2$, the first modular extension to the language $\mathcal{L}_1$ with parallel pairs. The syntax and typing rules pairs are a straightforward and standard addition:

$\mathsf{pair} : \mathsf{exp} \to \mathsf{exp} \to \mathsf{exp}.$
$\mathsf{fst} : \mathsf{exp} \to \mathsf{exp}.$
$\mathsf{snd} : \mathsf{exp} \to \mathsf{exp}.$

$\mathsf{v/pair} : \mathsf{value\,E_1} \to \mathsf{value\,E_2} \to \mathsf{value(pair\,E_1\,E_2)}.$

$\mathsf{pairtp} : \mathsf{tp} \to \mathsf{tp} \to \mathsf{tp}.$

$\mathsf{of/pair} : \mathsf{of\,E_1\,T_1} \to \mathsf{of\,E_2\,T_2} \to \mathsf{of\,(pair\,E_1E_2)\,(pairtp\,T_1\,T_2)}.$

$\mathsf{of/fst} : \ \mathsf{of\,E\,(pairtp\,T_1\,T_2)} \to \mathsf{of\,(fst\,E)\,T_1}.$

$\mathsf{of/snd} : \ \mathsf{of\,E\,(pairtp\,T_1\,T_2)} \to \mathsf{of\,(snd\,E)\,T_2}.$

5.3.1 Dynamic semantics

Nothing in the previous signature indicates that pairs are any more interesting than the language features we have already presented. It is in the specification of the dynamic semantics that we add a new capability to the language: *parallel evaluation*. Our dynamic semantics will evaluate both parts of a pair in parallel. First, we add a new type of latent ordered proposition, $(\mathsf{cont}_2\, \mathsf{F})$, representing a frame waiting on *two* values to be returned to it.

$$\mathsf{cont}_2 : \mathsf{frame} \rightarrow \mathsf{type}_o.$$

$$\mathsf{fst}_0 : \mathsf{frame}.$$
$$\mathsf{snd}_0 : \mathsf{frame}.$$
$$\mathsf{pair}_0 : \mathsf{frame}.$$

$$\mathsf{e/fst} : \quad \mathsf{eval}(\mathsf{fst}\, \mathsf{E}) \twoheadrightarrow \mathsf{cont}\, \mathsf{fst}_0 \cdot \mathsf{eval}\, \mathsf{E}.$$

$$\mathsf{e/fst}_0 : \quad \mathsf{cont}\, \mathsf{fst}_0 \cdot \mathsf{retn}(\mathsf{pair}\, \mathsf{V}_1\, \mathsf{V}_2) \twoheadrightarrow \mathsf{retn}\, \mathsf{V}_1.$$

$$\mathsf{e/snd} : \quad \mathsf{eval}(\mathsf{snd}\, \mathsf{E}) \twoheadrightarrow \mathsf{cont}\, \mathsf{snd}_0 \cdot \mathsf{eval}\, \mathsf{E}.$$

$$\mathsf{e/snd}_0 : \quad \mathsf{cont}\, \mathsf{snd}_0 \cdot \mathsf{retn}(\mathsf{pair}\, \mathsf{V}_1\, \mathsf{V}_2) \twoheadrightarrow \mathsf{retn}\, \mathsf{V}_2.$$

$$\mathsf{e/pair} : \quad \mathsf{eval}(\mathsf{pair}\, \mathsf{E}_1\, \mathsf{E}_2) \twoheadrightarrow \mathsf{cont}_2\, \mathsf{pair}_0 \cdot \mathsf{eval}\, \mathsf{E}_1 \cdot \mathsf{eval}\, \mathsf{E}_2.$$

$$\mathsf{e/pair}_0 : \quad \mathsf{cont}_2\, \mathsf{pair}_0 \cdot \mathsf{retn}\, \mathsf{V}_1 \cdot \mathsf{retn}\, \mathsf{V}_2 \twoheadrightarrow \mathsf{retn}(\mathsf{pair}\, \mathsf{V}_1\, \mathsf{V}_2).$$

Our specification no longer corresponds to an on-paper description of a stack machine with one expression evaluating on top of the stack: with this change, we have made the substructural context a treelike structure where multiple independent groups of propositions may be able to transition at any given time.

5.3.2 Static semantics

Beyond the typing rules for pairs given at the beginning of this section, we need a new typing judgment for continuation frames waiting on two values:

$$\mathsf{off2} : \mathsf{frame} \rightarrow \mathsf{tp} \rightarrow \mathsf{tp} \rightarrow \mathsf{tp} \rightarrow \mathsf{type}.$$

The fst_0 and snd_0 frames lead to new cases for the regular frame typing rule, and we give a parallel frame typing rule for the pair_0 frame:

$$\mathsf{off/fst}_0 : \mathsf{off}\, \mathsf{fst}_0\, (\mathsf{pairtp}\, \mathsf{T}_1\, \mathsf{T}_2) \mathsf{T}_1.$$

$$\mathsf{off/snd}_0 : \mathsf{off}\, \mathsf{fst}_0\, (\mathsf{pairtp}\, \mathsf{T}_1\, \mathsf{T}_2) \mathsf{T}_2.$$

$$\mathsf{off2/pair}_0 : \mathsf{off2}\, \mathsf{pair}_0\, \mathsf{T}_1\, \mathsf{T}_2\, (\mathsf{pairtp}\, \mathsf{T}_1\, \mathsf{T}_2).$$

And a rule in the static semantics explaining how parallel frames interact with $(\mathsf{abs}\, \mathsf{T})$:

$$\begin{aligned} \mathsf{t/cont}_2 : {} & \mathsf{cont}_2\, \mathsf{F} \cdot \mathsf{abs}\, \mathsf{T}_1 \cdot \mathsf{abs}\, \mathsf{T}_2 \cdot !\mathsf{off2}\, \mathsf{F}\, \mathsf{T}_1\, \mathsf{T}_2\, \mathsf{T}' \\ & \twoheadrightarrow \mathsf{abs}\, \mathsf{T}'. \end{aligned}$$

Those three declarations entirely describe the static semantics of parallel evaluation. The typing rule $\mathsf{off2/pair}_0$ and the rewriting rule $\mathsf{t/cont}_2$ could easily be merged, but by separating them it is possible to incorporate additional parallel features into the language in a modular way.

5.3.3 Safety

We have just seen that it is possible to modularly extend both the static and dynamic semantics of $\mathcal{L}_1$ to obtain the language $\mathcal{L}_2$ with parallel evaluation of pairs. We are also able to straightforwardly extend the safety proof of $\mathcal{L}_1$ to incorporate parallel evaluation. We have to do the following things to extend the existing proof:

- Add a new consumption lemma:
If $\mathcal{E} : \mathcal{S}[\mathsf{cont}_2\,\mathsf{F}, \mathsf{abs}\,\mathsf{T}_1, \mathsf{abs}\,\mathsf{T}_2] \leadsto^*_{\Sigma\mathsf{t}} \mathsf{abs}\,\mathsf{T}_\mathsf{f}$,
then $\mathcal{E} \approx (\mathsf{t/cont}(\mathcal{D}_t);\ \mathcal{E}')$*, where* $\mathcal{D}_f : \mathsf{off2}\,\mathsf{F}\,\mathsf{T}_1\,\mathsf{T}_2\,\mathsf{T}'$ *and* $\mathcal{E}' : \mathcal{S}[\mathsf{abs}\,\mathsf{T}'] \leadsto^*_{\Sigma\mathsf{t}} \mathsf{abs}\,\mathsf{T}_\mathsf{f}$ *for some* T'.

- Add a case for $\mathsf{t/cont}_2$ to the proof of each of the other consumption lemmas.

- Add cases for $\mathsf{e/pair}$ and $\mathsf{e/pair}_0$, $\mathsf{e/fst}$, etc. to the proof of preservation lemma.

- Add new subcases for $\mathsf{E} = \mathsf{pair}\,\mathsf{E}_1\,\mathsf{E}_2$, $\mathsf{E} = \mathsf{fst}\,\mathsf{E}$, and $\mathsf{E} = \mathsf{snd}\,\mathsf{E}$ to the second case of the progress lemma where it is the case that $\mathcal{S} = \mathsf{eval}\,\mathsf{E}$.

- Add new subcases for $\mathcal{D}_f = \mathsf{off/fst}_0$ and $\mathcal{D}_f = \mathsf{off/snd}_0$ to the third case of the progress lemma, both of which will appeal to a canonical forms lemma.

- Add a case for $\mathcal{T} = \mathsf{t/cont}_2(\mathcal{D}_f)$ to the proof of the progress lemma.

The essential structure of the safety proof is preserved under the extension of the language with parallel pairs. This is significant because, with the exception of the proof of the consumption lemma, the safety proof for $\mathcal{L}_1$ did not need to explicitly prepare for the possibility of parallel evaluation.

5.4 Exceptions

Exceptions and exception handling are another example of a feature that we can add to $\mathcal{L}_1$ or $\mathcal{L}_2$ in a modular fashion, though there is an important caveat. While we can add parallel evaluation *or* exceptions to $\mathcal{L}_1$ without reconsidering any aspects of the original specification, we cannot extend the base language with *both* exceptions and parallelism without considering their interaction. This should not be seen as a troublesome lack of modularity, however — rather, the SSOS specification in this section illustrates that exceptions and parallel evaluation are not truly orthogonal language features. Seen in this light, it is natural that we must clearly describe how the features interact.

The syntax of exceptions includes error, which raises an exception, and $\mathsf{trycatch}\,\mathsf{E}\,\mathsf{E}'$, which includes a primary expression E and a secondary expression E' that is evaluated only if the evaluation of E returns an error. It would not be much more difficult to allow exceptions to carry a value, but we do not do so here.

$\mathsf{error} : \mathsf{exp}.$
$\mathsf{trycatch} : \mathsf{exp} \to \mathsf{exp} \to \mathsf{exp}.$

$\mathsf{of/error} : \mathsf{of}\,\mathsf{error}\,\mathsf{T}.$

$\mathsf{of/trycatch} : \mathsf{of}\,\mathsf{E}\,\mathsf{T} \to \mathsf{of}\,\mathsf{E}'\,\mathsf{T} \to \mathsf{of}\,(\mathsf{trycatch}\,\mathsf{E}\,\mathsf{E}')\,\mathsf{T}.$

5.4.1 Dynamic semantics

To give the dynamic semantics of exceptions and exception handling, we introduce two new ordered atomic propositions. The first, $\mathsf{handle}\,\mathsf{E}'$, represents an exception-handling stack frame. The second, raise, represents an uncaught exception.

$\mathsf{raise} : \mathsf{type}_o.$
$\mathsf{handle} : \mathsf{exp} \to \mathsf{type}_o.$

$\mathsf{e/raise} : \quad \mathsf{eval}\,\mathsf{error} \twoheadrightarrow \mathsf{raise}.$

$$\begin{aligned}
&\mathsf{e/trycatch} : \mathsf{eval}(\mathsf{trycatch}\,\mathsf{E}\,\mathsf{E}') \twoheadrightarrow \mathsf{handle}\,\mathsf{E}' \cdot \mathsf{eval}\,\mathsf{E}.\\
&\mathsf{e/vhandle} : \mathsf{handle}\,\mathsf{E}' \cdot \mathsf{retn}\,\mathsf{V} \twoheadrightarrow \mathsf{retn}\,\mathsf{V}.\\
&\mathsf{e/xhandle} : \mathsf{handle}\,\mathsf{E}' \cdot \mathsf{raise} \twoheadrightarrow \mathsf{eval}\,\mathsf{E}'.\\
&\mathsf{e/xcont} : \mathsf{cont}\,\mathsf{F} \cdot \mathsf{raise} \twoheadrightarrow \mathsf{raise}.
\end{aligned}$$

As mentioned before, we must deal explicitly with the interaction of parallel evaluation and exception handling by explaining how exceptions interact with cont_2 frames. The following is one possibility in which both sub-computations must terminate before an error is returned.

$$\begin{aligned}
&\mathsf{e/xvcont} : \mathsf{cont}_2\,\mathsf{F} \cdot \mathsf{raise} \cdot \mathsf{retn}\mathsf{V} \twoheadrightarrow \mathsf{raise}.\\
&\mathsf{e/vxcont} : \mathsf{cont}_2\,\mathsf{F} \cdot \mathsf{retn}\mathsf{V} \cdot \mathsf{raise} \twoheadrightarrow \mathsf{raise}.\\
&\mathsf{e/xxcont} : \mathsf{cont}_2\,\mathsf{F} \cdot \mathsf{raise} \cdot \mathsf{raise} \twoheadrightarrow \mathsf{raise}.
\end{aligned}$$

Another obvious candidate for the interaction between parallel evaluation of pairs and exceptions would be for a pair to immediately raise an exception if either of its components raises an exception. However, it is not obvious how to gracefully implement this in the ordered SSOS style we have presented so far. This is a symptom of a broader problem, namely that ordered SSOS specifications don't, in general, handle non-local transfer of control particularly well. This is a limitation of the specification style and not the framework; a different style of specification known as *linear destination passing* could easily handle the more general specification [24].

5.4.2 Static semantics

The static semantics of exception handling are extremely simple; there are just two rules. An uncaught exception has any type, and an exception handler must have the same type as its sub-computation.

$$\begin{aligned}
&\mathsf{t/raise} : \mathsf{raise} \twoheadrightarrow \mathsf{abs}\,\mathsf{T}.\\
&\mathsf{t/handle} : \mathsf{handle}\,\mathsf{E} \cdot \mathsf{abs}\,\mathsf{T} \cdot !\mathsf{of}\,\mathsf{E}\,\mathsf{T} \twoheadrightarrow \mathsf{abs}\,\mathsf{T}.
\end{aligned}$$

5.4.3 Safety

The existing structure of the safety proof also extends straightforwardly to handle the addition of exceptions to the language; the most significant change is to the definition of safety itself, as a safe state either steps, is a returned value $\mathsf{retn}\,\mathsf{V}$, or is an unhandled exception raise.

5.5 Another view: context generators

In the discussion of safety in Section 5.2, we needed a special "context invariant" lemma saying that the dynamic semantics preserves the property that a substructural context contains no $(\mathsf{abs}\,\mathsf{T})$ propositions. Another way of specifying this context invariant is in terms of a transition system that *generates* the context. The degenerate "rewriting rules" that capture the context invariant for parallel evaluation are as follows:

$$\begin{aligned}
&\mathsf{gen/eval} : \forall \mathsf{E}.\,\mathsf{eval}\,\mathsf{E}.\\
&\mathsf{gen/retn} : \forall \mathsf{E}.\,\mathsf{retn}\,\mathsf{E}.\\
&\mathsf{gen/cont} : \forall \mathsf{F}.\,\mathsf{cont}\,\mathsf{F}.\\
&\mathsf{gen/cont_2} : \forall \mathsf{F}.\,\mathsf{cont}_2\,\mathsf{F}.
\end{aligned}$$

This changes the picture of safety presented in Section 5.2 to the following picture, which shows the two invariants given by the context generating rules and the static semantics. The context generating rules construct the context, and the static semantics then analyze it.

$$\begin{tikzcd}
\cdot \arrow[d, squiggly, "\Sigma\mathsf{gen}", "*"'] & \cdot \arrow[d, squiggly, dotted, "\Sigma\mathsf{gen}", "*"'] & & \cdot \arrow[d, squiggly, dotted, "\Sigma\mathsf{gen}", "*"'] & \cdot \arrow[d, squiggly, dotted, "\Sigma\mathsf{gen}", "*"'] & \\
\mathsf{eval}\,\mathsf{E} \arrow[r, squiggly, "\Sigma\mathsf{e}"] \arrow[d, squiggly, "\Sigma\mathsf{t}", "*"'] & \mathcal{S}_1 \arrow[d, squiggly, dotted, "\Sigma\mathsf{t}", "*"'] & \cdots & \mathcal{S}_n \arrow[r, squiggly, "\Sigma\mathsf{e}"] \arrow[d, squiggly, dotted, "\Sigma\mathsf{t}", "*"'] & \mathcal{S} \arrow[r, squiggly, dotted, "\Sigma\mathsf{e}"] \arrow[r, phantom, "="', shift right=3] \arrow[d, squiggly, dotted, "\Sigma\mathsf{t}", "*"'] & \mathcal{S}' \text{ or } \mathsf{retn}\,\mathsf{V} \\
\mathsf{abs}\,\mathsf{T} & \mathsf{abs}\,\mathsf{T} & & \mathsf{abs}\,\mathsf{T} & \mathsf{abs}\,\mathsf{T} &
\end{tikzcd}$$

This account of context generating rules bears a strong resemblance to the *regular worlds* specifications mentioned in Section 3 and implemented in the LF/Twelf metalogical framework [27]; I am interested in exploring the use of transition rules as a logical justification for regular worlds specifications. In this setting, however, there is no reason why we need to limit ourselves to regular worlds-like context generators. Another possibility is the use of context generators that capture the tree-like structure of the ordered context:

$$\begin{array}{ll}
\mathsf{gen/eval}: & \forall \mathsf{E}.\mathsf{gen} \twoheadrightarrow \mathsf{eval}\,\mathsf{E}. \\
\mathsf{gen/retn}: & \forall \mathsf{E}.\mathsf{gen} \twoheadrightarrow \mathsf{retn}\,\mathsf{E}. \\
\mathsf{gen/cont}: & \forall \mathsf{F}.\mathsf{gen} \twoheadrightarrow \mathsf{cont}\,\mathsf{F} \cdot \mathsf{gen}. \\
\mathsf{gen/cont_2}: & \forall \mathsf{F}.\mathsf{gen} \twoheadrightarrow \mathsf{cont_2}\,\mathsf{F} \cdot \mathsf{gen} \cdot \mathsf{gen}.
\end{array}$$

But what if we don't stop there? We could also have the gen atomic proposition carry a type, and then only allow the generation of well-typed tree structures (we return to leaving quantification implicit):

$$\begin{array}{ll}
\mathsf{gen/eval}: & \mathsf{gen}\,\mathsf{T} \cdot !\mathsf{of}\,\mathsf{E}\,\mathsf{T} \twoheadrightarrow \mathsf{eval}\,\mathsf{E}. \\
\mathsf{gen/retn}: & \mathsf{gen}\,\mathsf{T} \cdot !\mathsf{value}\,\mathsf{E} \cdot !\mathsf{of}\,\mathsf{E}\,\mathsf{T} \twoheadrightarrow \mathsf{retn}\,\mathsf{E}. \\
\mathsf{gen/cont}: & \mathsf{gen}\,\mathsf{T} \cdot !\mathsf{off}\,\mathsf{F}\,\mathsf{T}'\,\mathsf{T} \twoheadrightarrow \mathsf{cont}\,\mathsf{F} \cdot \mathsf{gen}\,\mathsf{T}'. \\
\mathsf{gen/cont_2}: & \mathsf{gen}\,\mathsf{T} \cdot !\mathsf{off_2}\,\mathsf{F}\,\mathsf{T_1}\,\mathsf{T_2}\,\mathsf{T} \twoheadrightarrow \mathsf{cont_2}\,\mathsf{F} \cdot \mathsf{gen}\,\mathsf{T_1} \cdot \mathsf{gen}\,\mathsf{T_2}.
\end{array}$$

The static semantics of the language now have been captured entirely within the generation rules, seemingly removing any need for a second invariant that analyzes the state. The statement of the safety theorem would remain as it was in Theorem 1, but its proof would look like this:

$$\begin{tikzcd}
\mathsf{gen}\,\mathsf{T} \arrow[d, squiggly, "\Sigma\mathsf{gen}", "*"'] & \mathsf{gen}\,\mathsf{T} \arrow[d, squiggly, dotted, "\Sigma\mathsf{gen}", "*"'] & & \mathsf{gen}\,\mathsf{T} \arrow[d, squiggly, dotted, "\Sigma\mathsf{gen}", "*"'] & \mathsf{gen}\,\mathsf{T} \arrow[d, squiggly, dotted, "\Sigma\mathsf{gen}", "*"'] & \\
\mathsf{eval}\,\mathsf{E} \arrow[r, squiggly, "\Sigma\mathsf{e}"] & \mathcal{S}_1 & \cdots & \mathcal{S}_n \arrow[r, squiggly, "\Sigma\mathsf{e}"] & \mathcal{S} \arrow[r, squiggly, dotted, "\Sigma\mathsf{e}"] \arrow[r, phantom, "="', shift right=3] & \mathcal{S}' \text{ or } \mathsf{retn}\,\mathsf{V}
\end{tikzcd}$$

The perspective suggested by this picture is quite different from our usual intuitions about static semantics. Usually, static semantics are seen as a way of *analyzing* or *abstractly evaluating* a state in order to generate an approximation (the type) that is invariant under evaluation and sufficient to ensure safety; here, the static semantics is more of a template for generating safe states. We have not explored this style of analysis in depth, but it does seem to address certain significant complications that have been encountered in the process of proving safety for the linear-destination-passing-style specifications [24].

6 Conclusion

In this report, I have presented the outlines of a logical framework based on a state-transition-based view of ordered linear logic and presented preliminary results indicating that the framework allows for both modular specification of programming languages and formal reasoning about their safety properties. The presentation of type safety for a SSOS specified language, the primary novel aspect of this report, is interesting for a number of reasons. In particular, the proofs retain much of the character of existing safety proofs — the general pattern of "Safety = Progress + Preservation" adapts well to the ordered SSOS specifications, and existing typing rules can be used without revision. The notion of what it means for an *expression* to be well typed, in other words, remains the same, and the notion of what it means for a *state* to be well typed is expressible in terms of an ordered logical specification. There is much future work in this area, but as that discussion is extensively explored in my thesis proposal [35], I will not discuss it further here. Instead, I will conclude by briefly discussing two strands of related work: work in similar logical frameworks and work in rewriting logic specifications of programming languages.

From the logical framework perspective, the most closely related work is the LF/LLF/OLF/CLF/HLF family of logical frameworks, though the focus that the framework presented here gives to state transitions exists elsewhere only in the CLF framework, and even there state transitions are second-class citizens. The only published work on proving properties of SSOS specifications in CLF is Schack-Nielsen's proof of the equivalence of a big-step operational semantics and an original-style SSOS specification [33].

The notion that ideas at the intersection of abstract machine specifications and linear logic can capture stateful and concurrent programming language features in a modular way is one with a fairly long history [4, 1, 2, 24, 20, 28]. However, while the power of these frameworks for specification has long been recognized, formal reasoning about properties of these specifications has typically concentrated on properties like approximation [19, 37] and equivalence [33] rather than on familiar properties like progress and preservation that I demonstrate in the transition-based framework. I am aware of two exceptions to this pattern, though in both cases only preservation lemmas were formalized, not progress lemmas. Cervesato's LLF specification of *MLR* can be seen as a transition-based specification that has been flipped around and turned into a backward-chaining specification (or, equivalently, as a small-step specification that has been forced to present itself as a big-step specification), and Reed's HLF is able to mechanically verify Cervesato's preservation theorem [30]. Felty and Momigliano used a similar style of specification in their work on Hybrid to encode abstract machines in ordered logic and reason about subject reduction in either Isabelle/HOL or Coq [8].

While the theoretical basis of this proposal is found in the study of logical frameworks, the most similar project in terms of goals and strategies is the rewriting logic semantics project [17], and in particular the K framework for language specifications [34, 32]. Based on the Maude rewriting framework, these two projects have proven successful in specifying, model-checking, and efficiently executing operational semantics of stateful and concurrent programming languages, including a number of large formalizations of object oriented [13] and functional [16] programming languages.

Many specifications in the K framework bear a strong resemblance to SSOS specifications, and the two approaches to language formalization seem to share a great deal of fundamental structure, even if they differ substantially in emphasis. I am only aware of one discussion of safety via progress and preservation for a K specification [7]; this approach was discussed further in Ellison's masters thesis [6]. The primary limitation of K relative to our approach is a lack of LF-like canonical forms and higher-order abstract syntax, both of which were critical to our specifications and to reasoning about their type safety; in the K specifications, the existing "off the shelf" typing rules for expressions could not be used in the way we could.

On the other hand, the primary limitation of our approach relative to K is the absence of any mechanism for capturing arbitrary sets of propositions at the rule level — in K there is no distinction between terms and propositions, so it is as if we could write this rule in a specification of first-class continuations:

$$\mathsf{CONT} \cdot \mathsf{eval}(\mathsf{callcc}(\lambda \mathsf{x}.\, \mathsf{E}\,\mathsf{x})) \twoheadrightarrow \mathsf{CONT} \cdot \mathsf{eval}(\mathsf{E}(\mathsf{continue}\,\mathsf{CONT}))$$

where CONT captures *all* the propositions $\mathsf{cont}\,\mathsf{F}$ to the left of $\mathsf{eval}(\mathsf{callcc}(\lambda \mathsf{x}.\, \mathsf{E}\,\mathsf{x}))$. This makes K's approach to the modular specification of continuations infeasible in our framework. This does not mean that SSOS specifications cannot express first-class continuations, however! Linear-destination-passing style SSOS specifications can easily and naturally express first-class continuations [24]; previous work has furthermore indicated that there may be a formal connection between ordered SSOS specifications and linear-destination-passing SSOS specifications [38].

References

[1] Iliano Cervesato and Frank Pfenning. A linear logical framework. *Information and Computation*, 179(1):19–75, 2002.

[2] Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-2002-002, Department of Computer Science, Carnegie Mellon University, March 2002. Revised May 2003.

[3] Iliano Cervesato and Andre Scedrov. Relating state-based and process-based concurrency through linear logic. *Information and Computation*, 207:1044–1077, 2009.

[4] Jawahar Lal Chirimar. *Proof Theoretic Approach To Specification Languages*. PhD thesis, University of Pennsylvania, 1995.

[5] Henry DeYoung and Frank Pfenning. Reasoning about the consequences of authorization policies in a linear epistemic logic. In *Workshop on Foundations of Computer Security*, pages 9–23. Informal Proceedings, 2009.

[6] Charles M. Ellison. A rewriting logic approach to defining type systems. Master's thesis, University of Illinois at Urbana-Champaign, 2008.

[7] Charles M. Ellison, Traian Florin Şerbănuţă, and Grigore Roşu. A rewriting logic approach to type inference. Technical Report UIUCDCS-R-2008-2934, University of Illinois at Urbana-Champaign, 2008.

[8] Amy Felty and Alberto Momigliano. Hybrid: A definitional two level approach to reasoning with higher-order abstract syntax. Submitted for publication, available online: http://www.site.uottawa.ca/~afelty/bib.html, September 2008.

[9] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly-threaded parallelism in Manticore. In *Proceedings of the International Conference on Functional Programming*, pages 119–130. ACM, 2008.

[10] Jean-Yves Girard. Locus Solum: From the rules of logic to the logic of rules. *Mathematical Structures in Computer Science*, 11(3):301–506, 2001.

[11] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

[12] Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, 17(4-5):613–673, 2007.

[13] Mark Hills and Grigore Roşu. KOOL: An application of rewriting logic to language prototyping and analysis. In *Rewriting Techniques and Applications*, pages 246–256. Springer LNCS 4533, 2007.

[14] Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.

[15] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996.

[16] Patrick Meredith, Mark Hills, and Grigore Roşu. A K definition of Scheme. Technical Report UIUCDCS-R-2007-2907, University of Illinois at Urbana-Champaign, 2007.

[17] José Meseguer and Grigore Roşu. The rewriting logic semantics project. *Theoretical Computer Science*, 373:213–237, 2007.

[18] Dale Miller. A multiple-conclusion meta-logic. *Theoretical Computer Science*, 165(1):201–232, 1996.

[19] Dale Miller. A proof-theoretic approach to the static analysis of logic programs. In Christoph Benzmueller, Chad E. Brown, Jörg Siekmann, and Richard Statman, editors, *Reasoning in Simple Type Theory: Festschrift in honour of Peter B. Andrews on his 70th birthday. Studies in Logic*, volume 17. College Publications, 2008.

[20] Dale Miller. Formalizing operational semantic specifications in logic. *Electronic Notes in Theoretical Computer Science*, 246:147–165, 2009. Proceedings of WFLP 2008.

[21] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML: Revised*. MIT Press, Cambridge, Massachusetts, 1997.

[22] Peter D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:195–228, 2004.

[23] Peter D. Mosses and Mark J. New. Implicit propagation in structural operational semantics. *Electronic Notes in Theoretical Computer Science*, 229(4):49–66, 2009. Proceedings of SOS 2008.

[24] Frank Pfenning. Substructural operational semantics and linear destination-passing style. In *Programming Languages and Systems*, page 196. Springer LNCS 3302, 2004. Abstract of invited talk.

[25] Frank Pfenning. On linear inference. Unpublished note, available online:
http://www.cs.cmu.edu/~fp/papers/lininf08.pdf, February 2008.

[26] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001. Notes to an invited talk at the *Workshop on Intuitionistic Modal Logics and Applications (IMLA'99)*, Trento, Italy, July 1999.

[27] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206. Springer LNAI 1632, 1999.

[28] Frank Pfenning and Robert J. Simmons. Substructural operational semantics as ordered logic programming. In *Proceedings of the 24th Annual Symposium on Logic in Computer Science (LICS'09)*, pages 101–110, Los Angeles, California, 2009.

[29] Jeff Polakow. *Ordered Linear Logic and Applications*. PhD thesis, Carnegie Mellon University, 2001.

[30] Jason Reed. *A Hybrid Logical Framework*. PhD thesis, Carnegie Mellon University, 2009.

[31] Jason Reed. A judgmental deconstruction of modal logic. Submitted for publication, available online: http://www.cs.cmu.edu/~jcreed/papers/jdml.pdf, May 2009.

[32] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 2010. Submitted for publication, available online: http://fsl.cs.uiuc.edu/index.php/An_Overview_of_the_K_Semantic_Framework.

[33] Anders Schack-Nielsen. Induction on concurrent terms. *Electronic Notes in Theoretical Computer Science*, 196:37–51, 2008. Proceedings of LFMTP 2007.

[34] Traian Florin Şerbănuţă and Grigore Roşu. K-Maude: A rewriting based tool for semantics of programming languages. In *Rewriting Logic and its Applications*. Springer LNCS, 2010. To appear.

[35] Robert J. Simmons. Logical frameworks for specifying and reasoning about stateful and concurrent languages. Thesis proposal at Carnegie Mellon University, July 2010.

[36] Robert J. Simmons and Frank Pfenning. Linear logical algorithms. In *Automata, Languages and Programming*, pages 336–347. Springer LNCS 5126, 2008.

[37] Robert J. Simmons and Frank Pfenning. Linear logical approximations. In *Partial Evaluation and Program Manipulation*, pages 9–20. ACM, 2009.

[38] Robert J. Simmons and Frank Pfenning. Logical approximation for program analysis. Submitted for publication, available online: http://www.cs.cmu.edu/~fp/papers/lapa10.pdf, 2010.

[39] Roberto Virga. *Higher-Order Rewriting with Dependent Types*. PhD thesis, Carnegie Mellon University, 1999.

[40] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-2002-101, Department of Computer Science, Carnegie Mellon University, March 2002. Revised May 2003.

[41] Noam Zeilberger. *The Logical Basis of Evaluation Order and Pattern-Matching*. PhD thesis, Carnegie Mellon University, 2009.