Chapter 4

Substructural logical specifications

In this chapter, we design a logical framework of substructural logical specifications (SLS), a framework heavily inspired by the Concurrent Logical Framework (CLF) [WCPW02]. The framework is justified as a fragment of the logic OL_3 from Chapter 3. There are a number of reasons why we do not just use the already-specified OL_3 outright as a logical framework.

* *Formality.* The specifics of the domain of first-order quantification in OL_3 were omitted in Chapter 3, so in Section 4.1 we give a careful presentation of the term language for SLS, Spine Form LF.
* *Clarity.* The syntax constructions that we presented for OL_3 proof terms had a 1-to-1 correspondence with the sequent calculus rules; the drawback of this presentation is that large proof terms are notationally heavy and difficult to read. The proof terms we present for SLS will leave implicit some of the information present in the diacritical marks of OL_3 proof terms.

  An implementation based on these proof terms would need to consider type reconstruction and/or bidirectional typechecking to recover the omitted information, but we will not consider those issues in this dissertation.
* *Separating concurrent and deductive reasoning.* Comparing CLF to OL_3 leads us to conclude that the single most critical design feature of CLF is its omission of the proposition $\uparrow A^+$. This single omission[1] means that stable sequents in CLF or SLS are effectively restricted to have the succedent $\langle p^- \rangle\ \mathit{true}$ or the succedent $A^+\ \mathit{lax}$.

  Furthermore, any left focus when the succedent is $\langle p^- \rangle\ \mathit{true}$ must conclude with the rule id^-, and any left focus when the succedent is $A^+\ \mathit{lax}$ must conclude with $\bigcirc_L$ – without the elimination of $\uparrow A^+$, left focus in both cases could additionally conclude with the rule $\uparrow_L$. This allows derivations that prove $\langle p^- \rangle\ \mathit{true}$ – the *deductive fragment* of CLF or SLS – to adequately represent deductive systems, conservatively extending deductive logical frameworks like LF and LLF. Derivations that prove $A^+\ \mathit{lax}$, on the other hand, fall into the *concurrent fragment* of CLF and SLS and can encode evolving systems. These fragments have interesting logic programming interpretations, which we explore in Section 4.6.

[1]In our development, the omission of right-permeable propositions p^-_{lax} from OL_3 is equally important, but permeable propositions as we have presented them in Section 2.5.4 were not a relevant consideration in the design of CLF.

- *Partial proofs.* The design of CLF makes it difficult to reason about and manipulate the proof terms corresponding to partial evaluations of evolving systems in the concurrent fragment: the proof terms in CLF correspond to complete proofs and partial evaluations naturally correspond to partial proofs.

  The syntax of SLS is designed to support the explicit representation of partial OL_3 proofs. The omission of the propositions $\mathbf{0}$, $A^+ \oplus B^+$, and the restrictions we place on $t \doteq_\tau s$ are made in the service of presenting a convenient and simple syntax for partial proofs. The three syntactic objects representing partial proofs, *patterns* (Section 4.2.4), *steps*, and *traces* (Section 4.2.6), allow us to treat proof terms for evolving systems as first-class members of SLS.

  The removal of $\mathbf{0}$ and $A^+ \oplus B^+$, and the restrictions we place on $t \doteq_\tau s$, also assist in imposing an equivalence relation, *concurrent equality*, on SLS terms in Section 4.3. Concurrent equality is a coarser equivalence relation than the α-equivalence of OL_3 terms.

- *Removal of* $\top$. The presence of $\top$ causes pervasive problems in the design of substructural logical frameworks. Many of these problems arise at the level of implementation and type reconstruction, which motivated Schack-Nielsen to remove $\top$ from the Celf implementation of CLF [SN11]. Even though those considerations are outside the scope of this dissertation, the presence of $\top$ causes other pervasive difficulties: for instance, the presence of $\top$ complicates the discussion of concurrent equality in CLF. We therefore follow Schack-Nielsen in removing $\top$ from SLS.

In summary, with SLS we *simplify* the presentation of OL_3 for convenience and readability, *restrict* the propositions of OL_3 to separate concurrent and deductive reasoning and to make the syntax for partial proofs feasible, and *extend* OL_3 with a syntax for partial proofs and a coarser equivalence relation.

In Section 4.1 we review the term language for SLS, Spine Form LF. In Section 4.2 we present SLS as a fragment of OL_3, and in Section 4.3 we discuss concurrent equality. In Section 4.4 we adopt the methodology of adequate encoding from LF to SLS, in the process introducing *generative signatures*, which play a starring role in Chapter 9. In Section 4.5 we cover the SLS prototype implementation, and in Section 4.6 we review some intuitions about logic programming in SLS. Finally, in Section 4.7, we discuss some of the decisions reflected in the design of SLS and how some decisions could have been potentially been made differently.

4.1 Spine Form LF as a term language

Other substructural logical frameworks, like Cervesato and Pfenning's LLF [CP02], Polakow's OLF [Pol01], and Watkins et al.'s CLF [WCPW02] are *fully-dependent type theories*: the language of terms (that is, the domain of first-order quantification) is the same as the language of proof terms, the representatives of logical derivations. The logical framework SLS presented in this chapter breaks from this tradition – a choice we discuss further in Section 4.7.3. The domain of first-order quantification, which was left unspecified in Chapter 3, will be presently described as Spine Form LF, a well-understood logical framework derived from the normal forms of the purely persistent type theory LF [HHP93].

All the information in this section is standard and adapted from various sources, especially Harper, Honsell, and Plotkin's original presentation of LF [HHP93], Cervesato and Pfenning's discussion of spine form terms [CP02], Watkins et al.'s presentation of the canonical forms of CLF [WCPW02], Nanevski et al.'s dependent contextual modal type theory [NPP08], Harper and Licata's discussion of Canonical LF [HL07], and Reed's spine form presentation of HLF [Ree09].

It would be entirely consistent for us to appropriate Harper and Licata's Canonical LF presentation instead of presenting Spine Form LF. Nevertheless, a spine-form presentation of canonical LF serves to make our presentation more uniform, as spines are used in the proof term language of SLS. Canonical term languages like Canonical LF correspond to normal natural deduction presentations of logic, whereas spine form term languages correspond to focused sequent calculus presentations like the ones we have considered thus far.

4.1.1 Core syntax

The syntax of Spine Form LF is extended in two places to handle SLS: rules $\mathsf{r} : A^-$ in the signature contain negative SLS types A^- (though it would be possible to separate out the LF portion of signatures from the SLS rules), and several new base kinds are introduced for the sake of SLS – prop, $\mathsf{prop\,ord}$, $\mathsf{prop\,lin}$, and $\mathsf{prop\,pers}$.

Signatures	$\Sigma ::= \cdot \mid \Sigma, \mathsf{c} : \tau \mid \Sigma, \mathsf{a} : \kappa \mid \Sigma, \mathsf{r} : A^-$
Variables	$a, b ::= \ldots$
Variable contexts	$\Psi ::= \cdot \mid \Psi, a{:}\tau$
Kinds	$\kappa ::= \Pi a{:}\tau.\kappa \mid \mathsf{type} \mid \mathsf{prop} \mid \mathsf{prop\,ord} \mid \mathsf{prop\,lin} \mid \mathsf{prop\,pers}$
Types	$\tau ::= \Pi a{:}\tau.\tau' \mid \mathsf{a} \cdot sp$
Heads	$h ::= a \mid \mathsf{c}$
Normal terms	$t, s ::= \lambda a.t \mid h \cdot sp$
Spines	$sp ::= t; sp \mid ()$
Substitutions	$\sigma ::= \cdot \mid t/a, \sigma \mid b/\!/a, \sigma$

Types τ and kinds κ overlap, and will be referred to generically as *classifiers* ν when it is convenient to do so; types and kinds can be seen as refinements of classifiers. Another important refinement are *atomic classifiers* $\mathsf{a} \cdot sp$, which we abbreviate as p.

LF spines sp are just sequences of terms $(t_1; (\ldots; (t_n; ())\ldots))$; we follow common convention and write $h\, t_1 \ldots t_n$ as a convenient shorthand for the atomic term $h \cdot (t_1; \ldots; (t_n; ())\ldots)$; similarly, we will write $\mathsf{a}\, t_1 \ldots t_n$ as a shorthand for atomic classifiers $\mathsf{a} \cdot (t_1; (\ldots; (t_n; ())\ldots))$. This shorthand is given a formal justification in [CP02]; we will use the same shorthand for SLS proof terms in Section 4.2.5.

4.1.2 Simple types and hereditary substitution

In addition to LF types like $\Pi a{:}(\Pi z{:}(\mathsf{a1} \cdot sp_1).\,(\mathsf{a2} \cdot sp_2)).\,\Pi y{:}(\mathsf{a3} \cdot sp_3).\,(\mathsf{a4} \cdot sp_4)$, both Canonical LF and Spine Form LF take *simple types* into consideration. The simple type corresponding

$$\boxed{t \circ sp}$$

$$\begin{aligned}(\lambda a.t') \circ (t; sp) &= [\![t/a]\!]t' \circ sp \\ h \cdot sp \circ () &= h \cdot sp\end{aligned}$$

$$\boxed{[\![t/a]\!]sp}$$

$$\begin{aligned}[\![t/a]\!](t'; sp) &= [\![t/a]\!]t'; [\![t/a]\!]sp \\ [\![t/a]\!]() &= ()\end{aligned}$$

$$\boxed{[\![t/a]\!]t'}$$

$$\begin{aligned}[\![t/a]\!](\lambda y.t') &= \lambda b.\, [\![t/a]\!]t' && (a \neq b) \\ [\![t/a]\!](a \cdot sp) &= t \circ [\![t/a]\!]sp \\ [\![t/a]\!](h \cdot sp) &= h \cdot [\![t/a]\!]sp && (\textit{if } h \neq a)\end{aligned}$$

Figure 4.1: Hereditary substitution on terms, spines, and classifiers

to the type above is $(\mathsf{a1} \to \mathsf{a2}) \to \mathsf{a3} \to \mathsf{a4}$, where $\to$ associates to the right. The simple type associated with the LF type τ is given by the function $|\tau|^- = \tau_s$, where $|\mathsf{a} \cdot sp|^- = \mathsf{a}$ and $|\Pi a{:}\tau.\tau'|^- = |\tau|^- \to |\tau'|^-$.

Variables and constants are treated as having an intrinsic simple type; these intrinsic simple types are sometimes written explicitly as annotations a^{τ_s} or c^{τ_s} (see [Pfe08] for an example), but we will leave them implicit. An atomic term $h\, t_1 \ldots t_n$ must have a simple atomic type a. This means that the head h must have simple type $\tau_{s1} \to \ldots \to \tau_{sn} \to \mathsf{a}$ and each t_i much have simple type τ_{si}. Similarly, a lambda term $\lambda a.t$ must have simple type $\tau_s \to \tau_s'$ where a is a variable with simple type τ_s and t has simple type τ_s'.

Simple types, which are treated in full detail elsewhere [HL07, Ree09], are critical because they allow us to define hereditary substitution and hereditary reduction as total functions in Figure 4.1. Intrinsically-typed Spine Form LF terms correspond to the proof terms for a focused presentation of (non-dependent) minimal logic. Hereditary reduction $t \circ sp$ and hereditary substitution $[\![t/a]\!]t'$, which are both implicitly indexed by the simple type τ_s of t, capture the computational content of structural cut admissibility on these proof terms. Informally, the action of hereditary substitution is to perform a substitution into a term and then continue to reduce any β-redexes that would introduced by a traditional substitution operation. Therefore, $[\![\lambda x.x/f]\!](\mathsf{a}\,(f\,\mathsf{b})\,(f\,\mathsf{c}))$ is not $\mathsf{a}\,((\lambda x.x)\,\mathsf{b})\,((\lambda x.x)\,\mathsf{c})$ – that's not even a syntactically well-formed term according to the grammar for Spine Form LF. Rather, the result of that hereditary substitution is $\mathsf{a\,b\,c}$.

4.1.3 Judgments

Hereditary substitution is necessary to define simultaneous substitution into types and terms in Figure 4.2. We will treat simultaneous substitutions in a mostly informal way, relying on the more careful treatment by Nanevski et al. [NPP08]. A substitution takes every variable in the context and either substitutes a term for it (the form $\sigma, t/a$) or substitutes another variable for it (the form $\sigma, b/\!\!/a$). The latter form is helpful for defining identity substitutions, which we write

$$\boxed{\sigma(sp)}$$

$$\begin{aligned}\sigma(t'; sp) &= \sigma(t'); \sigma(sp)\\ \sigma() &= ()\end{aligned}$$

$$\boxed{\sigma(t')}$$

$$\begin{aligned}\sigma(\lambda a.t') &= \lambda a.\,(\sigma, a/\!\!/a)(t') && (a\#\sigma)\\ \sigma(a \cdot sp) &= t \circ \sigma(sp) && t/a \in \sigma\\ \sigma(a \cdot sp) &= b \cdot \sigma(sp) && b/\!\!/a \in \sigma\\ \sigma(\mathsf{c} \cdot sp) &= \mathsf{c} \cdot \sigma(sp)\end{aligned}$$

$$\boxed{\sigma\nu}$$

$$\begin{aligned}\sigma(\Pi b{:}\nu.\nu') &= \Pi b{:}\sigma\nu.\,(\sigma, b/\!\!/b)\nu' && (a \neq b)\\ \sigma(\mathsf{type}) &= \mathsf{type}\\ \sigma(\mathsf{prop}) &= \mathsf{prop}\\ \sigma(\mathsf{prop\,ord}) &= \mathsf{prop\,ord}\\ \sigma(\mathsf{prop\,lin}) &= \mathsf{prop\,lin}\\ \sigma(\mathsf{prop\,pers}) &= \mathsf{prop\,pers}\\ \sigma(\mathsf{a} \cdot sp) &= \mathsf{a} \cdot \sigma sp\end{aligned}$$

Figure 4.2: Simultaneous substitution on terms, spines, and classifiers

as id or id_Ψ, as well as generic substitutions $[t/a]$ that act like the identity on all variables except for a; the latter notation is used in the definition of LF typing in in Figure 4.3, which is adapted to Spine Form LF from Harper and Licata's Canonical LF presentation [HL07]. The judgments $a\#\sigma$, $a\#\Psi$, $\mathsf{c}\#\Sigma$, $\mathsf{a}\#\Sigma$, and $\mathsf{r}\#\Sigma$ assert that the relevant variable or constant does not already appear in the context Ψ (as a binding $a{:}\tau$), the signature Σ (as a declaration $\mathsf{c} : \tau$, $\mathsf{a} : \nu$, or $\mathsf{r} : A^-$), or the substitution σ (as a binding t/a or $b/\!\!/a$).

All the judgments in Figure 4.3 are indexed by a transitive *subordination relation* $\mathcal{R}$, similar to the one introduced by Virga in [Vir99]. The subordination relation is used to determine if a term or variable of type τ_1 can be a (proper) subterm of a term of type τ_2. Uses of subordination appear in the definition of well-formed equality propositions $t \doteq_\tau s$ in Section 4.2, in the preservation proofs in Section 9.5, and in adequacy arguments (as discussed in [HL07]). We treat $\mathcal{R}$ as a binary relation on type family constants. Let $\mathsf{head}(\tau) = \mathsf{a}$ if $\tau = \Pi a_1{:}\tau_1 \ldots \Pi a_m{:}\tau_m.\,\mathsf{a} \cdot sp$. The signature formation operations depend on three judgments. The index subordination judgment, $\kappa \sqsubset_\mathcal{R} \mathsf{a}$, relates type family constants to types. It is always the case that $\kappa = \Pi a_1{:}\tau_1 \ldots \Pi a_n{:}\tau_n.\mathsf{type}$, and the judgment $\kappa \sqsubset_\mathcal{R} \mathsf{a}$ holds if $(\mathsf{head}(\tau_i), \mathsf{a}) \in \mathcal{R}$ for $1 \leq i \leq n$. The type subordination judgment $\tau \prec_\mathcal{R} \tau'$ holds if $(\mathsf{head}(\tau), \mathsf{head}(\tau')) \in \mathcal{R}$, and the judgment $\tau \preceq_\mathcal{R} \tau'$ is the symmetric extension of this relation.

In Figure 4.3, we define the formation judgments for LF. The first formation judgment is $\vdash_\mathcal{R} \Sigma\,\mathsf{sig}$, which takes a context Σ and determines whether it is well-formed. The premise $\tau \prec_\mathcal{R} \tau$ is used in the definition of term constants to enforce that only self-subordinate types can have constructors. This, conversely, means that types that are not self-subordinate can only

$\boxed{\vdash_{\mathcal{R}} \Sigma\ \mathsf{sig}}$

$$\frac{}{\vdash_{\mathcal{R}} \cdot\ \mathsf{sig}} \qquad \frac{\vdash_{\mathcal{R}} \Sigma\ \mathsf{sig} \quad \cdot \vdash_{\Sigma,\mathcal{R}} \tau\ \mathsf{type} \quad \tau \prec_{\mathcal{R}} \tau \quad \mathsf{c}\#\Sigma}{\vdash_{\mathcal{R}} (\Sigma, \mathsf{c} : \tau)\ \mathsf{sig}}$$

$$\frac{\vdash_{\mathcal{R}} \Sigma\ \mathsf{sig} \quad \cdot \vdash_{\Sigma,\mathcal{R}} \kappa\ \mathsf{kind} \quad \kappa \sqsubset_{\mathcal{R}} \mathsf{a} \quad \mathsf{a}\#\Sigma}{\vdash_{\mathcal{R}} (\Sigma, \mathsf{a} : \kappa)\ \mathsf{sig}} \qquad \frac{\vdash_{\mathcal{R}} \Sigma\ \mathsf{sig} \quad \cdot;\cdot \vdash_{\Sigma,\mathcal{R}} A^-\ \mathsf{prop}^- \quad \mathsf{r}\#\Sigma}{\vdash_{\mathcal{R}} (\Sigma, \mathsf{r} : A^-)\ \mathsf{sig}}$$

$\boxed{\vdash_{\Sigma,\mathcal{R}} \Psi\ \mathsf{ctx}}$ – presumes $\vdash_{\mathcal{R}} \Sigma\ \mathsf{sig}$

$$\frac{}{\vdash_{\Sigma,\mathcal{R}} \cdot\ \mathsf{ctx}} \qquad \frac{\vdash_{\Sigma,\mathcal{R}} \Psi\ \mathsf{ctx} \quad \Psi \vdash_{\Sigma,\mathcal{R}} \tau\ \mathsf{type} \quad a\#\Psi}{\vdash_{\Sigma,\mathcal{R}} (\Psi, a{:}\tau)\ \mathsf{ctx}}$$

$\boxed{\Psi \vdash_{\Sigma,\mathcal{R}} \kappa\ \mathsf{kind}}$ – presumes $\vdash_{\Sigma,\mathcal{R}} \Psi\ \mathsf{ctx}$

$$\frac{\Psi \vdash_{\Sigma,\mathcal{R}} \tau\ \mathsf{type} \quad \Psi, a{:}\tau \vdash_{\Sigma,\mathcal{R}} \kappa\ \mathsf{kind}}{\Psi \vdash_{\Sigma,\mathcal{R}} (\Pi a{:}\tau.\kappa)\ \mathsf{kind}} \qquad \frac{}{\Psi \vdash_{\Sigma,\mathcal{R}} \mathsf{type}\ \mathsf{kind}} \qquad \frac{}{\Psi \vdash_{\Sigma,\mathcal{R}} \mathsf{prop}\ \mathsf{kind}}$$

$$\frac{}{\Psi \vdash_{\Sigma,\mathcal{R}} (\mathsf{prop}\ \mathsf{ord})\ \mathsf{kind}} \qquad \frac{}{\Psi \vdash_{\Sigma,\mathcal{R}} (\mathsf{prop}\ \mathsf{lin})\ \mathsf{kind}} \qquad \frac{}{\Psi \vdash_{\Sigma,\mathcal{R}} (\mathsf{prop}\ \mathsf{pers})\ \mathsf{kind}}$$

$\boxed{\Psi \vdash_{\Sigma,\mathcal{R}} \tau\ \mathsf{type}}$ – presumes $\vdash_{\Sigma,\mathcal{R}} \Psi\ \mathsf{ctx}$

$$\frac{\Psi \vdash_{\Sigma,\mathcal{R}} \tau\ \mathsf{type} \quad \Psi, a{:}\tau \vdash_{\Sigma,\mathcal{R}} \tau'\ \mathsf{type} \quad \tau \preceq_{\mathcal{R}} \tau'}{\Psi \vdash_{\Sigma,\mathcal{R}} (\Pi a{:}\tau.\tau')\ \mathsf{type}} \qquad \frac{a{:}\kappa \in \Sigma \quad \Psi, [\kappa] \vdash_{\Sigma,\mathcal{R}} sp : \mathsf{type}}{\Psi \vdash_{\Sigma,\mathcal{R}} (\mathsf{a} \cdot sp)\ \mathsf{type}}$$

$\boxed{\Psi \vdash_{\Sigma,\mathcal{R}} t : \tau}$ – presumes $\Psi \vdash_{\Sigma,\mathcal{R}} \tau\ \mathsf{type}$

$$\frac{\Psi, a{:}\tau \vdash_{\Sigma,\mathcal{R}} t : \tau'}{\Psi \vdash_{\Sigma,\mathcal{R}} \lambda a.t : \Pi x{:}\tau.\tau'} \qquad \frac{\mathsf{c} : \tau \in \Sigma \quad \Psi, [\tau] \vdash_{\Sigma,\mathcal{R}} sp : \tau' \quad \tau' = p}{\Psi \vdash_{\Sigma,\mathcal{R}} \mathsf{c} \cdot sp : p}$$

$$\frac{a{:}\tau \in \Psi \quad \Psi, [\tau] \vdash_{\Sigma,\mathcal{R}} sp : \tau' \quad \tau' = p}{\Psi \vdash_{\Sigma,\mathcal{R}} a \cdot sp : p}$$

$\boxed{\Psi, [\nu] \vdash_{\Sigma,\mathcal{R}} sp : \nu_0}$ – presumes that either $\Psi \vdash_{\Sigma,\mathcal{R}} \nu\ \mathsf{type}$ or that $\Psi \vdash_{\Sigma,\mathcal{R}} \nu\ \mathsf{kind}$

$$\frac{}{\Psi, [\nu] \vdash_{\Sigma,\mathcal{R}} () : \nu} \qquad \frac{\Psi \vdash_{\Sigma,\mathcal{R}} t : \tau \quad [t/a]\nu = \nu' \quad \Psi, [\nu'] \vdash_{\Sigma,\mathcal{R}} sp : \nu_0}{\Psi, [\Pi a{:}\tau.\nu] \vdash_{\Sigma,\mathcal{R}} t; sp : \nu_0}$$

$\boxed{\Psi \vdash \sigma : \Psi'}$ – presumes $\vdash_{\Sigma,\mathcal{R}} \Psi\ \mathsf{ctx}$ and $\vdash_{\Sigma,\mathcal{R}} \Psi'\ \mathsf{ctx}$

$$\frac{}{\Psi \vdash_{\Sigma,\mathcal{R}} \cdot : \cdot} \qquad \frac{\Psi \vdash_{\Sigma,\mathcal{R}} \sigma : \Psi' \quad \Psi \vdash_{\Sigma,\mathcal{R}} t : \sigma\tau}{\Psi \vdash_{\Sigma,\mathcal{R}} (\sigma, t/a) : \Psi', a{:}\tau} \qquad \frac{\Psi \vdash_{\Sigma,\mathcal{R}} \sigma : \Psi' \quad b{:}\sigma\tau \in \Psi}{\Psi \vdash_{\Sigma,\mathcal{R}} (\sigma, b/\!/a) : \Psi', a{:}\tau}$$

Figure 4.3: LF formation judgments ($\tau' = p$ refers to α-equivalence)

be inhabited by variables a, which is important for one of the two types of equality $t \doteq_\tau s$ that SLS supports. The judgments $\vdash_{\Sigma,\mathcal{R}} \Psi\,\mathsf{ctx}$, $\Psi \vdash_{\Sigma,\mathcal{R}} \kappa\,\mathsf{kind}$, and $\Psi \vdash_{\Sigma,\mathcal{R}} \tau\,\mathsf{type}$ take contexts Ψ, kinds κ, and types τ and ensure that they are well-formed in the current signature or (if applicable) context. The judgment $\Psi \vdash_{\Sigma,\mathcal{R}} t{:}\tau$ takes a term and a type and typechecks the term against the type, and the judgment $\Psi \vdash_{\Sigma,\mathcal{R}} \sigma : \Psi'$ checks that a substitution σ can transport objects (terms, types, etc.) defined in the context Ψ' to objects defined in Ψ.

The judgment $\Psi, [\nu] \vdash_{\Sigma,\mathcal{R}} sp : \nu_0$ is read a bit differently than these other judgments. The notation, first of all, is meant to evoke the (exactly analogous) left-focus judgments from Chapters 2 and 3. In most other sources (for example, in [CP02]) this judgment is instead written as $\Psi \vdash_{\Sigma,\mathcal{R}} sp : \nu > \nu_0$. In either case, we read this judgment as checking a spine sp against a classifier ν (actually either a type τ or a kind κ) and *synthesizing* a return classifier ν_0. In other words, ν_0 is an output of the judgment $\Psi, [\nu] \vdash_{\Sigma,\mathcal{R}} sp : \nu_0$, and given that this judgment presumes that either $\Psi \vdash_{\Sigma,\mathcal{R}} \nu\,\mathsf{type}$ or $\Psi \vdash_{\Sigma,\mathcal{R}} \nu\,\mathsf{kind}$, it *ensures* that either $\Psi \vdash_{\Sigma,\mathcal{R}} \nu_0\,\mathsf{type}$ or $\Psi \vdash_{\Sigma,\mathcal{R}} \nu_0\,\mathsf{kind}$, where the classifiers of ν and ν_0 (type or kind) always match. It is because ν_0 is an output that we add an explicit premise to check that $\tau' = p$ in the typechecking rule for $\mathsf{c} \cdot sp$; this equality refers to the α-equality of Spine Form LF terms.

There are a number of well-formedness theorems that we need to consider, such as the fact that substitutions compose in a well-behaved way and that hereditary substitution is always well-typed. However, as these theorems are adequately covered in the aforementioned literature on LF, we will proceed with using LF as a term language and will treat term-level operations like substitution somewhat informally.

We will include annotations for the signature Σ and the subordination relation $\mathcal{R}$ in the definitions of this section and the next one. In the following sections and chapters, however, we will often leave the signature Σ implicit when it is unambiguous or unimportant. We will almost always leave the subordination relation implicit; we can assume where applicable that we are working with the *strongest* (that is, the smallest) subordination relation for the given signature [HL07].

4.1.4 Adequacy

Adequacy was the name given by Harper, Honsell, and Plotkin to the methodology of connecting inductive definitions to the canonical forms of a particular type family in LF. Consider, as a standard example, the untyped lambda calculus, which is generally specified by a BNF grammar such as the following:

$$e ::= x \mid \lambda x.e \mid e_1\, e_2$$

We can adequately encode this language of terms into LF (with a subordination relation $\mathcal{R}$ such that $(\mathsf{exp}, \mathsf{exp}) \in \mathcal{R}$) by giving the following signature:

$$\begin{aligned} \Sigma = {} & \cdot, \\ & \mathsf{exp} : \mathsf{type}, \\ & \mathsf{app} : \Pi a{:}\mathsf{exp}.\, \Pi b{:}\mathsf{exp}.\, \mathsf{exp}, \\ & \mathsf{lam} : \Pi a{:}(\Pi b{:}\mathsf{exp}.\, \mathsf{exp}).\, \mathsf{exp} \end{aligned}$$

Note that the variables a and b are bound by Π-quantifiers in the declaration of app and lam but never used. The usual convention is to abbreviate $\Pi a{:}\tau.\tau'$ as $\tau \rightarrow \tau'$ when a is not free in τ', which would give app type $\mathsf{exp} \rightarrow \mathsf{exp} \rightarrow \mathsf{exp}$ and lam type $(\mathsf{exp} \rightarrow \mathsf{exp}) \rightarrow \mathsf{exp}$.

Theorem 4.1 (Adequacy for terms)**.** *Up to standard α-equivalence, there is a bijection between expressions e (with free variables in the set $\{x_1, \ldots, x_n\}$) and Spine Form LF terms t such that* $x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp} \vdash t : \mathsf{exp}$.

Proof. By induction on the structure of the inductive definition of e in the forward direction and by induction on the structure of terms t with type exp in the reverse direction. □

We express the constructive content of this theorem as an invertible function $\ulcorner e \urcorner = t$ from object language terms e to representations LF terms t of type exp:

* $\ulcorner x \urcorner = x$,
* $\ulcorner e_1\, e_2 \urcorner = \mathsf{app}\,\ulcorner e_1 \urcorner \ulcorner e_2 \urcorner$, and
* $\ulcorner \lambda x.e \urcorner = \mathsf{lam}\,\lambda x.\ulcorner e \urcorner$.

If we had also defined substitution $[e/x]e'$ on terms, it would be necessary to show that the bijection is compositional: that is, that $[\ulcorner e \urcorner/x]\ulcorner e' \urcorner = \ulcorner [e/x]e' \urcorner$. Note that adequacy critically depends on the context having the form $x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp}$. If we had a context with a variable $y{:}(\mathsf{exp} \rightarrow \mathsf{exp})$, then we could form an LF term $y\,(\mathsf{lam}\,\lambda x.x)$ with type exp that does *not* adequately encode any term e in the untyped lambda calculus.

One of the reasons subordination is important in practice is that it allows us to consider the adequate encoding of expressions in contexts Ψ that have other variables $x{:}\tau$ as long as $(\mathsf{head}(\tau), \mathsf{exp}) \notin \mathcal{R}$. If $\Psi, x{:}\tau \vdash_{\Sigma,\mathcal{R}} t : \mathsf{exp}$ and $\tau \not\preceq_{\mathcal{R}} \mathsf{exp}$, then x cannot be free in t, so $\Psi \vdash_{\Sigma,\mathcal{R}} t : \mathsf{exp}$ holds as well. By iterating this procedure, it may be possible to strengthen a context Ψ into one of the form $x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp}$, in which case we can conclude that $t = \ulcorner e \urcorner$ for some untyped lambda calculus term e.

4.2 The logical framework SLS

In this section, we will describe the restricted set of polarized OL_3 propositions and focused OL_3 proof terms that make up the logical framework SLS. For the remainder of the dissertation, we will work exclusively with the following positive and negative SLS propositions, which are a syntactic refinement of the positive and negative propositions of polarized OL_3:

$$A^+, B^+, C^+ ::= p^+ \mid p^+_{eph} \mid p^+_{pers} \mid {\downarrow}A^- \mid {¡}A^- \mid {!}A^- \mid \mathbf{1} \mid A^+ \bullet B^+ \mid \exists a{:}\tau.A^+ \mid t \doteq_\tau s$$
$$A^-, B^-, C^- ::= p^- \mid \bigcirc A^+ \mid A^+ \rightarrowtail B^- \mid A^+ \twoheadrightarrow B^- \mid A^- \mathbin{\&} B^- \mid \forall a{:}\tau.A^-$$

We now have to deal with a point of notational dissonance: all existing work on CLF, all existing implementations of CLF, and the prototype implementation of SLS (Section 4.5) use the notation $\{A^+\}$ for the connective internalizing the judgment A^+ *lax*, which we have written as $\bigcirc A^+$, following Fairtlough and Mendler [FM97]. The traditional notation overloads curly braces, which

$\boxed{\Psi \vdash_{\Sigma,\mathcal{R}} \mathcal{C}\,\mathsf{satisfiable}}$ – presumes $\vdash_{\Sigma,\mathcal{R}} \Psi\,\mathsf{ctx}$, that the terms in

$$\frac{}{\Psi \vdash_{\Sigma,\mathcal{R}} \cdot\,\mathsf{satisfiable}} \qquad \frac{\Psi \vdash_{\Sigma,\mathcal{R}} t{:}\tau \quad \Psi \vdash_{\Sigma,\mathcal{R}} \mathcal{C}\,\mathsf{satisfiable}}{\Psi \vdash_{\Sigma,\mathcal{R}} (\mathcal{C}, t \doteq_\tau t)\,\mathsf{satisfiable}}$$

$$\frac{a{:}p \in \Psi \quad \Psi \vdash_{\Sigma,\mathcal{R}} t : p \quad \Psi' \vdash_{\Sigma,\mathcal{R}} [t/a] : \Psi \quad \Psi' \vdash_{\Sigma,\mathcal{R}} [t/a]\mathcal{C}\,\mathsf{satisfiable}}{\Psi \vdash_{\Sigma,\mathcal{R}} (\mathcal{C}, a \doteq_p t)\,\mathsf{satisfiable}}$$

Figure 4.4: Equality constraints (used to support notational definitions)

we also use for the context-framing notation $\Theta\{\Delta\}$ introduced in Section 3.2. We will treat $\bigcirc A^+$ and $\{A^+\}$ as synonyms in SLS, preferring the former in this chapter and the latter afterwards.

Positive ordered atomic propositions p^+ are atomic classifiers $\mathsf{a}\, t_1 \ldots t_n$ with kind $\mathsf{prop\,ord}$, positive linear and persistent atomic propositions p^+_{eph} and p^+_{pers} are (respectively) atomic classifiers with kind $\mathsf{prop\,lin}$ and $\mathsf{prop\,pers}$, and negative ordered atomic propositions p^- are atomic classifiers with kind prop. From this point on, we will unambiguously refer to atomic propositions p^- as negative atomic propositions, omitting "ordered." Similarly, we will refer to atomic propositions p^+, p^+_{eph}, and p^+_{pers} collectively as positive atomic propositions but individually as ordered, linear, and persistent propositions, respectively, omitting "positive." ("Mobile" and "ephemeral" will continue to be used as synonyms for "linear.")

4.2.1 Propositions

The formation judgments for SLS types are given in Figure 4.5. As discussed in the introduction to this chapter, the removal of $\uparrow A^+$ and p^-_{lax} is fundamental to the separation of the deductive and concurrent fragments of SLS; most of the other restrictions made to the language are for the purpose of giving partial proofs a list-like structure. In particular, all positive propositions whose left rules have more or less than one premise are restricted. The propositions $\mathbf{0}$ and $A^+ \oplus B^+$ are excluded from SLS to this end, and we must place rather draconian restrictions on the use of equality in order to ensure that $\doteq_L$ can always be treated as having exactly one premise.

The formation rules for propositions are given in Figure 4.5. Much of the complexity of this presentation, such as the existence of an additional constraint context $\mathcal{C}$, described in Figure 4.4, is aimed at allowing the inclusion of equality in SLS in a sufficiently restricted form. The intent of these restrictions is to ensure that, whenever we decompose a positive proposition $s \doteq t$ on the left, we have that s is some variable a in the context and that a is not free in t. When this is the case, $[t/a]$ is always a most general unifier of $s = a$ and t, which in turn means that the left rule for equality in OL$_3$

$$\frac{\forall(\Psi' \vdash \sigma : \Psi).\quad \sigma t = \sigma s \quad \longrightarrow \quad \Psi'; \sigma\Theta\{\cdot\} \vdash \sigma U}{\Psi; \Theta\{\!\{t \doteq_\tau s\}\!\} \vdash U} \doteq_L$$

is equivalent to a much simpler rule:

$$\frac{\Psi, [t/a]\Psi'; [t/a]\Theta\{\cdot\} \vdash [t/a]U}{\Psi, a{:}\tau, \Psi'; \Theta\{\!\{a \doteq_\tau t\}\!\} \vdash U} \doteq_{yes}$$

$\boxed{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} A^+ \; \mathsf{prop}^+}$ – presumes $\vdash_{\Sigma,\mathcal{R}} \Psi$ ctx and that $\Psi \vdash \mathcal{C}$ satisfiable.

$$\frac{\mathsf{a}{:}\kappa \in \Sigma \quad \Psi, [\kappa] \vdash_{\Sigma,\mathcal{R}} sp : \mathsf{prop\ ord}}{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} \mathsf{a} \cdot sp \; \mathsf{prop}^+} \qquad \frac{\mathsf{a}{:}\kappa \in \Sigma \quad \Psi, [\kappa] \vdash_{\Sigma,\mathcal{R}} sp : \mathsf{prop\ lin}}{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} \mathsf{a} \cdot sp \; \mathsf{prop}^+}$$

$$\frac{\mathsf{a}{:}\kappa \in \Sigma \quad \Psi, [\kappa] \vdash_{\Sigma,\mathcal{R}} sp : \mathsf{prop\ pers}}{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} \mathsf{a} \cdot sp \; \mathsf{prop}^+}$$

$$\frac{\Psi; \cdot \vdash_{\Sigma,\mathcal{R}} A^- \; \mathsf{prop}^-}{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} {\downarrow}A^- \; \mathsf{prop}^+} \qquad \frac{\Psi; \cdot \vdash_{\Sigma,\mathcal{R}} A^- \; \mathsf{prop}^-}{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} \text{¡}A^- \; \mathsf{prop}^+} \qquad \frac{\Psi; \cdot \vdash_{\Sigma,\mathcal{R}} A^- \; \mathsf{prop}^-}{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} !A^- \; \mathsf{prop}^+} \qquad \frac{}{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} \mathbf{1} \; \mathsf{prop}^+}$$

$$\frac{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} A^+ \; \mathsf{prop}^+ \quad \Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} B^+ \; \mathsf{prop}^+}{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} A^+ \bullet B^+ \; \mathsf{prop}^+}$$

$$\frac{\Psi \vdash_{\Sigma,\mathcal{R}} \tau \; \mathsf{type} \quad (t = a \text{ or } \Psi \vdash_{\Sigma,\mathcal{R}} t : \tau) \quad \Psi, a{:}\tau; \mathcal{C}, a \doteq_\tau t \vdash_{\Sigma,\mathcal{R}} A^+ \; \mathsf{prop}^+}{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} \exists a{:}\tau. A^+ \; \mathsf{prop}^+}$$

$$\frac{\Psi \vdash_{\Sigma,\mathcal{R}} p \; \mathsf{type} \quad a{:}p \in \Psi \quad b{:}p \in \Psi \quad p \not\approx_{\mathcal{R}} p}{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} a \doteq_p b \; \mathsf{prop}^+} \qquad \frac{t \doteq_p s \in \mathcal{C} \quad \Psi \vdash_{\Sigma,\mathcal{R}} s : p}{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} t \doteq_p s \; \mathsf{prop}^+}$$

$\boxed{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} A^- \; \mathsf{prop}^-}$ – presumes $\vdash_{\Sigma,\mathcal{R}} \Psi$ ctx and that $\Psi \vdash \mathcal{C}$ satisfiable.

$$\frac{\mathsf{a}{:}\kappa \in \Sigma \quad \Psi, [\kappa] \vdash_{\Sigma,\mathcal{R}} sp : \mathsf{prop}}{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} \mathsf{a} \cdot sp \; \mathsf{prop}^-} \qquad \frac{\Psi; \cdot \vdash_{\Sigma,\mathcal{R}} A^+ : \mathsf{prop}^+}{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} \bigcirc A^+ \; \mathsf{prop}^-}$$

$$\frac{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} A^+ \; \mathsf{prop}^+ \quad \Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} B^- \; \mathsf{prop}^-}{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} A^+ \rightarrowtail B^- \; \mathsf{prop}^-} \qquad \frac{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} A^+ \; \mathsf{prop}^+ \quad \Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} B^- \; \mathsf{prop}^-}{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} A^+ \twoheadrightarrow B^- \; \mathsf{prop}^-}$$

$$\frac{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} A^- \; \mathsf{prop}^- \quad \Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} B^- \; \mathsf{prop}^-}{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} A^- \,\&\, B^- \; \mathsf{prop}^-}$$

$$\frac{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} \tau \; \mathsf{type} \quad (t = a \text{ or } \Psi \vdash_{\Sigma,\mathcal{R}} t : \tau) \quad \Psi, a{:}\tau; \mathcal{C}, a \doteq_\tau t \vdash_{\Sigma,\mathcal{R}} A^- \; \mathsf{prop}^-}{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} \forall a{:}\tau. A^- \; \mathsf{prop}^-}$$

Figure 4.5: SLS propositions

Usually, when we require the "existence of most general unifiers," that signals that a most general unifier must exist if any unifier exists. The condition we are requiring is much stronger: for the unification problems we will encounter due to the $\doteq_L$ rule, a most general unifier *must* exist. Allowing unification problems that could fail would require us to consider positive inversion rules with zero premises, and the proposition **0** was excluded from SLS precisely to prevent us from needing to deal with positive inversion rules with zero premises.[2]

There are two distinct conditions under which we can be sure that unification problems always have a most general solution – when equality is performed over *pure variables* and when equality is used as a *notational definition* [PS99]. Equality of pure variable types is used in the destination-adding transformation in Chapter 7, and notational definitions are used extensively in Chapter 8.

Pure variables Equality at an atomic type p that is *not subordinate to itself* ($p \not\prec_{\mathcal{R}} p$) is always allowed. This is reflected in the first formation rule for $t \doteq s$ in Figure 4.5.

Types that are not self-subordinate can only be inhabited by variables: that is, if $p \not\prec_{\mathcal{R}} p$ and $\Psi \vdash_{\Sigma,\mathcal{R}} t : p$, then $t = a$ where $a{:}p \in \Psi$. For any unification problem $a \doteq b$, both $[a/b]$ and $[b/a]$ are most general unifiers.

Notational definitions Using equality as a notational definition allows us manipulate propositions in ways that have no effect on the structure of synthetic inference rules. When we check a universal quantifier $\forall a{:}p.A^-$ or existential quantifier $\exists a{:}p.A^+$, we are allowed to introduce one term t that will be forced, by the use of equality, to be unified with this newly-introduced variable. By adding the $a \doteq_p t$ to the set of constraints $\mathcal{C}$, we allow ourselves to mention $a \doteq_p t$ later on in the proposition by using the second formation rule for equality in Figure 4.5.

The notational definition $a \doteq_p t$ must be reachable from its associated quantifier without crossing a shift or an exponential – in Andreoli's terms, it must be in the same *monopole* (Section 2.4). This condition, which it might be possible to relax at the cost of further complexity elsewhere in the presentation, is enforced by the formation rules for shifts and exponentials, which clear the context $\mathcal{C}$ in their premise. The proposition $\forall a. {\downarrow}(\mathsf{p}\, a) \rightarrowtail a \doteq t \rightarrowtail \mathsf{p}\, t$ satisfies this condition but $\forall a. \bigcirc(a \doteq t)$ does not ($\bigcirc$ breaks focus), and the proposition $\bigcirc(\exists a.a \doteq_p t)$ satisfies this condition but $\bigcirc(\exists a.{\uparrow}(a \doteq t \rightarrowtail \mathsf{p}\, a))$ does not ($\uparrow$ breaks focus). The rule $\bigcirc(\exists a{:}p.\, a \doteq_p \mathsf{s}\, a)$ doesn't pass muster because the term t must be well-formed in a context that does not include a – this is related to the *occurs check* in unification. The rule $\bigcirc(\exists a.\, a \doteq t \bullet a \doteq s)$ is not well-formed if t and s are syntactically distinct. Each variable can only be notationally defined to be one term; otherwise we could encode an arbitrary unification problem $t \doteq s$.

4.2.2 Substructural contexts

Figure 4.6 describes the well-formed substructural contexts in SLS. The judgment $\Psi \vdash_{\Sigma,\mathcal{R}} T\ \mathsf{left}$ is used to check stable bindings $x{:}A^-\ \mathit{lvl}$ and $z{:}\langle p^+_{\mathit{lvl}}\rangle\ \mathit{lvl}$ that can appear as a part of stable, inverting, or left-focused sequents; the judgment $\Psi \vdash_{\Sigma,\mathcal{R}} \Delta\ \mathsf{stable}$ just maps this judgment over

[2]The other side of this observation is that, if we allow the proposition **0** and adapt the logical framework accordingly, it might be possible to relax the restrictions we have placed on equality.

$\boxed{\Psi \vdash_{\Sigma,\mathcal{R}} T \,\mathsf{left}}$ – presumes $\vdash_{\Sigma,\mathcal{R}} \Psi \,\mathsf{ctx}$

$$\frac{\Psi; \cdot \vdash_{\Sigma,\mathcal{R}} A^- \,\mathsf{prop}^-}{\Psi \vdash_{\Sigma,\mathcal{R}} (A^- \, \mathit{lvl}) \,\mathsf{left}} \qquad \frac{\mathsf{a} : \kappa \in \Sigma \quad \Psi, [\kappa] \vdash_{\Sigma,\mathcal{R}} \mathit{sp} : \mathsf{prop\,ord}}{\Psi \vdash_{\Sigma,\mathcal{R}} (\langle \mathsf{a} \cdot \mathit{sp} \rangle \, \mathit{ord}) \,\mathsf{left}}$$

$$\frac{\mathsf{a} : \kappa \in \Sigma \quad \Psi, [\kappa] \vdash_{\Sigma,\mathcal{R}} \mathit{sp} : \mathsf{prop\,lin}}{\Psi \vdash_{\Sigma,\mathcal{R}} (\langle \mathsf{a} \cdot \mathit{sp} \rangle \, \mathit{eph}) \,\mathsf{left}} \qquad \frac{\mathsf{a} : \kappa \in \Sigma \quad \Psi, [\kappa] \vdash_{\Sigma,\mathcal{R}} \mathit{sp} : \mathsf{prop\,pers}}{\Psi \vdash_{\Sigma,\mathcal{R}} (\langle \mathsf{a} \cdot \mathit{sp} \rangle \, \mathit{pers}) \,\mathsf{left}}$$

$\boxed{\Psi \vdash_{\Sigma,\mathcal{R}} \Delta \,\mathsf{stable}}$ – presumes $\vdash_{\Sigma,\mathcal{R}} \Psi \,\mathsf{ctx}$

$$\frac{}{\Psi \vdash_{\Sigma,\mathcal{R}} \cdot \,\mathsf{stable}} \qquad \frac{\Psi \vdash_{\Sigma,\mathcal{R}} \Delta \,\mathsf{stable} \quad \Psi \vdash_{\Sigma,\mathcal{R}} T \,\mathsf{left}}{\Psi \vdash_{\Sigma,\mathcal{R}} (\Delta, x{:}T) \,\mathsf{stable}}$$

$\boxed{\Psi \vdash_{\Sigma,\mathcal{R}} \Delta \,\mathsf{inv}}$ – presumes $\vdash_{\Sigma,\mathcal{R}} \Psi \,\mathsf{ctx}$

$$\frac{}{\Psi \vdash_{\Sigma,\mathcal{R}} \cdot \,\mathsf{inv}} \qquad \frac{\Psi \vdash_{\Sigma,\mathcal{R}} \Delta \,\mathsf{inv} \quad \Psi \vdash_{\Sigma,\mathcal{R}} T \,\mathsf{left}}{\Psi \vdash_{\Sigma,\mathcal{R}} (\Delta, x{:}T) \,\mathsf{inv}} \qquad \frac{\Psi \vdash_{\Sigma,\mathcal{R}} \Delta \,\mathsf{inv} \quad \Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} A^+ \,\mathsf{prop}^+}{\Psi \vdash_{\Sigma,\mathcal{R}} (\Delta, x{:}A^+ \, \mathit{ord}) \,\mathsf{inv}}$$

$\boxed{\Psi \vdash_{\Sigma,\mathcal{R}} \Delta \,\mathsf{infoc}}$ – presumes $\vdash_{\Sigma,\mathcal{R}} \Psi \,\mathsf{ctx}$

$$\frac{\Psi \vdash_{\Sigma,\mathcal{R}} \Delta \,\mathsf{infoc} \quad \Psi \vdash_{\Sigma,\mathcal{R}} T \,\mathsf{left}}{\Psi \vdash_{\Sigma,\mathcal{R}} (\Delta, x{:}T) \,\mathsf{infoc}} \qquad \frac{\Psi \vdash_{\Sigma,\mathcal{R}} \Delta \,\mathsf{stable} \quad \Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} A^- \,\mathsf{prop}^-}{\Psi \vdash_{\Sigma,\mathcal{R}} (\Delta, x{:}[A^-] \, \mathit{ord}) \,\mathsf{infoc}}$$

Figure 4.6: SLS contexts

the context. The judgment $\Psi \vdash_{\Sigma,\mathcal{R}} \Delta \,\mathsf{inv}$ describes contexts during the inversion phase, which can also contain inverting positive propositions A^+. The judgment $\Psi \vdash_{\Sigma,\mathcal{R}} \Delta \,\mathsf{infoc}$ describes a context that is stable aside from the one negative proposition in focus.

The rules for inverting and focusing on propositions in Figure 4.6 use non-empty constraint context. This is necessary because the property of being a well-formed proposition is not stable under arbitrary substitutions. Even though $\forall a{:}p.\,(a \doteq_p \mathsf{c}) \rightarrowtail A^-$ is a well-formed negative proposition according to Figure 4.5, $(a \doteq_p \mathsf{c}) \rightarrowtail A^-$ is only a well-formed proposition if we add $a \doteq_p \mathsf{c}$ to the set of constraints, and $(\mathsf{c} \doteq_p \mathsf{c}) \rightarrowtail [\mathsf{c}/a]A^-$ is only a well-formed proposition if we add $\mathsf{c} \doteq_p \mathsf{c}$ to the set of constraints.

The restrictions we make to contexts justify our continued practice of omitting the *ord* annotation when talking about inverting positive propositions A^+ or focused negative propositions $[A^-]$ in the context, since these context constituents only appear in conjunction with the *ord* judgment.

This discussion of well-formed propositions and contexts takes care of any issues dealing with variables that were swept under the rug in Chapter 3. We could stop here and use the refinement of OL$_3$ proof terms that corresponds to our refinement of propositions as the language of SLS proof terms. This is not desirable for two main reasons. First, the proof terms of focused OL_3 make it inconvenient (though not impossible) to talk about concurrent equality (Section 4.3). Second, one of our primary uses of SLS in this dissertation will be to talk about *traces*, which correspond roughly to partial proofs

$$\begin{array}{c}\Psi'; \Delta' \vdash A^+ \; lax \\ \vdots \\ \Psi; \Delta \vdash A^+ \; lax\end{array}$$

in OL_3, where both the top and bottom sequents are stable and where A^+ is some unspecified, parametric positive proposition. Using OL_3-derived proof terms makes it difficult to talk about about and manipulate proofs of this form.

In the remainder of this section, we will present a proof term assignment for SLS that facilitates discussing concurrent equality and partial proofs. SLS proof terms are in bijective correspondence with a refinement of OL_3 proof terms when we consider complete (deductive) proofs, but the introduction of patterns and traces reconfigures the structure of derivations and proof terms.

4.2.3 Process states

A *process state* is a disembodied left-hand side of a sequent that we use to describe the intermediate states of concurrent systems. Traces, introduced in Section 4.2.6, are intended to capture the structure of partial proofs:

$$\begin{array}{c}\Psi'; \Delta' \vdash A^+ \; lax \\ \vdots \\ \Psi; \Delta \vdash A^+ \; lax\end{array}$$

The type of a trace will be presented as a relation between two process states. As a first cut, we can represent the initial state as $(\Psi; \Delta)$ and the final state as $(\Psi'; \Delta')$, and we can omit Ψ and just write Δ when that is sufficiently clear.

Representing a process state as merely an LF context Ψ and a substructural context Δ is insufficient because of the way equality – pure variable equality in particular – can unify distinct variables. Consider the following partial proof:

$$\begin{array}{c} b{:}p; \;\; z{:}\langle \mathsf{foo}\, b\, b\rangle \; eph \vdash (\mathsf{foo}\, b\, b) \; lax \\ \vdots \\ a{:}p, b{:}p; \;\; x{:}\bigcirc(a \doteq_\tau b) \; eph, \;\; z{:}\langle \mathsf{foo}\, a\, a\rangle \; eph \vdash (\mathsf{foo}\, a\, b) \; lax \end{array}$$

This partial proof can be constructed in one focusing stage by a left focus on x. It is insufficient to capture the first process state as $(a{:}p, b{:}p; \;\; x{:}\bigcirc(a \doteq_\tau b), \;\; z{:}\langle \mathsf{foo}\, a\, a\rangle \; eph)$ and the second process state as $(b{:}p; \;\; z{:}\langle \mathsf{foo}\, b\, b\rangle \; eph)$, as this would fail to capture that the succedent $(\mathsf{foo}\, b\, b) \; lax$ is a substitution instance of the succedent $(\mathsf{foo}\, a\, b) \; lax$. In general, if the derivation above proved some arbitrary succedent $A^+ \; lax$ instead of the specific succedent $(\mathsf{foo}\, a\, b) \; lax$, then the missing subproof would have the succedent $[b/a]A^+ \; lax$.

A process state is therefore written as $(\Psi; \Delta)_\sigma$ and is well-formed under the signature Σ and the subordination relation $\mathcal{R}$ if $\Psi \vdash_{\Sigma,\mathcal{R}} \Delta \; \mathsf{inv}$ (which presumes that $\vdash_{\Sigma,\mathcal{R}} \Psi \; \mathsf{ctx}$, as defined in

Figure 4.3) and if $\Psi \vdash \sigma : \Psi_0$, where Ψ_0 is some other context that represents the starting point, the context in which the disconnected succedent $A^+\ lax$ is well-formed.

$$\frac{\Psi \vdash_{\Sigma,\mathcal{R}} \Delta\ \mathsf{inv} \quad \vdash_{\Sigma,\mathcal{R}} \Psi_0 : \mathsf{ctx} \quad \Psi \vdash \sigma : \Psi_0}{\vdash_{\Sigma,\mathcal{R}} (\Psi; \Delta)_\sigma\ \mathsf{state}}$$

Taking $\Psi_0 = a{:}p, b{:}p$, the partial proof above can thus be represented as a step (Section 4.2.6) between these two process states:

$$(a{:}p, b{:}p;\ \ x{:}\bigcirc(a \doteq_\tau b),\ \ z{:}\langle \mathsf{foo}\, a\, a\rangle\ eph)_{(a/a,\, b/b)} \leadsto_{\Sigma,\mathcal{R}} (b{:}p;\ \ z{:}\langle \mathsf{foo}\, b\, b\rangle\ eph)_{(b/a,\, b/b)}$$

Substitutions are just one of several ways that we could cope with free variables in succedents; another option, discussed in Section 4.3, is to track the set of constraints $a = b$ that have been encountered by unification. When we consider traces in isolation, we will generally let $\Psi_0 = \cdot$ and $\sigma = \cdot$, which corresponds to the case where the parametric conclusion A^+ is a closed proposition. When the substitution is not mentioned, it can therefore be presumed to be empty. Additionally, when the LF context Ψ is empty or clear from the context, we will omit it as well. One further simplification is that we will occasionally omit the judgment lvl associated with a suspended positive atomic proposition $\langle p^+_{lvl}\rangle\ lvl$, but only when it is unambiguous from the current signature that p^+_{lvl} is an ordered, linear, or persistent positive atomic proposition. In the examples above, we tacitly assumed that foo was given kind $p \to p \to \mathsf{prop\ lin}$ in the signature Σ when we tagged the suspended atomic propositions with the judgment eph. If it had been clear that foo was linear, then this judgment could have been omitted.

4.2.4 Patterns

A *pattern* is a syntactic entity that captures the list-like structure of left inversion on positive propositions. The OL$_3$ proof term for the proposition $(\exists a.\, \mathsf{p}\, a \bullet {¡}A^- \bullet {\downarrow}B^-) \rightarrowtail C^-$, is somewhat inscrutable: $\lambda^{<} a.{\bullet\bullet}\langle x\rangle.{¡}y.{\downarrow}z.N$. The SLS proof of this proposition, which uses patterns, is $(\lambda a, x, y, z.\, N)$. The pattern $P = a, x, y, z$ captures the structure of left inversion on the positive proposition $\exists a.\, \mathsf{p}\, a \bullet {¡}A^- \bullet {\downarrow}B^-$.

The grammar of patterns is straightforward. Inversion on positive propositions can only have the effect of introducing new bindings (either LF variables a or SLS variables x) or handling a unification $a \doteq_p t$, which by our discussion above can always be resolved by the most general unifier $[t/a]$, so the pattern associated with a proposition $a \doteq_p t$ is t/a.

$$P ::= () \mid x, P \mid a, P \mid t/a, P$$

For sequences with one or more elements, we omit the trailing comma and $()$, writing $x, \ldots, z$ instead of $x, \ldots, z, ()$.

SLS patterns have a list-like structure (the comma is right associative) because they capture the sequential structure of proofs. The associated decomposition judgment $P :: (\Psi; \Delta)_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}$ takes two process states. It operates a bit like the spine typing judgment from Figure 4.3 in that the process state $(\Psi; \Delta)_\sigma$ (and the pattern P) are treated as an input and the process state $(\Psi'; \Delta')_{\sigma'}$ is treated as an output. The typing rules for SLS patterns are given in

$\boxed{P :: (\Psi; \Delta)_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}$ – presumes $\vdash_{\Sigma,\mathcal{R}} (\Psi; \Delta)_\sigma$ state

$$\frac{\Psi \vdash_{\Sigma,\mathcal{R}} \Delta \text{ stable}}{() :: (\Psi; \Delta)_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi; \Delta)_\sigma}\ ()$$

$$\frac{P :: (\Psi; \Theta\{z{:}\langle p^+_{lvl}\rangle\ lvl\})_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}{z, P :: (\Psi; \Theta\{\!\{p^+_{lvl}\}\!\})_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}\ \eta^+ \qquad \frac{P :: (\Psi; \Theta\{x{:}A^-\ ord\})_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}{x, P :: (\Psi; \Theta\{\!\{\downarrow A^-\}\!\})_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}\ \downarrow_L$$

$$\frac{P :: (\Psi; \Theta\{x{:}A^-\ eph\})_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}{x, P :: (\Psi; \Theta\{\!\{¡A^-\}\!\})_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}\ ¡_L \qquad \frac{P :: (\Psi; \Theta\{x{:}A^-\ pers\})_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}{x, P :: (\Psi; \Theta\{\!\{!A^-\}\!\})_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}\ !_L$$

$$\frac{P :: (\Psi; \Theta\{\cdot\})_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}{P :: (\Psi; \Theta\{\!\{\mathbf{1}\}\!\})_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}\ \mathbf{1}_L \qquad \frac{P :: (\Psi; \Theta\{A^+, B^+\})_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}{P :: (\Psi; \Theta\{\!\{A^+ \bullet B^+\}\!\})_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}\ \bullet_L$$

$$\frac{P :: (\Psi, a{:}\tau; \Theta\{A^+\})_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}{a, P :: (\Psi; \Theta\{\!\{\exists a{:}\tau.A^+\}\!\})_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}\ \exists_L$$

$$\frac{P :: (\Psi, [t/a]\Psi'; [t/a]\Theta\{\cdot\})_{[t/a]\sigma} \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}{t/a, P :: (\Psi, a{:}\tau, \Psi'; \Theta\{\!\{a \doteq_\tau t\}\!\})_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}\ \doteq_L$$

Figure 4.7: SLS patterns

Figure 4.7. We preserve the side conditions from the previous chapter: when we frame off a inverting positive proposition in the process state, it is required to be the left-most one. As in focused OL$_3$, this justifies our omission of the variables associated with positive propositions: the positive proposition we frame off is always uniquely identified not by its associated variable but by its position in the context.

Note that there no longer appears to be a one-to-one correspondence between proof terms and rules: $\downarrow_L$, $¡_L$, and $!_L$ appear to have the same proof term, and $\mathbf{1}_L$ and $\bullet_L$ appear to have no proof term at all. To view patterns as being intrinsically typed – that is, to view them as actual representatives of (incomplete) derivations – we must think of patterns as carrying extra annotations that allow them to continue matching the structure of proof rules.

4.2.5 Values, terms, and spines

Notably missing from the SLS types are the upshifts $\uparrow A^+$ and right-permeable negative atomic propositions p^-_{lax}. The removal of these two propositions effectively means that the succedent of a stable SLS sequent can only be $\langle p^-\rangle\ true$ or $A^+\ lax$. The SLS framework only considers *complete* proofs of judgments $\langle p^-\rangle\ true$, whereas traces, associated with proofs of $A^+\ lax$ and introduced below in Section 4.2.6, are a proof term assignment for partial proofs. Excepting the proof term $\{\mathsf{let}\, T \,\mathsf{in}\, V\}$, which we present as part of the *concurrent* fragment of SLS in Section 4.2.6 below, the values, terms, and spines that stand for complete proofs will be referred

$\boxed{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} V : [A^+]}$ – presumes $\Psi \vdash_{\Sigma,\mathcal{R}} \Delta$ stable, and $\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} A^+$ prop$^+$

$$\frac{\Delta \textit{ matches } z{:}\langle A^+\rangle}{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} z : [A^+]}\ id^+$$

$$\frac{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} N : A^-}{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} N : [{\downarrow}A^-]}\ {\downarrow}_R \qquad \frac{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} N : A^-}{\Psi; \Delta\restriction_{eph} \vdash_{\Sigma,\mathcal{R}} ¡N : [¡A^-]}\ ¡_R \qquad \frac{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} N : A^-}{\Psi; \Delta\restriction_{pers} \vdash_{\Sigma,\mathcal{R}} !N : [!A^-]}\ !_R$$

$$\frac{\Delta \textit{ matches } \cdot}{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} () : [\mathbf{1}]}\ \mathbf{1}_R \qquad \frac{\Psi; \Delta_1 \vdash_{\Sigma,\mathcal{R}} V_1 : [A_1^+] \quad \Psi; \Delta_2 \vdash_{\Sigma,\mathcal{R}} V_2 : [A_2^+]}{\Psi; \Delta_1, \Delta_2 \vdash_{\Sigma,\mathcal{R}} V_1 \bullet V_2 : [A_1^+ \bullet A_2^+]}\ \bullet_R$$

$$\frac{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} t : \tau \quad \Psi; \Delta \vdash_{\Sigma,\mathcal{R}} V : [[t/a]A^+]}{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} t, V : [\exists a{:}\tau.A^+]}\ \exists_R \qquad \frac{\Delta \textit{ matches } \cdot}{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} \text{REFL} : t \doteq_\tau t}\ \doteq_R$$

Figure 4.8: SLS values

$\boxed{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} R : U}$ – presumes $\Psi \vdash_{\Sigma,\mathcal{R}} \Delta$ stable and $U = \langle C^-\rangle\ ord$

$$\frac{\Psi; \Theta\{[A^-]\} \vdash_{\Sigma,\mathcal{R}} Sp : U}{\Psi; \Theta\{\!\{x{:}A^-\}\!\} \vdash_{\Sigma,\mathcal{R}} x \cdot Sp : U}\ focus_L \qquad \frac{\mathsf{r} : A^- \in \Sigma \quad \Psi; \Theta\{[A^-]\} \vdash_{\Sigma,\mathcal{R}} Sp : U}{\Psi; \Theta\{\!\{\cdot\}\!\} \vdash_{\Sigma,\mathcal{R}} \mathsf{r} \cdot Sp : U}\ rule$$

Figure 4.9: SLS atomic terms

$\boxed{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} N : A^-\ ord}$ – presumes $\Psi \vdash_{\Sigma,\mathcal{R}} \Delta$ stable and $\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} A^-$ prop$^-$

$$\frac{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} R : \langle p^-\rangle\ ord}{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} R : p^-\ ord}\ \eta^-$$

$$\frac{P :: (\Psi; A^+, \Delta)_{\mathsf{id}_\Psi} \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_\sigma \quad \Psi'; \Delta' \vdash_{\Sigma,\mathcal{R}} N : \sigma B^-\ ord}{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} \lambda P.N : A^+ \rightarrowtail B^-\ ord}\ \rightarrowtail_R$$

$$\frac{P :: (\Psi; \Delta, A^+)_{\mathsf{id}_\Psi} \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_\sigma \quad \Psi'; \Delta' \vdash_{\Sigma,\mathcal{R}} N : \sigma B^-\ ord}{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} \lambda P.N : A^+ \twoheadrightarrow B^-\ ord}\ \twoheadrightarrow_R$$

$$\frac{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} N_1 : A_1^-\ ord \quad \Psi; \Delta \vdash_{\Sigma,\mathcal{R}} N_2 : A_2^-\ ord}{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} N_1 \mathbin{\&} N_2 : A_1^- \mathbin{\&} A_2^-\ ord}\ \&_R$$

$$\frac{\Psi, a{:}\tau; \Delta \vdash_{\Sigma,\mathcal{R}} N : A^-\ ord}{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} \lambda a.N : \forall a{:}\tau.A^-\ ord}\ \forall_R$$

$$\frac{T :: (\Psi; \Delta)_{\mathsf{id}_\Psi} \leadsto^*_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'} \quad \Psi'; \Delta' \vdash_{\Sigma,\mathcal{R}} V : [\sigma' A^+]}{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} \{\mathsf{let}\ T\ \mathsf{in}\ V\} : \bigcirc A^+\ ord}\ \bigcirc_R$$

Figure 4.10: SLS terms

$\boxed{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} Sp : U}$ – presumes $\Psi \vdash_{\Sigma,\mathcal{R}} \Delta\ \mathsf{infoc}$

$$\frac{\Delta\ \mathit{matches}\ [A^-]}{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} \textsc{nil} : \langle A^- \rangle\ \mathit{ord}}\ \mathit{id}^-$$

$$\frac{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} V : [A^+] \quad \Psi; \Theta\{[B^-]\} \vdash_{\Sigma,\mathcal{R}} Sp : U}{\Psi; \Theta\{\!\{\Delta, [A^+ \rightarrowtail B^-]\}\!\} \vdash_{\Sigma,\mathcal{R}} V; Sp : U}\ {\rightarrowtail}_L$$

$$\frac{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} V : [A^+] \quad \Psi; \Theta\{[B^-]\} \vdash_{\Sigma,\mathcal{R}} Sp : U}{\Psi; \Theta\{\!\{[A^+ \twoheadrightarrow B^-], \Delta\}\!\} \vdash_{\Sigma,\mathcal{R}} V; Sp : U}\ {\twoheadrightarrow}_L$$

$$\frac{\Psi; \Theta\{[A_1^-]\} \vdash_{\Sigma,\mathcal{R}} Sp : U}{\Psi; \Theta\{\!\{[A_1^- \mathbin{\&} A_2^-]\}\!\} \vdash_{\Sigma,\mathcal{R}} \pi_1; Sp : U}\ \&_{L1} \qquad \frac{\Psi; \Theta\{[A_2^-]\} \vdash_{\Sigma,\mathcal{R}} Sp : U}{\Psi; \Theta\{\!\{[A_1^- \mathbin{\&} A_2^-]\}\!\} \vdash_{\Sigma,\mathcal{R}} \pi_2; Sp : U}\ \&_{L2}$$

$$\frac{\Psi \vdash_{\Sigma,\mathcal{R}} t : \tau \quad [t/a]B^- = B'^- \quad \Psi; \Theta\{[B'^-]\} \vdash_{\Sigma,\mathcal{R}} Sp : U}{\Psi; \Theta\{\!\{[\forall a{:}\tau.B^-]\}\!\} \vdash_{\Sigma,\mathcal{R}} t; Sp : U}\ \forall_L$$

Figure 4.11: SLS spines

to as the *deductive fragment* of SLS.

SLS values (Figure 4.8)	$V ::= z \mid N \mid \text{¡}N \mid !N \mid () \mid V_1 \bullet V_2 \mid t, V \mid \textsc{refl}$
SLS atomic terms (Figure 4.9)	$R ::= x \cdot Sp \mid \mathsf{r} \cdot Sp$
SLS terms (Figure 4.10)	$N ::= R \mid \lambda P.N \mid N_1 \mathbin{\&} N_2 \mid \lambda a.N \mid \{\mathsf{let}\, T \,\mathsf{in}\, V\}$
SLS spines (Figure 4.11)	$Sp ::= \textsc{nil} \mid V; Sp \mid \pi_1; Sp \mid \pi_2; Sp \mid t; Sp$

In contrast to OL$_3$, we distinguish the syntactic category R of atomic terms that correspond to stable sequents. As with patterns, we appear to conflate the proof terms associated with different proof rules – we have a single $\lambda P.N$ constructor and a single $V; Sp$ spine rather than one term $\lambda^{>}N$ and spine $V^{>}Sp$ associated with propositions $A^+ \twoheadrightarrow B^-$ and another term $\lambda^{<}N$ and spine $V^{<}Sp$ associated with propositions $A^+ \rightarrowtail B^-$. As with patterns, it is possible to think of these terms as just having extra annotations ($\lambda^{>}$ or $\lambda^{<}$) that we have omitted. Without these annotations, proof terms carry less information than derivations, and the rules for values, terms, and spines in Figures 4.8–4.11 must be seen as typing rules. With these extra implicit annotations (or, possibly, with some of the technology of bidirectional typechecking), values, terms, and spines can continue to be seen as representatives of derivations.

Aside from $\bigcirc_R$ and its associated term $\{\mathsf{let}\, T \,\mathsf{in}\, V\}$, which belongs to the concurrent fragment of SLS, there is one rule in Figures 4.8–4.11 that does not have an exact analogues as a rule in OL$_3$, the rule labeled rule in Figure 4.9. This rule corresponds to an atomic term $\mathsf{r} \cdot Sp$ and accounts for the fact that there is an additional source of persistent facts in SLS, the signature Σ, that is not present in OL$_3$. To preserve the bijective correspondence between OL$_3$ and SLS proof terms, we need to place every rule $\mathsf{r} : A^-$ in the SLS signature Σ into the corresponding OL$_3$ context as a persistent proposition.

As with LF terms, we will use a shorthand for atomic terms $x \cdot Sp$ and $\mathsf{r} \cdot Sp$, writing $(\mathsf{foo}\, t\, s\, V\, V')$ instead of $\mathsf{foo} \cdot (t; s; V; V'; \textsc{nil})$ when we are not concerned with the fact that

$\boxed{S :: (\Psi; \Delta)_{\sigma} \leadsto_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}$ – presumes $\vdash_{\Sigma,\mathcal{R}} (\Psi; \Delta)_{\sigma}$ state and $\Psi \vdash_{\Sigma,\mathcal{R}} \Delta$ stable

$$\frac{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} R : \langle \bigcirc B^+ \rangle\ ord \quad P :: (\Psi, \Theta\{B^+\})_{\sigma} \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}{\{P\} \leftarrow R :: (\Psi; \Theta\{\!\{\Delta\}\!\})_{\sigma} \leadsto_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}$$

Figure 4.12: SLS steps

$\boxed{T :: (\Psi; \Delta)_{\sigma} \leadsto^*_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}$ – presumes $\vdash_{\Sigma,\mathcal{R}} (\Psi; \Delta)_{\sigma}$ state and $\Psi \vdash_{\Sigma,\mathcal{R}} \Delta$ stable

$$\frac{}{\diamond :: (\Psi; \Delta)_{\sigma} \leadsto^*_{\Sigma,\mathcal{R}} (\Psi; \Delta)_{\sigma}} \qquad \frac{S :: (\Psi; \Delta)_{\sigma} \leadsto_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}{S :: (\Psi; \Delta)_{\sigma} \leadsto^*_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}$$

$$\frac{T :: (\Psi_1; \Delta_1)_{\sigma_1} \leadsto^*_{\Sigma,\mathcal{R}} (\Psi_2; \Delta_2)_{\sigma_2} \quad T' :: (\Psi_2; \Delta_2)_{\sigma_2} \leadsto^*_{\Sigma,\mathcal{R}} (\Psi_3; \Delta_3)_{\sigma_3}}{T; T' :: (\Psi_1; \Delta_1)_{\sigma_1} \leadsto^*_{\Sigma,\mathcal{R}} (\Psi_3; \Delta_3)_{\sigma_3}}$$

Figure 4.13: SLS traces

atomic terms consist of a variable and a spine.

4.2.6 Steps and traces

The deductive fragment of SLS presented in Figures 4.7–4.11 covers every SLS proposition except for the lax modality $\bigcirc A^+$. It is in the context of the lax modality that we will present proof terms corresponding to partial proofs; we call this fragment the *concurrent* fragment of SLS because of its relationship with concurrent equality, described in Section 4.3.

Steps	$S ::= \{P\} \leftarrow R$
Traces	$T ::= \diamond \mid T_1; T_2 \mid S$

A step $S = \{P\} \leftarrow x \cdot Sp$ corresponds precisely to the notion of a *synthetic inference rule* as discussed in Section 2.4. A step in SLS corresponds to a use of left focus, a use of the left rule for the lax modality, and a use of the admissible focal substitution lemma in OL$_3$:

$$\dfrac{\dfrac{\begin{matrix}\vdots\\ \Psi; \Theta'\{[A^-]\} \vdash \langle \bigcirc B^+ \rangle\ true\end{matrix}}{\Psi; \Theta'\{x{:}A^-\ true\} \vdash \langle \bigcirc B^+ \rangle\ ord}\,focus_L \qquad \dfrac{\begin{matrix}\Psi'; \Delta' \vdash \sigma' A^+\ lax\\ \vdots\\ \Psi; \Theta\{B^+\} \vdash \sigma A^+\ lax\end{matrix}}{\Psi; \Theta\{[\bigcirc B^+]\} \vdash \sigma A^+\ lax}\,\bigcirc_L}{\Psi; \Theta\{\!\{\Theta'\{x{:}A^-\ ord\}\}\!\} \vdash \sigma A^+\ lax}\,subst^-$$

The spine Sp corresponds to the complete proof of $\Psi; \Theta'\{[A^-]\} \vdash \langle \bigcirc B^+ \rangle\ ord$, and the pattern P corresponds to the partial proof from $\Psi'; \Delta' \vdash \sigma' A^+\ lax$ to $\Psi; \Theta\{B^+\} \vdash \sigma A^+\ lax$. The typing rules for steps are given in Figure 4.12. Because we understand these synthetic inference rules as relations between process states, we call $(\Psi; \Delta) \leadsto_{\Sigma,\mathcal{R}} (\Psi'; \Delta')$ a *synthetic transition*. Traces T

are monoids over steps – $\diamond$ is an empty trace, S is a trace consisting of a single step, and $T_1; T_2$ is the sequential composition of traces. The typing rules for traces in Figure 4.13 straightforwardly reflect this monoid structure. Both of the judgments $S :: (\Psi; \Delta)_\sigma \leadsto_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}$ and $T :: (\Psi; \Delta)_\sigma \leadsto^*_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}$ work like the rules for patterns, in that the step S or trace T is treated as an input along with the initial process state $(\Psi; \Delta)_\sigma$, whereas the final process state $(\Psi'; \Delta')_{\sigma'}$ is treated as an output.

Steps incorporate left focus and the left rule for $\bigcirc$, and *let-expressions* $\{\mathsf{let}\, T \,\mathsf{in}\, V\}$, which include traces in deductive terms, incorporate right focus and the right rule for the lax modality in OL$_3$:

$$\dfrac{\dfrac{\vdots}{\Psi'; \Delta' \vdash [\sigma' A^+]}}{\Psi'; \Delta' \vdash \sigma' A^+ \; \mathit{lax}}\;\mathit{focus}_R$$

$$\dfrac{\begin{array}{c}\vdots\\ \Psi; \Delta \vdash A^+ \; \mathit{lax}\end{array}}{\Psi; \Delta \vdash \bigcirc A^+}\;\bigcirc_R$$

The trace T represents the entirety of the partial proof from $\Psi; \Delta \vdash A^+ \; \mathit{lax}$ to $\Psi'; \Delta' \vdash \sigma' A^+ \; \mathit{lax}$ that proceeds by repeated use of steps or synthetic transitions, and the eventual conclusion V represents the complete proof of $\Psi'; \Delta' \vdash [\sigma' A^+] \; \mathit{lax}$ that follows the series of synthetic transitions.

Both of the endpoints of a trace are stable sequents, but it will occasionally be useful to talk about steps and traces that start from unstable sequents and immediately decompose positive propositions. We will use the usual trace notation $(\Psi; \Delta)_\sigma \leadsto^*_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}$ to describe the type of these partial proofs. The proof term associated with this type will be written as $\lambda P.T$, where $P :: (\Psi; \Delta)_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi''; \Delta'')_{\sigma''}$ and $T :: (\Psi''; \Delta'')_{\sigma''} \leadsto^*_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}$.

4.2.7 Presenting traces

To present traces in a readable way, we will use a notation that interleaves process states among the steps of a trace, a common practice in Hoare-style reasoning [Hoa71]. As an example, recall the series of transitions that our money-store-battery-robot system took in Section 2.3.9:

$$\begin{array}{c} \textit{\$6 (1)} \\ \textit{battery-less robot (1)} \\ \textit{turn \$6 into a battery} \\ \textit{(all you want)} \end{array} \;\leadsto\; \begin{array}{c} \textit{battery (1)} \\ \textit{battery-less robot (1)} \\ \textit{turn \$6 into a battery} \\ \textit{(all you want)} \end{array} \;\leadsto\; \begin{array}{c} \textit{robot (1)} \\ \textit{turn \$6 into a battery} \\ \textit{(all you want)} \end{array}$$

This evolution can now be precisely captured as a trace in SLS:

$$\begin{array}{ll} & (x{:}\langle \mathsf{6bucks}\rangle \; \mathit{eph},\;\; f{:}(\mathsf{battery} \rightarrowtail \bigcirc\mathsf{robot}) \; \mathit{eph},\;\; g{:}(\mathsf{6bucks} \rightarrowtail \bigcirc\mathsf{battery}) \; \mathit{pers}) \\ \{y\} \leftarrow g\, x; & \\ & (y{:}\langle \mathsf{battery}\rangle \; \mathit{eph},\;\; f{:}(\mathsf{battery} \rightarrowtail \bigcirc\mathsf{robot}) \; \mathit{eph},\;\; g{:}(\mathsf{6bucks} \rightarrowtail \bigcirc\mathsf{battery}) \; \mathit{pers}) \\ \{z\} \leftarrow f\, y & \\ & (z{:}\langle \mathsf{robot}\rangle \; \mathit{eph},\;\; g{:}(\mathsf{6bucks} \rightarrowtail \bigcirc\mathsf{battery}) \; \mathit{pers}) \end{array}$$

4.2.8 Frame properties

The *frame rule* is a concept from separation logic [Rey02]. It states that if a property holds of some program, then the property holds under any extension of the mutable state. The frame rule increases the modularity of separation logic proofs, because two program fragments that reason about different parts of the state can be reasoned about independently.

Similar frame properties hold for SLS traces. The direct analogue of the frame rule is the observation that a trace can always have some extra state framed on to the outside. This is a generalization of weakening to SLS traces.

Theorem 4.2 (Frame weakening)**.**
If $T :: (\Psi; \Delta) \leadsto^*_{\Sigma,\mathcal{R}} (\Psi'; \Delta')$*, then* $T :: (\Psi, \Psi''; \Theta\{\Delta\}) \leadsto^*_{\Sigma,\mathcal{R}} (\Psi', \Psi''; \Theta\{\Delta\})$.

Proof. Induction on T and case analysis on the first steps of T, using admissible weakening and the properties of matching constructs at each step. □

The frame rule is a weakening property which ensures that new, irrelevant state can always be added to a state. Conversely, any state that is never accessed or modified by a trace can be always be removed without making the trace ill-typed. This property is a generalization of strengthening to SLS traces.

Theorem 4.3 (Frame strengthening)**.**
If $T :: (\Psi; \Theta\{x{:}Y\ \mathit{lvl}\}) \leadsto^*_{\Sigma,\mathcal{R}} (\Psi'; \Theta'\{x{:}Y\ \mathit{lvl}\})$ *and* x *is not free in any of the steps of* T*, then* $T :: (\Psi; \Theta\{\cdot\}) \leadsto^*_{\Sigma,\mathcal{R}} (\Psi'; \Theta\{\cdot\})$.

Proof. Induction on T and case analysis on the first steps of T, using a lemma to enforce that, if x is not free in an individual step, it is either not present in the context of the subderivation (if $\mathit{lvl} = \mathit{ord}$ or eph) or else it can be strengthened away (if $\mathit{lvl} = \mathit{pers}$). □

4.3 Concurrent equality

Concurrent equality is a notion of equivalence on traces that is coarser than the equivalence relation we would derive from partial OL$_3$ proofs. Consider the following SLS signature:

$$\begin{aligned}\Sigma = \cdot,\ & \mathsf{a} : \mathsf{prop\,lin},\ \mathsf{b} : \mathsf{prop\,lin},\ \mathsf{c} : \mathsf{prop\,lin},\ \mathsf{d} : \mathsf{prop\,lin},\ \mathsf{e} : \mathsf{prop\,lin},\ \mathsf{f} : \mathsf{prop\,lin},\\ & \mathsf{first} : \mathsf{a} \rightarrowtail \bigcirc(\mathsf{b} \bullet \mathsf{c}),\\ & \mathsf{left} : \mathsf{b} \rightarrowtail \bigcirc\mathsf{d},\\ & \mathsf{right} : \mathsf{c} \rightarrowtail \bigcirc\mathsf{e},\\ & \mathsf{last} : \mathsf{d} \bullet \mathsf{e} \rightarrowtail \bigcirc\mathsf{f}\end{aligned}$$

Under the signature Σ, we can create two traces with the type $x_a{:}\langle\mathsf{a}\rangle \leadsto^*_{\Sigma,\mathcal{R}} x_f{:}\langle\mathsf{f}\rangle$:

$$\begin{aligned}T_1 \;=\; & \{x_b, x_c\} \leftarrow \mathsf{first}\, x_a;\\ & \{x_d\} \leftarrow \mathsf{left}\, x_b;\\ & \{x_e\} \leftarrow \mathsf{right}\, x_c;\\ & \{x_f\} \leftarrow \mathsf{last}\, (x_d \bullet x_e)\end{aligned} \qquad \text{versus} \qquad \begin{aligned}T_2 \;=\; & \{x_b, x_c\} \leftarrow \mathsf{first}\, x_a;\\ & \{x_e\} \leftarrow \mathsf{right}\, x_c;\\ & \{x_d\} \leftarrow \mathsf{left}\, x_b;\\ & \{x_f\} \leftarrow \mathsf{last}\, (x_d \bullet x_e)\end{aligned}$$

In both cases, there is an $x_a{:}\langle \mathsf{a} \rangle$ resource that transitions to a resource $x_b{:}\langle \mathsf{b} \rangle$ and another resource $x_c{:}\langle \mathsf{c} \rangle$, and then $x_b{:}\langle \mathsf{b} \rangle$ transitions to $x_d{:}\langle \mathsf{d} \rangle$ while, independently, $x_c{:}\langle \mathsf{c} \rangle$ transitions to $x_d{:}\langle \mathsf{d} \rangle$. Then, finally, the $x_d{:}\langle \mathsf{d} \rangle$ and $x_e{:}\langle \mathsf{e} \rangle$ combine to transition to $x_f{:}\langle \mathsf{f} \rangle$, which completes the trace.

The independence here is key: if two steps consume different resources, then we want to treat them as independent concurrent steps that could have equivalently happened in the other order. However, if we define equivalence only in terms of the α-equivalence of partial OL$_3$ derivations, the two traces above are distinct. In this section, we introduce a coarser equivalence relation, *concurrent equality*, that allows us to treat traces that differ only in the interleaving of independent and concurrent steps as being equal. The previous section considered the proof terms of SLS as a fragment of OL$_3$ that is better able to talk about partial proofs. The introduction of concurrent equality takes a step beyond OL$_3$, because it breaks the bijective correspondence between OL$_3$ proofs and SLS proofs. As the example above indicates, there are simply more OL$_3$ proofs than SLS proofs when we quotient the latter modulo concurrent equality and declare T_1 and T_2 to be (concurrently) equal.

Concurrent equality was first was introduced and explored in the context of CLF [WCPW02], but our presentation follows the reformulation in [CPS$^+$12], which defines concurrent equivalence based on an analysis of the variables that are used (inputs) and introduced (outputs) by a given step. Specifically, our strategy will be to take a particular well-typed trace T and define a set I of pairs of states (S_1, S_2) with the property that, if $S_1; S_2$ is a well-typed trace, then $S_2; S_1$ is a concurrently equivalent and well-typed trace. This *independency relation* allows us to treat the trace T as a *trace monoid*. Concurrent equality, in turn, is just α-equality of SLS proof terms combined with treating $\{\mathsf{let}\, T \,\mathsf{in}\, V\}$ and $\{\mathsf{let}\, T' \,\mathsf{in}\, V\}$ as equivalent if T and T' are equivalent according to the equivalence relation imposed by treating T and T' as trace monoids.

This formulation of concurrent equality facilitates applying the rich theory developed around trace monoids to SLS traces. For example, it is decidable whether two traces T and T' are equivalent as trace monoids, and there are algorithms for determining whether T' is a subtrace of T (that is, whether there exist T_{pre} and T_{post} such that T is equivalent $T_{pre}; T'; T_{post}$) [Die90]. A different sort of matching problem, in which we are given T, T_{pre}, and T_{post} and must determine whether there exists a T' such that T is equivalent $T_{pre}; T'; T_{post}$, was considered in [CPS$^+$12].

Unfortunately, the presence of equality in SLS complicates our treatment of independency. The *interface* of a step is used to define independency on steps $S = (\{P\} \leftarrow R)$. Two components of the interface, the input variables ${}^\bullet S$ and the output variables $S^\bullet$ are standard in the literature on Petri nets – see, for example, [Mur89, p. 553]. The third component, unified variables ${}^\circledast S$, is unique to our presentation.

Definition 4.4 (Interface of a step)**.**

* *The* input variables *of a step, denoted* ${}^\bullet S$*, are all the LF variables* a *and SLS variables* x *free in the normal term* R*.*
* *The* output variables *of a step* $S = (\{P\} \leftarrow R)$*, denoted by* $S^\bullet$*, are all the LF variables* a *and SLS variables* x *bound by the pattern* P *that are not subsequently consumed by a substitution* t/a *in the same pattern.*
* *The* unified variables *of a step, denoted by* ${}^\circledast S$*, are the free variables of a step that are modified by unification. If* t/a *appears in a pattern and* a *is free in the pattern, then* $t = b$

for some other variable b; both a and b (if the latter is free in the pattern) are included in the step's unified variables.

Consider a well-typed trace $S_1; S_2$ with two steps. It is possible, by renaming variables bound in patterns, to ensure that $\emptyset = {}^\bullet S_1 \cap S_2{}^\bullet = {}^\bullet S_1 \cap S_1{}^\bullet = {}^\bullet S_2 \cap S_2{}^\bullet$. We will generally assume that, in the traces we consider, the variables introduced in each step's pattern are renamed to be distinct from the input or output variables of all previous steps.

If $S_1; S_2$ is a well-typed trace, then the order of S_1 and S_2 is fixed if S_1 introduces variables that are used by S_2 – that is, if $\emptyset \neq S_1{}^\bullet \cap {}^\bullet S_2$. For example, if $S_1 = (\{x_b, x_c\} \leftarrow \mathsf{first}\, x_a)$ and $S_2 = (\{x_d\} \leftarrow \mathsf{left}\, x_b)$, then $\{x_b\} = S_1{}^\bullet \cap {}^\bullet S_2$, and the two steps cannot be reordered relative to one another. Conversely, the condition that $\emptyset = S_1{}^\bullet \cap {}^\bullet S_2$ is sufficient to allow reordering in a CLF-like framework [CPS+12], and is also sufficient to allow reordering in SLS when neither step contains unified variables (that is, when $\emptyset = {}^\circledast S_1 = {}^\circledast S_2$). The unification driven by equality, however, can have subtle effects. Consider the following two-step trace:

$$\begin{array}{l} \qquad\qquad (a{:}p, b{:}p;\;\; x{:}(\bigcirc(b \doteq_p a))\; eph,\;\; y{:}(\mathsf{foo}\, b \rightarrowtail \bigcirc(\mathsf{bar}\, a))\; eph,\;\; z{:}\langle \mathsf{foo}\, a\rangle\; eph) \\ \{b/a\} \leftarrow x; \\ \qquad\qquad (b{:}p;\;\; y{:}(\mathsf{foo}\, b \rightarrowtail \bigcirc(\mathsf{bar}\, b))\; eph,\;\; z{:}\langle \mathsf{foo}\, b\rangle\; eph) \\ \{w\} \leftarrow y\, z \\ \qquad\qquad (b{:}p;\;\; w{:}\langle \mathsf{bar}\, b\rangle\; eph) \end{array}$$

This trace cannot be reordered even though $\emptyset = \emptyset \cap \{y, z\} = (\{b/a\} \leftarrow x)^\bullet \cap {}^\bullet(\{w\} \leftarrow y\, z)$, because the atomic term $y\, z$ is only well typed after the LF variables a and b are unified. It is not even sufficient to compare the free and unified variables (requiring that $\emptyset = {}^\circledast S_1 \cap {}^\bullet S_2$), as in the example above ${}^\circledast(\{b/a\} \leftarrow x) = \{a, b\}$ and ${}^\bullet(\{w\} \leftarrow y\, z) = \{y, z\}$ – and obviously $\emptyset = \{a, b\} \cap \{y, z\}$.

The simplest solution is to forbid steps with unified variables from being reordered at all: we can say that $(S_1, S_2) \in I$ if $\emptyset = S_1{}^\bullet \cap {}^\bullet S_2 = {}^\circledast S_1 = {}^\circledast S_2$. It is unlikely that this condition is satisfying in general, but it is sufficient for all the examples in this dissertation. Therefore, we will define concurrent equality on the basis of this simple solution. Nevertheless, three other possibilities are worth considering; all three are equivalent to this simple solution as far as the examples given here are concerned.

Restricting open propositions Part of the problem with the example above was that there were variables free in the *type* of a transition that were not free in the *term*. A solution is to restrict propositions so that negative propositions in the context are always *closed* relative to the LF context (or at least relative to the part of the LF context that mentions types subject to pure variable equality, which would be simple enough to determine with a subordination-based analysis). This restriction means that a step $S = \{P\} \leftarrow R$ can only have the parameter a free in R's type if a is free in R, allowing us to declare that S_1 and S_2 are reorderable – meaning (S_1, S_2) and (S_2, S_1) are in the independency relation I – whenever $\emptyset = S_1{}^\bullet \cap {}^\bullet S_2 = {}^\circledast S_1 \cap {}^\bullet S_2 = {}^\bullet S_1 \cap {}^\circledast S_2$.

While this restriction would be sufficient for the examples in this dissertation, it would preclude a conjectured extension of the destination-adding transformation given in Chapter 7 to nested specifications (nested versus flat specifications are discussed in Section 5.1).

Re-typing Another alternative would be to follow CLF and allow any reordering permitted by the input and output interfaces, but then forbid those that cannot be re-typed. (This was necessary in CLF to deal with the presence of $\top$.) This is very undesirable, however, because it leads to strange asymmetries. The following trace would be reorderable by this definition, for example, but in a symmetric case where the equality was $b \doteq_p a$ instead of $a \doteq_p b$, that would no longer be the case.

$$\begin{array}{ll} & (a{:}p, b{:}p;\;\; x{:}(\bigcirc(a \doteq_p b))\;\mathit{eph},\;\; y{:}(\forall w.\, \mathsf{foo}\, w \rightarrowtail \bigcirc(\mathsf{bar}\, w))\;\mathit{eph},\;\; z{:}\langle \mathsf{foo}\, b\rangle\;\mathit{eph}) \\ \{w\} \leftarrow y\, b\, z; & \\ & (a{:}p, b{:}p;\;\; x{:}(\bigcirc(a \doteq_p b))\;\mathit{eph},\;\; w{:}\langle \mathsf{bar}\, a\rangle\;\mathit{eph}) \\ \{b/a\} \leftarrow x & \\ & (b{:}p;\;\; w{:}\langle \mathsf{bar}\, b\rangle\;\mathit{eph}) \end{array}$$

Process states with equality constraints A third possible solution is be to change the way we handle the interaction of process states and unification. In this formulation of SLS, the process state $(\Psi; \Delta)_\sigma$ uses σ to capture the constraints that have been introduced by equality. As an alternative, we could have process states mention an explicit constraint store of equality propositions that have been encountered, as in Jagadeesan et al.'s formulation of concurrent constraint programming [JNS05]. Process states with equality constraints might facilitate talking explicitly about the interaction of equality and typing, which in our current formulation is left rather implicit.

4.3.1 Multifocusing

Concurrent equality is related to the equivalence relation induced by *multifocusing* [CMS09]. Like concurrent equality, multifocusing imposes a coarser equivalence relation on focused proofs. The coarser equivalence relation is enabled by a somewhat different mechanism: we are allowed to begin focus on multiple propositions simultaneously.

Both multifocusing and concurrent equality seek to address the sequential structure of focused proofs. The sequential structure of a computation needs to be addressed somehow, because it obscures the fact that the interaction between resources in a focused proof has the structure of a directed acyclic graph (DAG), not a sequence. We sketch a radically different, vaguely Feynman-diagram-inspired, way of presenting traces in Figure 4.14. Resources are the edges in the DAG and steps or synthetic inference rules are the vertexes. (The crossed edges that exchange x_2 and x_3 are only well-formed because, in our example trace, e and d were both declared to be ephemeral propositions.) Multifocusing gives a unique normal form to proofs by gathering all the focusing steps that can be rotated all the way to the beginning, then all the focusing steps that can happen as soon as those first steps have been rotated all the way to the beginning, etc.

$$\begin{tikzcd}
 & & \{x_1\} \leftarrow \mathsf{left}\, v_1 \arrow[dr, "x_1{:}\langle \mathsf{d}\rangle"] & & \\
{} \arrow[r, "u_1{:}\langle \mathsf{a}\rangle"] & \{v_1, w_1\} \leftarrow \mathsf{first}\, u_1 \arrow[ur, "v_1{:}\langle \mathsf{b}\rangle"] \arrow[dr, "w_1{:}\langle \mathsf{c}\rangle"'] & & \{z_1\} \leftarrow \mathsf{last}\,(x_1 \bullet y_1) \arrow[r, "z_1{:}\langle \mathsf{f}\rangle"] & {} \\
 & & \{y_1\} \leftarrow \mathsf{right}\, w_1 \arrow[ur, "y_1{:}\langle \mathsf{e}\rangle"'] & & \\
 & & \{x_2\} \leftarrow \mathsf{left}\, v_2 \arrow[ddddr, "x_2{:}\langle \mathsf{d}\rangle"] & & \\
{} \arrow[r, "u_2{:}\langle \mathsf{a}\rangle"] & \{v_2, w_2\} \leftarrow \mathsf{first}\, u_2 \arrow[ur, "v_2{:}\langle \mathsf{b}\rangle"] \arrow[dr, "w_2{:}\langle \mathsf{c}\rangle"'] & & \{z_2\} \leftarrow \mathsf{last}\,(x_3 \bullet y_2) \arrow[r, "z_2{:}\langle \mathsf{f}\rangle"] & {} \\
 & & \{y_2\} \leftarrow \mathsf{right}\, w_2 \arrow[ur, "y_2{:}\langle \mathsf{e}\rangle"] & & \\
 & & \{x_3\} \leftarrow \mathsf{left}\, v_3 \arrow[uur, "x_3{:}\langle \mathsf{d}\rangle"'] & & \\
{} \arrow[r, "u_3{:}\langle \mathsf{a}\rangle"] & \{v_3, w_3\} \leftarrow \mathsf{first}\, u_3 \arrow[ur, "v_3{:}\langle \mathsf{b}\rangle"] \arrow[dr, "w_3{:}\langle \mathsf{c}\rangle"'] & & \{z_3\} \leftarrow \mathsf{last}\,(x_2 \bullet y_3) \arrow[r, "z_3{:}\langle \mathsf{f}\rangle"] & {} \\
 & & \{y_3\} \leftarrow \mathsf{right}\, w_3 \arrow[ur, "y_3{:}\langle \mathsf{e}\rangle"'] & &
\end{tikzcd}$$

Figure 4.14: Interaction diagram for a trace $(u_1{:}\langle \mathsf{a}\rangle, u_2{:}\langle \mathsf{a}\rangle, u_3{:}\langle \mathsf{a}\rangle) \leadsto^*_\Sigma (z_1{:}\langle \mathsf{f}\rangle, z_2{:}\langle \mathsf{f}\rangle, z_3{:}\langle \mathsf{f}\rangle)$

In SLS, by contrast, we are content to represent the DAG structure as a list combined with the equivalence relation given by concurrent equality.

Multifocusing has only been explored carefully in the context of classical linear logic. We conjecture that derivations in OL_3 with a suitably-defined notion of multifocusing would be in bijective correspondence with SLS terms modulo concurrent equivalence, at least if we omit equality. Of course, without a formal notion of multifocusing for intuitionistic logic, this conjecture is impossible to state explicitly. The analogy with multifocusing may be able shed light on our difficulties in integrating concurrent equality and unification of pure variable types, because multifocusing has an independent notion of correctness: the equivalence relation given by multifocusing coincides with the the least equivalence relation that includes all permutations of independent rules in an unfocused sequent calculus proof [CMS09].

4.4 Adequate encoding

In Section 4.1.4 we discussed encoding untyped λ-calculus terms as LF terms of type exp, captured by the invertible function $\ulcorner e \urcorner$. Adequacy was extended to Linear LF (LLF) by Cervesato and Pfenning [CP02] and was extended to Ordered LF (OLF) by Polakow [Pol01]. The deductive fragment of SLS approximately extends both LLF and OLF, and the adequacy arguments made by Cervesato and Polakow extend straightforwardly to the deductive fragment of SLS. These adequacy arguments do not extend to the systems we want to encode in the concurrent fragment of SLS, however. The more general techniques we consider in this section will be explored further in Chapter 9 as a general technique for capturing invariants of SLS specifications.

The example that we will give to illustrate adequate encoding is the following signature, the SLS encoding of the push-down automata for parentheses matching from the introduction; we replace the atomic proposition $<$ with L and the proposition $>$ with R:

$$\begin{aligned}\Sigma_{PDA} = \cdot,\ & \mathsf{L} : \mathsf{prop\,ord},\\ & \mathsf{R} : \mathsf{prop\,ord},\\ & \mathsf{hd} : \mathsf{prop\,ord},\\ & \mathsf{push} : \mathsf{hd} \bullet \mathsf{L} \rightarrowtail \bigcirc(\mathsf{L} \bullet \mathsf{hd}),\\ & \mathsf{pop} : \mathsf{L} \bullet \mathsf{hd} \bullet \mathsf{R} \rightarrowtail \bigcirc(\mathsf{hd})\end{aligned}$$

We will relate this specification to a push-down automata defined in terms of stacks k and strings s, which we define inductively:

$$\begin{aligned}k &::= \cdot \mid k{<}\\ s &::= \cdot \mid {<}s \mid {>}s\end{aligned}$$

The transition system defined in terms of the stacks and strings has two transitions:

$$\begin{aligned}(k \rhd ({<}s)) &\mapsto ((k{<}) \rhd s)\\ ((k{<}) \rhd ({>}s)) &\mapsto (k \rhd s)\end{aligned}$$

Existing adequacy arguments for CLF specifications by Cervesato et al. [CPWW02] and by Schack-Nielsen [SN07] have a three-part structure structure. The first step is to define an encoding function $\ulcorner k \rhd s \urcorner = \Delta$ from PDA states $k \rhd s$ to process states Δ, so that, for example, the PDA state $(\cdot{<}{<}) \rhd ({>}{>}{>}{<}\cdot)$ is encoded as the process state

$$x_2{:}\langle\mathsf{L}\rangle\ ord,\ \ x_1{:}\langle\mathsf{L}\rangle\ ord,\ \ h{:}\langle\mathsf{hd}\rangle\ ord,\ \ y_1{:}\langle\mathsf{R}\rangle\ ord,\ \ y_2{:}\langle\mathsf{R}\rangle\ ord,\ \ y_3{:}\langle\mathsf{R}\rangle\ ord,\ \ y_4{:}\langle\mathsf{L}\rangle\ ord$$

The second step is to prove a preservation-like property: if $\ulcorner k \rhd s \urcorner \leadsto_{\Sigma_{PDA}} \Delta'$, then $\Delta' = \ulcorner k' \rhd s' \urcorner$ for some k' and s'. The third step is the main adequacy result: that $\ulcorner k \rhd s \urcorner \leadsto_{\Sigma_{PDA}} \ulcorner k' \rhd s' \urcorner$ if and only if $k \rhd s \mapsto k' \rhd s'$.

The second step is crucial in general: without it, we might transition in SLS from the encoding of some $k \rhd s$ to a state Δ' that is not in the image of encoding. We will take the opportunity to re-factor Cervesato et al.'s approach, replacing the second step with a general statement about transitions in Σ_{PDA} preserving a well-formedness invariant. The invariant we discuss is a simple instance of the well-formedness invariants that we will explore further in Chapter 9.

The first step in our revised methodology is to describe a generative signature Σ_{Gen} that precisely captures the set of process states that encode machine states (Theorem 4.6 below). The second step is showing that the generative signature Σ_{Gen} describes an invariant of the signature Σ_{PDA} (Theorem 4.7). The third step, showing that $\ulcorner k \rhd s \urcorner \leadsto_{\Sigma_{PDA}} \ulcorner k' \rhd s' \urcorner$ if and only if $k \rhd s \mapsto k' \rhd s'$, is straightforward and follows other developments.

4.4.1 Generative signatures

A critical aspect of any adequacy argument is an understanding of the structure of the relevant context(s) (the LF context in LF encodings, the substructural context in CLF encodings, both in

$\Sigma_{Gen} = \cdot,\ \mathsf{L} : \mathsf{prop\,ord},$	
$\mathsf{R} : \mathsf{prop\,ord},$	
$\mathsf{hd} : \mathsf{prop\,ord},$	
$\mathsf{gen} : \mathsf{prop\,ord},$	
$\mathsf{gen\_stack} : \mathsf{prop\,ord},$	
$\mathsf{gen\_string} : \mathsf{prop\,ord},$	
$\mathsf{state} : \mathsf{gen} \rightarrowtail \bigcirc(\mathsf{gen\_stack} \bullet \mathsf{hd} \bullet \mathsf{gen\_string})$	$G \to G_k\,\mathsf{hd}\,G_s$
$\mathsf{stack/left} : \mathsf{gen\_stack} \rightarrowtail \bigcirc(\mathsf{L} \bullet \mathsf{gen\_stack})$	$G_k \to\, < G_k$
$\mathsf{stack/done} : \mathsf{gen\_stack} \rightarrowtail \bigcirc(\mathbf{1})$	$G_k \to \epsilon$
$\mathsf{string/left} : \mathsf{gen\_string} \rightarrowtail \bigcirc(\mathsf{gen\_string} \bullet \mathsf{L})$	$G_s \to G_s <$
$\mathsf{string/right} : \mathsf{gen\_string} \rightarrowtail \bigcirc(\mathsf{gen\_string} \bullet \mathsf{R})$	$G_s \to G_s >$
$\mathsf{string/done} : \mathsf{gen\_string} \rightarrowtail \bigcirc(\mathbf{1})$	$G_s \to \epsilon$

Figure 4.15: Generative signature for PDA states and an analogous context-free grammar

SLS encodings). In the statement of adequacy for untyped λ-calculus terms (Section 4.1.4), for instance, it was necessary to require that the LF context Ψ take the form $a_1{:}\mathsf{exp}, \ldots, a_n{:}\mathsf{exp}$. In the adequacy theorems that have been presented for deductive logical frameworks, the structure of the context is quite simple. We can describe a set of building blocks that build small pieces of the context, and then define the set of valid process states (the *world*) any process state that can be built from a particular set of building blocks.

The structure of our PDA is too complex to represent as an arbitrary collection of building blocks. The process state is organized into three distinct zones:

[the stack] [the head] [the string being read]

We can't freely generate this structure with building blocks, but we can generate it with a context-free grammar. Conveniently, context-free grammars can be characterized within the machinery of SLS itself by describing *generative signatures* that can generate a set of process states we are interested in from a single seed context. The signature Σ_{Gen} in Figure 4.15 treats all the atomic propositions of Σ_{PDA} – the atomic propositions L, R and hd – as *terminals*, and introduces three *nonterminals* gen, $\mathsf{gen\_stack}$, and $\mathsf{gen\_string}$.

An informal translation of the signature Σ_{Gen} as a context-free grammar is given on the right-hand side of Figure 4.15. Observe that the sentences in the language G encode the states of our PDA as a string. We will talk much more about generative signatures in Chapter 9.

4.4.2 Restriction

The operation of *restriction* adapts the concept of "terminal" and "non-terminal" to SLS. Note that process states Δ such that $(x{:}\langle\mathsf{gen}\rangle\ \mathit{ord}) \leadsto^*_{\Sigma_{Gen}} \Delta$ are only well-formed under the signature

Σ_{PDA} if they are free of nonterminals; we can define an operation of *restriction* that filters out the non-terminal process states by checking whether they are well-formed in a signature that only declares the terminals.

Definition 4.5 (Restriction)**.**

* $\Psi\lightning_\Sigma$ *is a total function that returns the largest context* $\Psi' \subseteq \Psi$ *such that* $\vdash_\Sigma \Psi\ \mathsf{ctx}$ *(defined in Figure 4.3) by removing all the LF variables in* Ψ *whose types are not well-formed in the context* Σ*.*
* $(\Psi;\Delta)\lightning_\Sigma$ *is a partial function that is defined exactly when, for every variable declaration* $x{:}T\ \mathit{ord}$ *or* $x{:}T\ \mathit{eph}$ *in* Δ*, we have that* $(\Psi\lightning_\Sigma) \vdash_\Sigma T\ \mathsf{left}$ *(defined in Figure 4.6). When it is defined,* $(\Psi;\Delta)\lightning_\Sigma = ((\Psi\lightning_\Sigma);\Delta')$*, where* Δ' *is* Δ *except for the variable declarations* $x{:}T\ \mathit{pers}$ *in* Δ *for which it was not the case that* $(\Psi\lightning_\Sigma) \vdash_\Sigma T\ \mathsf{left}$.
* *We will also use* $(\Psi;\Delta)\lightning_\Sigma$ *as a judgment which expresses that the function is defined.*

Because restriction is only defined if all the ordered and linear propositions in Δ are well-formed in Σ; this means that $(x{:}\langle\mathsf{gen}\rangle\ \mathit{ord})\lightning_{\Sigma_{PDA}}$ is not defined. Restriction acts as a semi-permeable membrane on process states: some process states cannot pass through at all, and others pass through with some of their LF variables and persistent propositions removed. We can represent context restriction $(\Psi;\Delta)\lightning_\Sigma = (\Psi';\Delta')$ in a two-dimensional notation as a dashed line annotated with the restricting signature:

$$\begin{array}{rc} & (\Psi;\Delta) \\ \Sigma & /\!/\!/\!/\!/\!/\!/\!/ \\ & (\Psi';\Delta') \end{array}$$

For all process states that evolve from the initial state $(x{:}\langle\mathsf{gen}\rangle\ \mathit{ord})$ under the signature Σ_{Gen}, restriction to Σ_{PDA} is the identity function whenever it is defined. Therefore, in the statement of Theorem 4.6, we use restriction as a judgment $\Delta\lightning_{\Sigma_{PDA}}$ that holds whenever the partial function is defined.

Theorem 4.6 (Encoding)**.** *Up to variable renaming, there is a bijective correspondence between PDA states* $k \rhd s$ *and process states* Δ *such that* $T :: (x{:}\langle\mathsf{gen}\rangle\ \mathit{ord}) \leadsto^*_{\Sigma_{Gen}} \Delta$ *and* $\Delta\lightning_{\Sigma_{PDA}}$*.*

Proof. To establish the bijective correspondence, we first define an encoding function from PDA states to process states:

* $\ulcorner k \rhd s \urcorner = \ulcorner k \urcorner,\ \ h{:}\langle\mathsf{hd}\rangle\ \mathit{ord},\ \ \ulcorner s \urcorner$
* $\ulcorner \cdot \urcorner = \cdot$
* $\ulcorner k{<} \urcorner = \ulcorner k \urcorner,\ \ x{:}\langle\mathsf{L}\rangle\ \mathit{ord}$
* $\ulcorner {<}s \urcorner = y{:}\langle\mathsf{L}\rangle\ \mathit{ord},\ \ \ulcorner s \urcorner$
* $\ulcorner {>}s \urcorner = y{:}\langle\mathsf{R}\rangle\ \mathit{ord},\ \ \ulcorner s \urcorner$

It is always the case that $\ulcorner k \rhd s \urcorner\lightning_{\Sigma_{PDA}}$ – the encoding only includes terminals.

It is straightforward to observe that if $\ulcorner k \rhd s \urcorner = \ulcorner k' \rhd s' \urcorner$ if an only if $k = k'$ and $s = s'$. The interesting part of showing that context interpretation is an injective function is just showing that it is a function: that is, showing that, for any $k \rhd s$, there exists a trace

$T :: (x{:}\langle \mathsf{gen} \rangle\ ord) \leadsto^*_{Gen} \ulcorner k \rhd s \urcorner$. To show that the encoding function is surjective, we must show that if $T :: (x{:}\langle \mathsf{gen} \rangle\ ord) \leadsto^*_{\Sigma_{Gen}} \Delta$ and $\Delta \lightning_{\Sigma_{PDA}}$ then $\Delta = \ulcorner k \rhd s \urcorner$ for some k and s. This will complete the proof: an injective and surjective function is bijective.

Encoding is injective

We prove that for any $k \rhd s$, there exists a trace $T :: (x{:}\langle \mathsf{gen} \rangle\ ord) \leadsto^*_{Gen} \ulcorner k \rhd s \urcorner$ with a series of three lemmas.

Lemma. *For all k, there exists $T :: (x{:}\langle \mathsf{gen\_stack} \rangle\ ord) \leadsto^*_{\Sigma_{Gen}} (\ulcorner k \urcorner, x'{:}\langle \mathsf{gen\_stack} \rangle\ ord)$.*

By induction on k.

* If $k = \cdot$, $T = \diamond :: (x{:}\langle \mathsf{gen\_stack} \rangle\ ord) \leadsto^*_{\Sigma_{Gen}} (x{:}\langle \mathsf{gen\_stack} \rangle\ ord)$
* If $k = k'{<}$, we have $T' :: (x{:}\langle \mathsf{gen\_stack} \rangle\ ord) \leadsto^*_{\Sigma_{Gen}} (\ulcorner k' \urcorner, x''{:}\langle \mathsf{gen\_stack} \rangle\ ord)$ by the induction hypothesis, so $T = (T'; \{x_1, x_2\} \leftarrow \mathsf{stack/left}\, x'') :: (x{:}\langle \mathsf{gen\_stack} \rangle\ ord) \leadsto^*_{\Sigma_{Gen}} (\ulcorner k' \urcorner, x_1{:}\langle \mathsf{L} \rangle\ ord, x_2{:}\langle \mathsf{gen\_stack} \rangle\ ord)$

Lemma. *For all s, there exists $T :: (y{:}\langle \mathsf{gen\_string} \rangle\ ord) \leadsto^*_{\Sigma_{Gen}} (y'{:}\langle \mathsf{gen\_string} \rangle\ ord, \ulcorner s \urcorner)$.*

By induction on s.

* If $s = \cdot$, $T = \diamond :: (y{:}\langle \mathsf{gen\_string} \rangle\ ord) \leadsto^*_{\Sigma_{Gen}} (y{:}\langle \mathsf{gen\_string} \rangle\ ord)$
* If $s = {<}s'$, we have $T' :: (y{:}\langle \mathsf{gen\_string} \rangle\ ord) \leadsto^*_{\Sigma_{Gen}} (y''{:}\langle \mathsf{gen\_string} \rangle\ ord, \ulcorner s' \urcorner)$ by the induction hyp., so $T = (T'; \{y_1, y_2\} \leftarrow \mathsf{string/left}\, y'') :: (y{:}\langle \mathsf{gen\_string} \rangle\ ord) \leadsto^*_{\Sigma_{Gen}} (y_1{:}\langle \mathsf{gen\_string} \rangle\ ord, y_2{:}\langle \mathsf{L} \rangle\ ord, \ulcorner s' \urcorner)$
* If $s = {>}s'$, we have $T' :: (y{:}\langle \mathsf{gen\_string} \rangle\ ord) \leadsto^*_{\Sigma_{Gen}} (y''{:}\langle \mathsf{gen\_string} \rangle\ ord, \ulcorner s' \urcorner)$ by the induction hyp., so $T = (T'; \{y_1, y_2\} \leftarrow \mathsf{string/right}\, y'') :: (y{:}\langle \mathsf{gen\_string} \rangle\ ord) \leadsto^*_{\Sigma_{Gen}} (y_1{:}\langle \mathsf{gen\_string} \rangle\ ord, y_2{:}\langle \mathsf{R} \rangle\ ord, \ulcorner s' \urcorner)$

Lemma. *For all k and s, there exists $T :: (g{:}\langle \mathsf{gen} \rangle\ ord) \leadsto^*_{\Sigma_{Gen}} (\ulcorner k \rhd s \urcorner)$.*

By straightforward construction using the first two lemmas and frame weakening (Theorem 4.2):

$$
\begin{array}{ll}
 & (g{:}\langle \mathsf{gen} \rangle\ ord) \\
\{x, h, y\} \leftarrow \mathsf{state}\, g; & \\
 & (x{:}\langle \mathsf{gen\_stack} \rangle\ ord,\ \ h{:}\langle \mathsf{hd} \rangle\ ord,\ \ y{:}\langle \mathsf{gen\_string} \rangle\ ord) \\
T_k;\ \ \textit{(given by the first lemma and frame weakening)} & \\
 & (\ulcorner k \urcorner,\ \ x'{:}\langle \mathsf{gen\_stack} \rangle\ ord,\ \ h{:}\langle \mathsf{hd} \rangle\ ord,\ \ y{:}\langle \mathsf{gen\_string} \rangle\ ord) \\
\{()\} \leftarrow \mathsf{stack/done}\, x' & \\
 & (\ulcorner k \urcorner,\ \ h{:}\langle \mathsf{hd} \rangle\ ord,\ \ y{:}\langle \mathsf{gen\_string} \rangle\ ord) \\
T_s;\ \ \textit{(given by the second lemma and frame weakening)} & \\
 & (\ulcorner k \urcorner,\ \ h{:}\langle \mathsf{hd} \rangle\ ord,\ \ y'{:}\langle \mathsf{gen\_string} \rangle\ ord,\ \ \ulcorner s \urcorner) \\
\{()\} \leftarrow \mathsf{string/done}\, y' & \\
 & (\ulcorner k \urcorner,\ \ h{:}\langle \mathsf{hd} \rangle\ ord,\ \ \ulcorner s \urcorner) \\
 & = \ulcorner k \rhd s \urcorner
\end{array}
$$

Encoding is surjective

We prove that if $T :: (x{:}\langle \mathsf{gen} \rangle\ \mathit{ord}) \leadsto^*_{\Sigma_{Gen}} \Delta$ and $\Delta \lightning_{\Sigma_{PDA}}$ then $\Delta = \ulcorner k \rhd s \urcorner$ for some k and s with a series of two lemmas.

Lemma. *If* $T :: (\ulcorner k \urcorner, x{:}\langle \mathsf{gen\_stack} \rangle\ \mathit{ord}, h{:}\langle \mathsf{hd} \rangle\ \mathit{ord}, y{:}\langle \mathsf{gen\_store} \rangle\ \mathit{ord}, \ulcorner s \urcorner) \leadsto^*_{\Sigma_{Gen}} \Delta$ *and* $\Delta \lightning_{\Sigma_{PDA}}$*, then* $\Delta = \ulcorner k' \rhd s' \urcorner$ *for some* k' *and* s'*.*

By induction on the structure of T and case analysis on the first steps in T. Up to concurrent equality, there are four possibilities:

* $T = (\{()\} \leftarrow \mathsf{stack/done}\, x; \{()\} \leftarrow \mathsf{string/done}\, y)$ – this is a base case, and we can finish by letting $k' = k$ and $s' = s$.
* $T = (\{x_1, x_2\} \leftarrow \mathsf{stack/left}\, x; T')$ – apply the ind. hyp. (letting $x = x_2$, $k = k{<}$).
* $T = (\{y_1, y_2\} \leftarrow \mathsf{string/left}\, y; T')$ – apply the ind. hyp. (letting $y = y_1$, $s = {<}s$).
* $T = (\{y_1, y_2\} \leftarrow \mathsf{string/right}\, y; T')$ – apply the ind. hyp. (letting $y = y_1$, $s = {>}s$).

The proof above takes a number of facts about concurrent equality for granted. For example, the trace $T = (\{()\} \leftarrow \mathsf{stack/done}\, x; \{y_1, y_2\} \leftarrow \mathsf{string/right}\, y; T')$ does not syntactically match any of the traces above if we do not account for concurrent equality. Modulo concurrent equality, on the other hand, $T = (\{y_1, y_2\} \leftarrow \mathsf{string/right}\, y; \{()\} \leftarrow \mathsf{stack/done}\, x; T')$, matching the last branch of the case analysis. If we didn't implicitly rely on concurrent equality in this way, the resulting proof would have twice as many cases. We will take these finite uses of concurrent equality for granted when we specify that a proof proceeds by case analysis on the first steps of T (or, conversely, by case analysis on the last steps of T).

Lemma. *If* $T :: (g{:}\langle \mathsf{gen} \rangle\ \mathit{ord}) \leadsto^*_{\Sigma_{Gen}} \Delta$ *and* $\Delta \lightning_{\Sigma_{PDA}}$*, then* $\Delta = \ulcorner k' \rhd s' \urcorner$ *for some* k' *and* s'*.*

This is a corollary of the previous lemma, as it can only be the case that $T = \{x, h, y\} \leftarrow \mathsf{state}\, g; T'$. We can apply the previous lemma to T', letting $k = s = \cdot$. This establishes that encoding is a surjective function, which in turn completes the proof. □

Theorem 4.6 establishes that the generative signature Σ_{Gen} describes a world – a set of SLS process states – that precisely corresponds to the states of a push-down automata. We can (imperfectly) illustrate the content of this theorem in our two-dimensional notation as follows, where $\Delta \Leftrightarrow k \rhd s$ indicates the presence of a bijection:

$$\begin{tikzcd}
(x{:}\langle \mathsf{gen} \rangle\ \mathit{ord}) \arrow[d, squiggly, "\Sigma_{Gen}"', "*"] & \\
\Delta \arrow[d, phantom, "\Sigma_{PDA}\ ////"] & \\
\Delta \arrow[dr, Leftrightarrow] & \\
& k \rhd s
\end{tikzcd}$$

It is interesting to note how the proof of Theorem 4.6 takes advantage of the associative structure of traces: the inductive process that constructed traces in the first two lemmas treated trace composition as left-associative, but the induction we performed on traces in the next-to-last lemma treated trace composition as right-associative.

4.4.3 Generative invariants

The generative signature Σ_{Gen} precisely captures the world of SLS process states that are in the image of the encoding $\ulcorner k \rhd s \urcorner$ of PDA states as process states. In order for the signature Σ_{PDA} to encode a reasonable notion of transition between PDA states, we need to show that steps in this signature only take encoded PDA states to encoded PDA states. Because the generative signature Σ_{Gen} precisely captures the process states that represent encoded PDA states, we can describe and prove this property without reference to the actual encoding function:

Theorem 4.7 (Preservation)**.** *If* $T_1 :: (x{:}\langle \mathsf{gen} \rangle\ \mathit{ord}) \leadsto^*_{\Sigma_{Gen}} \Delta_1$*,* $\Delta_1 \lightning_{\Sigma_{PDA}}$*, and* $S :: \Delta_1 \leadsto_{\Sigma_{PDA}} \Delta_2$*, then* $T_2 :: (x{:}\langle \mathsf{gen} \rangle\ \mathit{ord}) \leadsto^*_{\Sigma_{Gen}} \Delta_2$*.*

If we illustrate the given elements as solid lines and elements that we have to prove as dashed lines, the big picture of the encoding and preservation theorems is the following:

$$\begin{tikzcd}
(x{:}\langle \mathsf{gen} \rangle\ \mathit{ord}) \arrow[d, squiggly, "\Sigma_{Gen}"', "*"] & & (x{:}\langle \mathsf{gen} \rangle\ \mathit{ord}) \arrow[d, squiggly, dotted, "\Sigma_{Gen}"', "*"] \\
\Delta \arrow[d, phantom, "/\!/\!/\!/"] & & \Delta' \arrow[d, phantom, "/\!/\!/\!/"] \\
\Delta \arrow[rr, squiggly, "\Sigma_{PDA}"] \arrow[dr, Leftrightarrow] & & \Delta' \\
& k \rhd s &
\end{tikzcd}$$

The proof of Theorem 4.7 relies on two lemmas, which we will consider before the proof itself. They are both *inversion lemmas*: they help uncover the structure of a trace based on the type of that trace. Treating traces modulo concurrent equality is critical in both cases.

Lemma. *Let* $\Delta = \Theta\{x{:}\langle \mathsf{gen\_stack} \rangle\ \mathit{ord}, h{:}\langle \mathsf{hd} \rangle\ \mathit{ord}, y{:}\langle \mathsf{gen\_string} \rangle\ \mathit{ord}\}$*. If* $T :: \Delta \leadsto^*_{\Sigma_{Gen}} \Delta'$ *and* $\Delta' \lightning_{\Sigma_{PDA}}$*, then* $T = (T'; \{()\} \leftarrow \mathsf{stack/done}\, x'; \{()\} \leftarrow \mathsf{string/done}\, y')$*, where* $T' :: \Delta \leadsto^*_{\Sigma_{Gen}} \Theta'\{x'{:}\langle \mathsf{gen\_stack} \rangle\ \mathit{ord}, h{:}\langle \mathsf{hd} \rangle\ \mathit{ord}, y'{:}\langle \mathsf{gen\_string} \rangle\ \mathit{ord}\}$ *and* $\Delta' = \Theta'\{h{:}\langle \mathsf{hd} \rangle\ \mathit{ord}\}$*. Or, as a picture:*

$$\begin{tikzcd}
\Delta = \Theta\{x{:}\langle \mathsf{gen\_stack} \rangle\ \mathit{ord}, h{:}\langle \mathsf{hd} \rangle\ \mathit{ord}, y{:}\langle \mathsf{gen\_string} \rangle\ \mathit{ord}\} \arrow[ddd, squiggly, "T"', "*"] \arrow[dr, squiggly, dotted, "T'"', "*"] & \\
{} \arrow[r, phantom, "="] & \Theta'\{x'{:}\langle \mathsf{gen\_stack} \rangle\ \mathit{ord}, h{:}\langle \mathsf{hd} \rangle\ \mathit{ord}, y'{:}\langle \mathsf{gen\_string} \rangle\ \mathit{ord}\} \arrow[d, dotted, "\{()\} \leftarrow \mathsf{stack/done}\, x'"'] \\
& \Theta'\{h{:}\langle \mathsf{hd} \rangle\ \mathit{ord}, y'{:}\langle \mathsf{gen\_string} \rangle\ \mathit{ord}\} \arrow[d, dotted, "\{()\} \leftarrow \mathsf{string/done}\, y'"'] \\
\Delta' & \Delta' = \Theta'\{h{:}\langle \mathsf{hd} \rangle\ \mathit{ord}\}
\end{tikzcd}$$

Proof. By induction on the structure of T and case analysis on the first steps in T. Up to concurrent equality, there are five possibilities:

* $T = (\{()\} \leftarrow \mathsf{stack/done}\, x; \{()\} \leftarrow \mathsf{string/done}\, y)$. Immediate, letting $T' = \diamond$.
* $T = (\{x_1, x_2\} \leftarrow \mathsf{stack/left}\, x; T'')$. By the induction hypothesis (where the new frame incorporates $x_1{:}\langle \mathsf{L}\rangle\ \mathit{ord}$), we have $T'' = (T'''; \{()\} \leftarrow \mathsf{stack/done}\, x; \{()\} \leftarrow \mathsf{string/done}\, y)$. Let $T' = (\{x_1, x_2\} \leftarrow \mathsf{stack/left}\, x; T''')$.
* $T = (\{y_1, y_2\} \leftarrow \mathsf{string/left}\, y; T'')$. By the induction hypothesis (where the new frame incorporates $y_1{:}\langle \mathsf{L}\rangle\ \mathit{ord}$), we have $T'' = (T'''; \{()\} \leftarrow \mathsf{stack/done}\, x; \{()\} \leftarrow \mathsf{string/done}\, y)$. Let $T' = (\{x_1, x_2\} \leftarrow \mathsf{string/left}\, y; T''')$.
* $T = (\{y_1, y_2\} \leftarrow \mathsf{string/right}\, y; T'')$. By the induction hypothesis (where the new frame incorporates $y_2{:}\langle \mathsf{R}\rangle\ \mathit{ord}$), we have $T'' = (T'''; \{()\} \leftarrow \mathsf{stack/done}\, x; \{()\} \leftarrow \mathsf{string/done}\, y)$. Let $T' = (\{x_1, x_2\} \leftarrow \mathsf{string/right}\, y; T''')$.
* $T = (S; T'')$, where x and y are not free in S. By the induction hypothesis, we have $T'' = (T'''; \{()\} \leftarrow \mathsf{stack/done}\, x; \{()\} \leftarrow \mathsf{string/done}\, y)$. Let $T' = (S; T''')$. (This case will not arise in the way we use this lemma, but the statement of the theorem leaves open the possibility that there are other nonterminals in Θ.)

This completes the proof. □

A corollary of this lemma is that if $T :: (g{:}\langle \mathsf{gen}\rangle\ \mathit{ord}) \leadsto^*_{\Sigma_{Gen}} \Delta$ and $\Delta \not\leadsto_{\Sigma_{PDA}}$, then $T = (T'; \{()\} \leftarrow \mathsf{stack/done}\, x; \{()\} \leftarrow \mathsf{string/done}\, y)$ – modulo concurrent equality, naturally – where $T' :: (g{:}\langle \mathsf{gen}\rangle\ \mathit{ord}) \leadsto^*_{\Sigma_{Gen}} \Theta\{x'{:}\langle \mathsf{gen\_stack}\rangle\ \mathit{ord}, h{:}\langle \mathsf{hd}\rangle\ \mathit{ord}, y'{:}\langle \mathsf{gen\_string}\rangle\ \mathit{ord}\}$ and $\Delta = \Theta\{h{:}\langle \mathsf{hd}\rangle\ \mathit{ord}\}$. To prove the corollary, we observe that $T = (\{x, h, r\} \leftarrow \mathsf{state}\, g\,; T'')$ and apply the lemma to T''.

Lemma. *The following all hold:*

* *If* $T :: (g{:}\langle \mathsf{gen}\rangle\ \mathit{ord}) \leadsto^*_{\Sigma_{Gen}} \Theta\{x_1{:}\langle \mathsf{L}\rangle\ \mathit{ord}, x_2{:}\langle \mathsf{gen\_stack}\rangle\ \mathit{ord}\}$, *then* $T = (T'; \{x_1, x_2\} \leftarrow \mathsf{stack/left}\, x')$ *for some* x'.
* *If* $T :: (g{:}\langle \mathsf{gen}\rangle\ \mathit{ord}) \leadsto^*_{\Sigma_{Gen}} \Theta\{y_1{:}\langle \mathsf{gen\_string}\rangle\ \mathit{ord}, y_2{:}\langle \mathsf{L}\rangle\ \mathit{ord}\}$, *then* $T = (T'; \{y_1, y_2\} \leftarrow \mathsf{string/left}\, y')$ *for some* y'.
* *If* $T :: (g{:}\langle \mathsf{gen}\rangle\ \mathit{ord}) \leadsto^*_{\Sigma_{Gen}} \Theta\{y_1{:}\langle \mathsf{gen\_string}\rangle\ \mathit{ord}, y_2{:}\langle \mathsf{R}\rangle\ \mathit{ord}\}$, *then* $T = (T'; \{y_1, y_2\} \leftarrow \mathsf{string/right}\, y')$ *for some* y'.

To give the last of the three statements as a picture:

$$\begin{tikzcd}
g{:}\langle \mathsf{gen}\rangle\ \mathit{ord} \arrow[dd, squiggly, "T"', "*"] & & g{:}\langle \mathsf{gen}\rangle\ \mathit{ord} \arrow[d, dotted, "T'"', "*"] \\
 & = & \Theta'\{y'{:}\langle \mathsf{gen\_string}\rangle\ \mathit{ord}\} \arrow[d, dotted, "\{y_1, y_2\} \leftarrow \mathsf{string/right}\, y'"'] \\
\Theta'\{y_1{:}\langle \mathsf{gen\_string}\rangle\ \mathit{ord}, y_2{:}\langle \mathsf{R}\rangle\ \mathit{ord}\} & & \Theta'\{y_1{:}\langle \mathsf{gen\_string}\rangle\ \mathit{ord}, y_2{:}\langle \mathsf{R}\rangle\ \mathit{ord}\}
\end{tikzcd}$$

Proof. The proofs are all by induction on the structure of T and case analysis on the last steps in T; we will prove the last statement, as the other two are similar. Up to concurrent equality, there are two possibilities:

- $*$ $T = (T'; \{y_1, y_2\} \leftarrow \mathsf{string/right}\, y')$ – Immediate.
- $*$ $T = (T''; S)$, where y_1 and y_2 are not among the input variables ${}^\bullet S$ or the output variables $S^\bullet$. By the induction hypothesis, $T'' = (T'''; \{y_1, y_2\} \leftarrow \mathsf{string/right}\, y')$. Let $T' = (T''; S)$.

This completes the proof. $\square$

Note that we do not consider any cases where $T = (T'; \{y_1', y_2\} \leftarrow \mathsf{string/right}\, y')$ (for $y_1 \neq y_1'$), $T = (T'; \{y_1', y_2\} \leftarrow \mathsf{string/right}\, y')$ (for $y_2 \neq y_2'$), or (critically) where $T = (T'; \{y_1, y_2'\} \leftarrow \mathsf{string/left}\, y')$. There is no way for any of these traces to have the correct type, which makes the resulting case analysis quite simple.

Proof of Theorem 4.7 (Preservation). By case analysis on the structure of S.

Case 1: $S = \{x', h'\} \leftarrow \mathsf{push}\,(h \bullet y)$, which means that we are given the following generative trace in Σ_{Gen}:

$$\begin{array}{ll} & (g{:}\langle \mathsf{gen}\rangle\, ord) \\ T & \\ & \Theta\{h{:}\langle \mathsf{hd}\rangle\, ord,\ \ y{:}\langle \mathsf{L}\rangle\, ord\} \end{array}$$

and we must construct a trace $(g{:}\langle \mathsf{gen}\rangle\, ord) \leadsto^*_{\Sigma_{Gen}} \Theta\{x'{:}\langle \mathsf{L}\rangle\, ord, h'{:}\langle \mathsf{hd}\rangle\, ord\}$. Changing h to h' is just renaming a bound variable, so we have

$$\begin{array}{ll} & (g{:}\langle \mathsf{gen}\rangle\, ord) \\ T' & \\ & \Theta\{h'{:}\langle \mathsf{hd}\rangle\, ord,\ \ y{:}\langle \mathsf{L}\rangle\, ord\} \end{array}$$

The corollary to the first inversion lemma above on T' gives us

$$\begin{array}{lll} T' = & & (g{:}\langle \mathsf{gen}\rangle\, ord) \\ & T''; & \\ & & \Theta\{x_g{:}\langle \mathsf{gen\_stack}\rangle\, ord,\ \ h'{:}\langle \mathsf{hd}\rangle\, ord,\ \ y_g{:}\langle \mathsf{gen\_string}\rangle\, ord,\ \ y{:}\langle \mathsf{L}\rangle\, ord\} \\ & \{()\} \leftarrow \mathsf{stack/done}\, x_g; & \\ & \{()\} \leftarrow \mathsf{string/done}\, y_g & \\ & & \Theta\{h'{:}\langle \mathsf{hd}\rangle\, ord,\ \ y{:}\langle \mathsf{L}\rangle\, ord\} \end{array}$$

The second inversion lemma (second part) on T'' gives us

$$\begin{array}{lll} T'' = & & (g{:}\langle \mathsf{gen}\rangle\, ord) \\ & T'''; & \\ & & \Theta\{x_g{:}\langle \mathsf{gen\_stack}\rangle\, ord,\ \ h'{:}\langle \mathsf{hd}\rangle\, ord,\ \ y_g'{:}\langle \mathsf{gen\_string}\rangle\, ord\} \\ & \{y_g, y\} \leftarrow \mathsf{string/left}\, y_g' & \\ & & \Theta\{x_g{:}\langle \mathsf{gen\_stack}\rangle\, ord,\ \ h'{:}\langle \mathsf{hd}\rangle\, ord,\ \ y_g{:}\langle \mathsf{gen\_string}\rangle\, ord,\ \ y{:}\langle \mathsf{L}\rangle\, ord\} \end{array}$$

Now, we can construct the trace we need using T''':

$$\begin{array}{ll} & \quad (g{:}\langle \mathsf{gen} \rangle\ \mathit{ord}) \\ T'''; & \\ & \quad \Theta\{x_g{:}\langle \mathsf{gen\_stack} \rangle\ \mathit{ord},\ \ h'{:}\langle \mathsf{hd} \rangle\ \mathit{ord},\ \ y'_g{:}\langle \mathsf{gen\_string} \rangle\ \mathit{ord}\} \\ \{x', x'_g\} \leftarrow \mathsf{stack/left}\, x_g; & \\ & \quad \Theta\{x'{:}\langle \mathsf{L} \rangle\ \mathit{ord},\ \ x'_g{:}\langle \mathsf{gen\_stack} \rangle\ \mathit{ord},\ \ h'{:}\langle \mathsf{hd} \rangle\ \mathit{ord},\ \ y'_g{:}\langle \mathsf{gen\_string} \rangle\ \mathit{ord}\} \\ \{()\} \leftarrow \mathsf{stack/done}\, x'_g; & \\ \{()\} \leftarrow \mathsf{string/done}\, y'_g & \\ & \quad \Theta\{x'{:}\langle \mathsf{L} \rangle\ \mathit{ord},\ \ h'{:}\langle \mathsf{hd} \rangle\ \mathit{ord}\} \end{array}$$

Case 2: $S = \{h'\} \leftarrow \mathsf{pop}\,(x \bullet h \bullet y)$, which means that we are given the following generative trace in Σ_{Gen}:

$$\begin{array}{ll} & \quad (g{:}\langle \mathsf{gen} \rangle\ \mathit{ord}) \\ T & \\ & \quad \Theta\{x{:}\langle \mathsf{L} \rangle\ \mathit{ord},\ \ h{:}\langle \mathsf{hd} \rangle\ \mathit{ord},\ \ y{:}\langle \mathsf{R} \rangle\ \mathit{ord}\} \end{array}$$

and we must construct a trace $(g{:}\langle \mathsf{gen} \rangle\ \mathit{ord}) \leadsto^*_{\Sigma_{Gen}} \Theta\{h'{:}\langle \mathsf{hd} \rangle\ \mathit{ord}\}$. Changing h to h' is just renaming a bound variable, so we have

$$\begin{array}{ll} & \quad (g{:}\langle \mathsf{gen} \rangle\ \mathit{ord}) \\ T' & \\ & \quad \Theta\{x{:}\langle \mathsf{L} \rangle\ \mathit{ord},\ \ h'{:}\langle \mathsf{hd} \rangle\ \mathit{ord},\ \ y{:}\langle \mathsf{R} \rangle\ \mathit{ord}\} \end{array}$$

The corollary to the first inversion lemma above on T' gives us

$$\begin{array}{lll} T' = & & \quad (g{:}\langle \mathsf{gen} \rangle\ \mathit{ord}) \\ & T''; & \\ & & \quad \Theta\{x{:}\langle \mathsf{L} \rangle\ \mathit{ord},\ \ x_g{:}\langle \mathsf{gen\_stack} \rangle\ \mathit{ord},\ \ h'{:}\langle \mathsf{hd} \rangle\ \mathit{ord},\ \ y_g{:}\langle \mathsf{gen\_string} \rangle\ \mathit{ord},\ \ y{:}\langle \mathsf{R} \rangle\ \mathit{ord}\} \\ & \{()\} \leftarrow \mathsf{stack/done}\, x_g; & \\ & \{()\} \leftarrow \mathsf{string/done}\, y_g & \\ & & \quad \Theta\{x{:}\langle \mathsf{L} \rangle\ \mathit{ord},\ \ h'{:}\langle \mathsf{hd} \rangle\ \mathit{ord},\ \ y{:}\langle \mathsf{R} \rangle\ \mathit{ord}\} \end{array}$$

The second inversion lemma (first part) on T'' gives us

$$\begin{array}{lll} T'' = & & \quad (g{:}\langle \mathsf{gen} \rangle\ \mathit{ord}) \\ & T'''; & \\ & & \quad \Theta\{x'_g{:}\langle \mathsf{gen\_stack} \rangle\ \mathit{ord},\ \ h'{:}\langle \mathsf{hd} \rangle\ \mathit{ord},\ \ y_g{:}\langle \mathsf{gen\_string} \rangle\ \mathit{ord},\ \ y{:}\langle \mathsf{R} \rangle\ \mathit{ord}\} \\ & \{x, x_g\} \leftarrow \mathsf{stack/left}\, x'_g & \\ & & \quad \Theta\{x{:}\langle \mathsf{L} \rangle\ \mathit{ord},\ \ x_g{:}\langle \mathsf{gen\_stack} \rangle\ \mathit{ord},\ \ h'{:}\langle \mathsf{hd} \rangle\ \mathit{ord},\ \ y_g{:}\langle \mathsf{gen\_string} \rangle\ \mathit{ord},\ \ y{:}\langle \mathsf{R} \rangle\ \mathit{ord}\} \end{array}$$

The second inversion lemma (third part) on T''' gives us

$$
\begin{array}{lll}
T''' = & & (g{:}\langle \mathsf{gen} \rangle\ \mathit{ord}) \\
& T'''';& \\
& & \Theta\{x_g'{:}\langle \mathsf{gen\_stack} \rangle\ \mathit{ord},\ \ h'{:}\langle \mathsf{hd} \rangle\ \mathit{ord},\ \ y_g'{:}\langle \mathsf{gen\_string} \rangle\ \mathit{ord}\} \\
& \{y_g, y\} \leftarrow \mathsf{string/right}\, y_g' & \\
& & \Theta\{x_g'{:}\langle \mathsf{gen\_stack} \rangle\ \mathit{ord},\ \ h'{:}\langle \mathsf{hd} \rangle\ \mathit{ord},\ \ y_g{:}\langle \mathsf{gen\_string} \rangle\ \mathit{ord},\ \ y{:}\langle \mathsf{R} \rangle\ \mathit{ord}\}
\end{array}
$$

Now, we can construct the trace we need using T'''':

$$
\begin{array}{ll}
& (g{:}\langle \mathsf{gen} \rangle\ \mathit{ord}) \\
T'''';& \\
& \Theta\{x_g'{:}\langle \mathsf{gen\_stack} \rangle\ \mathit{ord},\ \ h'{:}\langle \mathsf{hd} \rangle\ \mathit{ord},\ \ y_g'{:}\langle \mathsf{gen\_string} \rangle\ \mathit{ord}\} \\
\{()\} \leftarrow \mathsf{stack/done}\, x_g'; & \\
\{()\} \leftarrow \mathsf{string/done}\, y_g' & \\
& \Theta\{h'{:}\langle \mathsf{hd} \rangle\ \mathit{ord}\}
\end{array}
$$

These two cases represent the only two synthetic transitions that are possible under the signature Σ_{PDA}, so we are done. □

Theorem 4.7 establishes that the generative signature Σ_{Gen} is a *generative invariant* of the signature Σ_{PDA}. We consider theorems of this form further in Chapter 9, but they all essentially follow the structure of Theorem 4.7. First, we enumerate the synthetic transitions associated with a given signature. Second, in each of those cases, we use the type of the synthetic transition to perform inversion on the structure of the given generative trace. Third, we construct a generative trace that establishes the fact that the invariant was preserved.

4.4.4 Adequacy of the transition system

The hard work of adequacy is established by the preservation theorem; the actual adequacy theorem is just an enumeration in both directions.

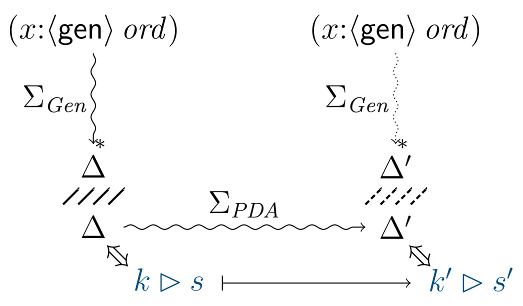
Theorem 4.8 (Adequacy). $\ulcorner k \rhd s \urcorner \leadsto_{\Sigma_{PDA}} \ulcorner k' \rhd s' \urcorner$ *if and only if* $k \rhd s \mapsto k' \rhd s'$.

Proof. Both directions can be established by case analysis on the structure of k and s. □

As an immediate corollary of this theorem and preservation (Theorem 4.7), we have the stronger adequacy property that $\ulcorner k \rhd s \urcorner \leadsto_{\Sigma_{PDA}} \Delta'$, then $\Delta' = \ulcorner k' \rhd s' \urcorner$ for some k and s' such that $k \rhd s \mapsto k' \rhd s'$. In our two-dimensional notation, the complete discussion of adequacy for SLS is captured by the following picture:

$\downarrow A^- = \texttt{A}$	$\bigcirc A^+ = \{\texttt{A}\}$	$\lambda a.t = \texttt{\textbackslash a.t}$
$¡A^- = \texttt{\$A}$	$A^+ \rightarrowtail B^- = \texttt{A >-> B}$	$\mathsf{foo}\, t_1 \ldots t_n = \texttt{foo t1...tn}$
$!A^- = \texttt{!A}$	$A^+ \twoheadrightarrow B^- = \texttt{A ->> B}$	
$\mathbf{1} = \texttt{one}$	$A^- \mathbin{\&} B^- = \texttt{A \& B}$	$\Pi a{:}\tau.\nu = \texttt{Pi x.nu}$
$A^+ \bullet B^+ = \texttt{A * B}$	$\forall a.\tau.A^- = \texttt{All x.A}$	$\tau \to \nu = \texttt{tau -> nu}$
$\exists a.\tau.A^+ = \texttt{Exists x.A}$	$¡A^- \rightarrowtail B^- = \texttt{A -o B}$	$\mathsf{bar}\, t_1 \ldots t_n = \texttt{bar t1...tn}$
$t \doteq_\tau s = \texttt{t == s}$	$!A^- \rightarrowtail B^- = \texttt{A -> B}$	

Figure 4.16: Mathematical and ASCII representations of propositions, terms, and classifiers



4.5 The SLS implementation

The prototype implementation of SLS contains a parser and typechecker for the SLS language, and is available from `https://github.com/robsimmons/sls`. Code that is checked by this prototype implementation will appear frequently in the rest of this document, always in a `fixed-width font`.

The checked SLS code differs slightly from mathematical SLS specifications in a few ways – the translation between the mathematical notation we use for SLS propositions and the ASCII representation used in the implementation is outlined in Figure 4.16. Following CLF and the Celf implementation, we write the lax modality $\bigcirc A$ in ASCII as `{A}` – recall that in Section 4.2 we introduced the $\{A^+\}$ notation from CLF as a synonym for Fairtlough and Mendler's $\bigcirc A^+$. The exponential $¡A$ doesn't have an ASCII representation, so we write `$A`. Upshifts and downshifts are always inferred: this means that we can't write down $\uparrow\downarrow A$ or $\downarrow\uparrow A$, but neither of these OL_3 propositions are part of the SLS fragment anyway.

The SLS implementation also supports conventional abbreviations for arrows that we won't use in mathematical notation: $¡A^- \rightarrowtail B^-$ can be written as `A -o B` or `$A >-> B` in the SLS implementation, and $!A^- \rightarrowtail B^-$ can be written as `A -> B` or `!A >-> B`. This final proposition is ambiguous, because `X -> Y` can be an abbreviation for $!X \rightarrowtail Y$ or $\Pi a{:}X.Y$, but SLS can figure out which form was intended by analyzing the structure of `Y`. Also note that we could have just as easily made `A -o B` an abbreviation for `$A ->> B`, but we had to pick one and the choice absolutely doesn't matter. All arrows can also be written backwards: `B <-< A` is equivalent to `A >-> B`, `B o- A` is equivalent to `A -o B`, and so on.

Also following traditional conventions, upper-case variables that are free in a rule will be treated as implicitly quantified. Therefore, the line

```
rule: foo X <- (bar Y -> baz Z).
```

will be reconstructed as the SLS declaration

$$\mathsf{rule} : \forall Y{:}\tau_1.\, \forall Z{:}\tau_2.\, \forall X{:}\tau_3.\, !(!\mathsf{bar}\, Y \rightarrowtail \mathsf{baz}\, Z) \rightarrowtail \mathsf{foo}\, X$$

where the implementation infers the types τ_1, τ_2, and τ_3 appropriately from the declarations of the negative predicates foo, bar, and baz. The type annotation associated with equality is similarly inferred.

Another significant piece of syntactic sugar introduced for the sake of readability is less conventional, if only because positive atomic propositions are not conventional. If `P` is a persistent atomic proposition, we can optionally write `!P` wherever `P` is expected, and if `P` is a linear atomic proposition, we can write `$P` wherever `P` is expected. This means that if a, b, and c are (respectively) ordered, linear, and persistent positive atomic propositions, we can write the positive proposition a • b • c in the SLS implementation as `(a * b * c)`, `(a * $b * c)`, `(a * b * !c)`, or `(a * $b * !c)`. Without these annotations, it is difficult to tell at a glance which propositions are ordered, linear, or persistent when a signature uses more than one variety of proposition. When all of these optional annotations are included, the rules in a signature that uses positive atomic propositions look the same as rules in a signature that uses the pseudo-positive negative atomic propositions described in Section 4.7.1.

In the code examples given in the remainder of this document, we will use these optional annotations in a consistent way. We will omit the optional `$A` annotations only in specifications with no ordered atomic propositions, and we will omit the optional `!A` annotations in specifications with no ordered or linear atomic propositions. This makes the mixture of different exponentials explicit while avoiding the need for rules like `($a * $b * $c >-> {$d * $e})` when specifications are entirely linear (and likewise when specifications are entirely persistent).

4.6 Logic programming

One logic programming interpretation of CLF was explored by the Lollimon implementation [LPPW05] and adapted by the Celf implementation [SNS08, SN11]. Logic programming interpretations of SLS are not a focus this dissertation, but we will touch on a few points in this section.

Logic programming is important because it provides us with operational intuitions about the intended behavior of the systems we specify in SLS. One specific set of intuitions will form the basis of the operationalization transformations on SLS specifications considered in Chapter 6. Additionally, logic programming intuitions are relevant because they motivated the design of SLS, in particular the presentation of the concurrent fragment in terms of partial, rather than complete, proofs. We discuss this point in Section 4.6.2.

4.6.1 Deductive computation and backward chaining

Deductive computation in SLS is the search for *complete* proofs of sequents of the form $\Psi; \Delta \vdash \langle p^- \rangle\ \mathit{true}$. A common form of deductive computation is *goal-directed search*, or what Andreoli calls the *proof construction paradigm* [And01]. In SLS, goal-directed search for the proof of a sequent $\Psi; \Delta \vdash \langle p^- \rangle\ \mathit{true}$ can only proceed by focusing on a proposition like $\downarrow p_n^- \rightarrowtail \ldots \rightarrowtail \downarrow p_1^- \rightarrowtail p^-$ which has a head p^- that matches the succedent. This replaces the goal sequent $\Psi; \Delta \vdash \langle p^- \rangle\ \mathit{true}$ with n subgoals: $\Psi; \Delta_1 \vdash \langle p_1^- \rangle\ \mathit{true} \ldots \Psi; \Delta_n \vdash \langle p_n^- \rangle\ \mathit{true}$, where Δ matches $\Delta_1, \ldots, \Delta_n$.

When goal-directed search only deals with the unproved subgoals of a single coherent derivation at a time, it is called *backward chaining*, because we're working backwards from the goal we want to prove.[3] The term *top-down logic programming* is also used, and refers to the fact that, in the concrete syntax of Prolog, the rule $\downarrow p_n^- \rightarrowtail \ldots \rightarrowtail \downarrow p_1^- \rightarrowtail p^-$ would be written with p^- on the first line, p_1^- on the second, etc. This is exactly backwards from a proof-construction perspective, as we think of backward chaining as building partial proofs from the bottom up, the root towards the leaves, so we will avoid this terminology.

The backward-chaining interpretation of intuitionistic logics dates back to the work by Miller et al. on uniform proofs [MNPS91]. An even older concept, Clark's *negation-as-failure* [Cla87], is based on a *partial completeness* criteria for logic programming interpreters. Partial completeness demands that if the interpreter gives up up on finding a proof, no proof should exist. (The interpreter is allowed to run forever without succeeding or giving up.) Partial completeness requires *backtracking* in backward-chaining search: if we we try to prove $\Psi; \Delta \vdash \langle p^- \rangle\ \mathit{true}$ by focusing on a particular proposition and one of the resulting subgoals fails to be provable, we have to consider any other propositions that could have been used to prove the sequent before giving up. Backtracking can be extremely powerful in certain cases and incredibly expensive in others, and so most logic programming languages have an escape hatch that modifies or limits backtracking at the user's discretion, such as the Prolog cut (no relation to the admissible rule cut) or Twelf's deterministic declarations. Non-backtracking goal-oriented deductive computation is called *flat resolution* [AK99].

One feature of backward chaining and goal directed search is that it usually allows for terms that are not completely specified – these unspecified pieces are are traditionally called *logic variables*. Because LF variables are also "logic variables," the literature on λProlog and Twelf calls unspecified pieces of terms *existential variables*, but as they bear no relation to the variables introduced by the left rule for $\exists a{:}\tau.A^+$, that terminology is also unhelpful here. Consider the

[3]The alternative is to try and derive the same sequent in multiple ways simultaneously, succeeding whenever some way of proving the sequent is discovered. Unlike backward chaining, this strategy of breadth-first search is complete: if a proof exists, it will be found. Backward chaining as we define it is only nondeterministically or partially complete, because it can fail to terminate when a proof exists. We will call this alternative to backtracking *breadth-first theorem proving*, as it amounts to taking a breadth-first, instead of depth-first, view of the so-called *failure continuation* [Pfe12].

following SLS signature:

$$\begin{aligned}\Sigma_{Add} = \cdot,\ & \mathsf{nat} : \mathsf{type},\ \ \mathsf{z} : \mathsf{nat},\ \ \mathsf{s} : \mathsf{nat} \rightarrow \mathsf{nat},\\ & \mathsf{plus} : \mathsf{nat} \rightarrow \mathsf{nat} \rightarrow \mathsf{nat} \rightarrow \mathsf{prop},\\ & \mathsf{plus/z} : \forall N{:}\mathsf{nat}.\,(\mathsf{plus}\,\mathsf{z}\,N\,N),\\ & \mathsf{plus/s} : \forall N{:}\mathsf{nat}.\,\forall M{:}\mathsf{nat}.\,\forall P{:}\mathsf{nat}.\,{!}(\mathsf{plus}\,N\,M\,P) \rightarrowtail (\mathsf{plus}\,(\mathsf{s}\,N)\,M\,(\mathsf{s}\,P))\end{aligned}$$

In addition to searching for a proof of $\mathsf{plus}\,(\mathsf{s}\,\mathsf{z})\,(\mathsf{s}\,\mathsf{z})\,(\mathsf{s}\,(\mathsf{s}\,\mathsf{z}))$ (which will succeed, as $1 + 1 = 2$) or searching for a proof of $\mathsf{plus}\,(\mathsf{s}\,\mathsf{z})\,(\mathsf{s}\,\mathsf{z})\,(\mathsf{s}\,(\mathsf{s}\,(\mathsf{s}\,\mathsf{z})))$ (which will fail, as $1 + 1 \neq 3$), we can use goal-oriented deductive computation to search for $\mathsf{plus}\,(\mathsf{s}\,\mathsf{z})\,(\mathsf{s}\,\mathsf{z})\,X$, where X represents an initially unspecified term. This search will succeed, reporting that $X = (\mathsf{s}\,(\mathsf{s}\,\mathsf{z}))$. Unification is generally used in backward-chaining logic programming languages as a technique for implementing partially unspecified terms, but this implementation technique should not be confused with our use of unification-based equality $t \doteq s$ as a proposition in SLS.

We say that plus in the signature above is a *well-moded* predicate with *mode* $(\mathsf{plus} + + -)$, because whenever we perform deductive computation to derive $(\mathsf{plus}\,n\,m\,p)$ where n and m are fully specified, any unspecified portion of p must be fully specified in any completed derivation. Well-moded predicates can be treated as nondeterministic (in the sense of potentially having zero, one, or many outputs) partial functions from their inputs (the indices marked "$+$" in the mode) to their outputs (the indices marked "$-$" in the mode). A predicate can sometimes be given more than one mode: $(\mathsf{plus} + - +)$ is a valid mode for plus, but $(\mathsf{plus} + - -)$ is not.

The implementation of backward chaining in substructural logic has been explored by Hodas [HM94], Polakow [Pol00, Pol01], Armelín and Pym [AP01], and others. Efficient implementation of these languages is complicated by the problem of *resource management*. In linear logic proof search, it would be technically correct but highly inefficient to perform proof search by enumerating the ways that a context can be split and then backtracking over each possible split. Resource management allows the interpreter to avoid this potentially exponential backtracking, but describing resource management and proving it correct, especially for richer substructural logics, can be complex and subtle [CHP00].

The term *deductive computation* is meant to be interpreted very broadly, and goal-directed search is not the only form of deductive computation. Another paradigm for deductive computation is the *inverse method*, where the interpreter attempts to prove a sequent $\Psi; \Delta \vdash \langle p^- \rangle$ *true* by creating and growing database of sequents that are derivable, attempting to build the appropriate derivation from the leaves down. The inverse method is generally associated with theorem proving and not logic programming. However, Chaudhuri, Pfenning, and Price have shown that that deductive computation with the inverse method in a focused linear logic can simulate both backward chaining and forward chaining (considered below) for persistent Horn-clause logic programs [CPP08].

Figure 4.17 gives an taxonomy (incomplete and imperfect) of the forms of deductive computation mentioned in this section. Note that, while we will generally use *backward chaining* to describe backtracking search, backward chaining does not always imply full backtracking and partial completeness. This illustration, and the preceding discussion, leaves out many important categories, especially tabled logic programming, and many potentially relevant implementation choices, such as breath-first versus depth-first or parallel exploration of the success continuation.

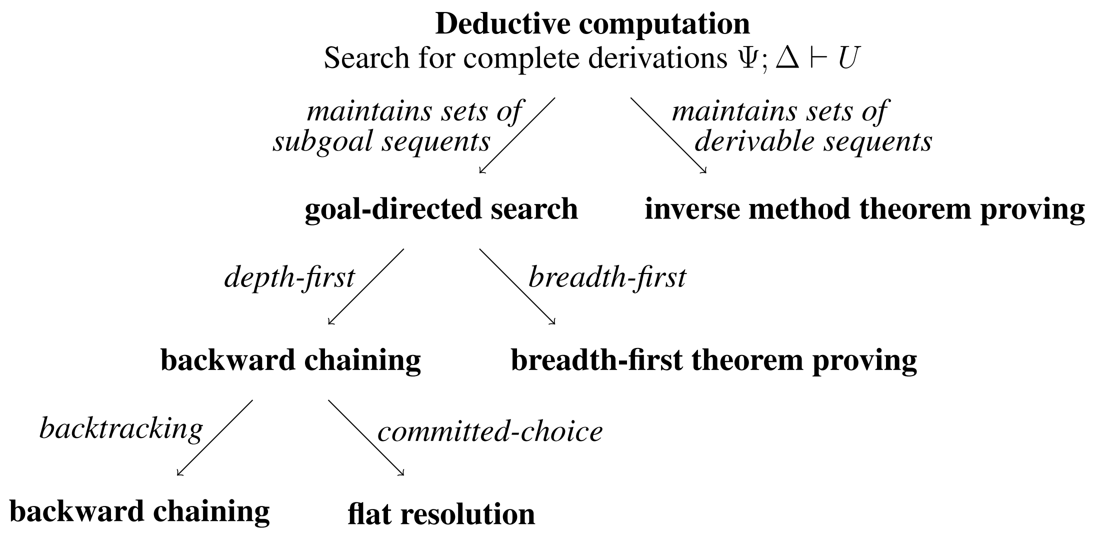

Deductive computation
Search for complete derivations $\Psi; \Delta \vdash U$
maintains sets of subgoal sequents
maintains sets of derivable sequents
goal-directed search
inverse method theorem proving
depth-first
breadth-first
backward chaining
breadth-first theorem proving
backtracking
committed-choice
backward chaining
flat resolution

Figure 4.17: A rough taxonomy of deductive computation

4.6.2 Concurrent computation

Concurrent computation is the search for *partial* proofs of sequents. As the name suggests, in SLS concurrent computation is associated with the search for partial proofs of the judgment $A^+ \, lax$, which correspond to traces $(\Psi; \Delta) \leadsto^* (\Psi'; \Delta')$.

The paradigm we will primarily associate with concurrent computation is *forward chaining*, which implies that we take an initial process state $(\Psi; \Delta)$ and allow it to evolve freely by the application of synthetic transitions. Additional conditions can be imposed on forward chaining: for instance, synthetic transitions like $(\Delta, x{:}\langle p^+_{pers}\rangle \, pers) \leadsto (\Delta, x{:}\langle p^+_{pers}\rangle \, pers, y{:}\langle p^+_{pers}\rangle \, pers)$ that do not meaningfully change the state can be excluded (if a persistent proposition already exists, two copies of that proposition don't add anything).[4] Forward chaining with this restriction in a purely-persistent logic is strongly associated with the Datalog language and its implementations; we will refer to forward chaining in persistent logics as *saturating logic programming* in Chapter 8. Forward chaining does not always deal with partially-unspecified terms; when persistent logic programming languages support forward chaining with partially-unspecified terms, it is called *hyperresolution* [FLHT01].

The presence of ephemeral or ordered resources in substructural logic means that a process state may evolve in multiple mutually-incompatible ways. *Committed choice* is a version of forward chaining that never goes back and reconsiders alternative evolutions from the initial state. Just as the default interpretation of backward chaining includes backtracking, we will consider the default interpretation of forward chaining to be committed choice, following [LPPW05]. An alternative interpretation would consider multiple evolutionary paths, which is a version of *exhaustive search*. Trace computation that works backwards from a final state instead of forward from an initial state can also be considered, and *planning* can be seen as specifying both the initial and final process states and trying to extrapolate a trace between them by working in both

[4]Incidentally, Lollimon implements this restriction and Celf, as of version 2.9, does not.

directions.

Outside of this work and Saurin's work on Ludics programming [Sau08], there is not much work on explicitly characterizing and searching for partial proofs in substructural logics.[5] Other forms of computation can be characterized as trace computation, however. Multiset rewriting and languages like GAMMA can be partially or completely understood in terms of forward chaining in linear logic [CS09, BG96], and the ordered aspects of SLS allow it to capture fragments of rewriting logic. Rewriting logic, and in particular the Maude implementation of rewriting logic [CDE+11], implements both the committed choice and the exhaustive search interpretations, as well as a *model checking* interpretation that characterize sets of process states or traces using logical formulas. Constraint handling rules [BRF10] and concurrent constraint programming [JNS05] are other logic programming models can be characterized as forms of concurrent computation.

4.6.3 Integrating deductive and trace computation

In the logic programming interpretation of CLF used by Lollimon and Celf, backtracking backward chaining is associated with the deductive fragment, and committed-choice forward chaining is associated with the lax modality. We will refer to an adaptation of the Lollimon/Celf semantics to SLS as "the Lollimon semantics" for brevity in this section.

Forward chaining and backward chaining have an uneasy relationship in the Lollimon semantics. Consider the following SLS signature:

$$\begin{aligned}\Sigma_{Demo} = \cdot,\ & \mathsf{posA} : \mathsf{prop\,ord},\ \ \mathsf{posB} : \mathsf{prop\,ord},\ \ \mathsf{posC} : \mathsf{prop\,ord},\ \ \mathsf{negD} : \mathsf{prop},\\ & \mathsf{fwdruleAB} : \mathsf{posA} \rightarrowtail \bigcirc\mathsf{posB},\\ & \mathsf{fwdruleAC} : \mathsf{posA} \rightarrowtail \bigcirc\mathsf{posC},\\ & \mathsf{bwdrule} : (\mathsf{posA} \rightarrowtail \bigcirc\mathsf{posB}) \rightarrowtail \mathsf{negD}\end{aligned}$$

In an empty context, there is only one derivation of negD under this signature: it is represented by the proof term $\mathsf{bwdrule}\,(\lambda x.\,\{\mathsf{let}\,\{y\} \leftarrow \mathsf{fwdruleAB}\,x\,\mathsf{in}\,y\})$. The partially complete interpretation of backward chaining stipulates that an interpreter tasked with finding a proof of negD should either find this proof or never terminate, but the Lollimon semantics only admits this interpretation for purely deductive proofs. To see why, consider backward-chaining search attempting to prove negD in a closed context. This can only be done with the rule bwdrule, generating the subgoal $\mathsf{posA} \rightarrowtail \bigcirc\mathsf{posB}$. At this point, the Lollimon semantics will switch from backward chaining to forward chaining and attempt to satisfy this subgoal by constructing a trace $(x{:}\langle\mathsf{posA}\rangle\ \mathit{ord}) \leadsto (y{:}\langle\mathsf{posB}\rangle\ \mathit{ord})$.

There are *two* nontrivial traces in this signature starting from the process state $(x{:}\langle\mathsf{posA}\rangle\ \mathit{ord})$ – the first is $(\{y\} \leftarrow \mathsf{fwdruleAB}\,x) :: (x{:}\langle\mathsf{posA}\rangle\ \mathit{ord}) \leadsto (y{:}\langle\mathsf{posB}\rangle\ \mathit{ord})$, and the second is $(\{y\} \leftarrow \mathsf{fwdruleAC}\,x) :: (x{:}\langle\mathsf{posA}\rangle\ \mathit{ord}) \leadsto (y{:}\langle\mathsf{posC}\rangle\ \mathit{ord})$. Forward chaining can plausibly come up with either one, and if it happens to derive the second one, the subgoal fails. Lollimon then tries to backtrack to find other rules that can prove the conclusion negD, but there are none, so the Lollimon semantics will report a failure to prove negD.

[5] As such, "concurrent computation," while appropriate for SLS, may or may not prove to be a good name for the general paradigm.

This example indicates that it is difficult to make backward chaining (in its default backtracking form) reliant on committed-choice forward chaining (in its default committed-choice form) in the Lollimon semantics. Either we can restrict forward chaining to confluent systems (excluding Σ_{Demo}) or else we can give up on the usual partially complete interpretation of backward chaining. In the other direction, however, it is entirely natural to make forward chaining dependent upon backward chaining. The fragment of CLF that encodes this kind of computation was labeled the *semantic effects* fragment by DeYoung [DP09]. At the logical level, the semantic effects fragment of SLS removes the right rule for $\bigcirc A^+$, which corresponds to the proof term $\{\mathsf{let}\, T\, \mathsf{in}\, V\}$. As discussed in Section 4.2.6, these let-expressions are the only point where traces are included into the language of deductive terms.

4.7 Design decisions

Aside from ordered propositions, there are several significant differences between the framework SLS presented in this chapter and the existing logical framework CLF, including the presence of positive atomic propositions, the introduction of traces as an explicit notation for partial proofs, the restriction of the term language to LF, and the presence of equality $t \doteq_\tau s$ as a proposition. In this section, we will discuss design choices that were made in terms of each of these features, their effects, and what choices could have been made differently.

4.7.1 Pseudo-positive atoms

Unlike SLS, the CLF framework does not include positive atomic propositions. Positive atomic propositions make it easy to characterize the synthetic transitions associated with a particular rule. For example, if foo, bar, and baz are all linear atomic propositions, then the presence of a rule somerule : $(\mathsf{foo} \bullet \mathsf{bar} \rightarrowtail \bigcirc \mathsf{baz})$ in the signature is associated with synthetic transitions of the form $(\Psi; \Delta, x{:}\langle \mathsf{foo}\rangle\ \mathit{eph}, y{:}\langle \mathsf{bar}\rangle\ \mathit{eph}) \leadsto (\Psi; \Delta, z{:}\langle \mathsf{baz}\rangle\ \mathit{eph})$. The presence of the rule somerule enables steps of this form, and every step made by focusing on the rule has this form.

CLF has no positive propositions, so the closest analogue that we can consider is where foo, bar, and baz are negative propositions, and the rule $¡\mathsf{foo} \bullet ¡\mathsf{bar} \rightarrowtail \bigcirc(¡\mathsf{baz})$ appears in the signature. Such a rule is associated with synthetic transitions of the form $(\Psi; \Delta, \Delta_1, \Delta_2) \leadsto (\Psi; \Delta, z{:}\mathsf{baz}\ \mathit{ord})$ such that $\Psi; \Delta_1{\restriction_{eph}} \vdash \langle \mathsf{foo}\rangle\ \mathit{true}$ and $\Psi; \Delta_2{\restriction_{eph}} \vdash \langle \mathsf{bar}\rangle\ \mathit{true}$. In SLS, it is a relatively simple syntactic criterion to enforce that a sequent like $\Psi; \Delta_1 \vdash \langle \mathsf{foo}\rangle\ \mathit{true}$ can only be derived if Δ_1 matches $x{:}\mathsf{foo}$; we must simply ensure that there are no propositions of the form $\ldots \rightarrowtail \mathsf{foo}$ or $\ldots \twoheadrightarrow \mathsf{foo}$ in the signature or context. (In fact, this is essentially the SLS version of the subordination criteria that allows us to conclude that an LF type is only inhabited by variables in Section 4.2.) Note that, in full OL_3, this task would not be so easy: we might prove $\langle \mathsf{foo}\rangle\ \mathit{true}$ indirectly by forward chaining. This is one reason why association of traces with the lax modality is so important!

When it is the case that $\Psi; \Delta_1 \vdash \langle \mathsf{foo}\rangle\ \mathit{true}$ can only be derived if Δ_1 matches $x{:}\mathsf{foo}$, we can associate the rule $¡\mathsf{foo} \bullet ¡\mathsf{bar} \rightarrowtail \bigcirc(\downarrow(¡\mathsf{baz}))$ with a unique synthetic transition of the form $(\Psi; \Delta, x{:}\mathsf{foo}\ \mathit{lvl}, y{:}\mathsf{bar}\ \mathit{lvl}') \leadsto (\Psi; \Delta, z{:}\langle \mathsf{baz}\rangle\ \mathit{eph})$ under the condition that neither lvl or lvl' are ord. Negative atomic propositions that can only be concluded when they are the sole member of the context, like foo and bar in this example, can be called *pseudo-positive*. Pseudo-positive atoms can actually be used a bit more generally than SLS's positive atomic propositions. A positive atomic proposition is necessarily associated with one of the three judgments ord, eph, or $pers$, but pseudo-positive propositions can associate with any of the contexts. This gives pseudo-positive atoms in CLF or SLS the flavor of positive atomic propositions under Andreoli's atom optimization (Section 2.5.1).

It is, of course, possible to consistently associate particular pseudo-positive propositions with particular modalities, which means that pseudo-positive propositions can subsume the positive propositions of SLS. The trade-off between positive and pseudo-positive propositions could be resolved either way. By including positive atomic propositions, we made SLS more complicated, but in a local way – we needed a few more kinds (the kinds prop ord, prop lin, and prop pers, to be precise) and a few more rules. On the other hand, if we used pseudo-positive propositions, the notion of synthetic transitions would be intertwined with the subordination-like analysis that enforces their correct usage.

4.7.2 The need for traces

One of the most important differences between SLS and its predecessors, especially CLF, is that traces are treated as first-class syntactic objects. This allows us to talk about partial proofs and thereby encode our earlier money-store-battery-robot example as a trace with this type:

$$\begin{aligned}&(x{:}\langle \mathsf{6bucks}\rangle\, eph,\;\; f{:}(\mathsf{battery} \rightarrowtail \bigcirc\mathsf{robot})\, eph,\;\; g{:}(\mathsf{6bucks} \rightarrowtail \bigcirc\mathsf{battery})\, pers)\\ \leadsto^{*}\, &(z{:}\langle \mathsf{robot}\rangle\, eph,\;\; g{:}(\mathsf{6bucks} \rightarrowtail \bigcirc\mathsf{battery})\, pers)\end{aligned}$$

It is also possible to translate the example from Chapter 2 as a *complete* proof of the following proposition:

$$\mathsf{6bucks} \bullet \text{¡}(\mathsf{battery} \rightarrowtail \bigcirc\mathsf{robot}) \bullet !(\mathsf{6bucks} \rightarrowtail \bigcirc\mathsf{battery}) \rightarrowtail \bigcirc\mathsf{robot}$$

Generally speaking, we can try to represent a trace $T :: (\Psi;\Delta) \leadsto^{*} (\Psi';\Delta')$ as a closed deductive proof $\lambda P.\, \{\mathsf{let}\, T \,\mathsf{in}\, V\}$ of the proposition $(\exists\Psi.\, \bullet\Delta) \rightarrowtail \bigcirc(\exists\Psi'.\, \bullet\Delta)$,[6] where the pattern P re-creates the initial process state $(\Psi;\Delta)$ and all the components of the final state are captured in the value V. The problem with this approach is that the final proposition is under no particular obligation to faithfully capture the structure of the final process state. This can be seen in the example above: to actually capture the structure of the final process state, we should have concluded $\mathsf{robot} \bullet !(\mathsf{6bucks} \rightarrowtail \bigcirc\mathsf{battery})$ instead of simply robot. It is also possible to conclude any of the following:

1. $\mathsf{robot} \bullet !(\mathsf{6bucks} \rightarrowtail \bigcirc\mathsf{battery}) \bullet !(\mathsf{6bucks} \rightarrowtail \bigcirc\mathsf{battery})$, or
2. $\mathsf{robot} \bullet {\downarrow}(\mathsf{6bucks} \rightarrowtail \bigcirc\mathsf{battery}) \bullet \text{¡}(\mathsf{6bucks} \rightarrowtail \bigcirc\mathsf{battery})$, or even
3. $\mathsf{robot} \bullet \text{¡}(\mathsf{6bucks} \bullet !(\mathsf{battery} \rightarrowtail \bigcirc\mathsf{robot}) \rightarrowtail \bigcirc\mathsf{robot}) \bullet {\downarrow}(\mathsf{robot} \rightarrowtail \bigcirc\mathsf{robot})$.

[6] The notation $\bullet\Delta$ fuses together all the propositions in the context. For example, if $\Delta = w{:}\langle p^{+}_{eph}\rangle\, eph \bullet x{:}A^{-}\, ord, y{:}B^{-}\, eph, z{:}C^{-}\, pers$, then $\bullet\Delta = p^{+}_{eph} \bullet {\downarrow}A^{-} \bullet \text{¡}B^{-} \bullet !C^{-}$. The notation $\exists\Psi.A^{+}$ turns all the bindings in the context $\Psi = a_1{:}\tau_1, \ldots, a_n{:}\tau_n$ into existential bindings $\exists a_1{:}\tau_1 \ldots \exists a_n{:}\tau_n.A^{+}$.

The problem with encoding traces as complete proofs, then, is that values cannot be forced to precisely capture the structure of contexts, especially when dealing with variables or persistent propositions. Cervesato and Scedrov approach this problem by severely restricting the logic and changing the interpretation of the existential quantifier so that it acts like a nominal quantifier on the right [CS09]. The introduction of traces allows us to avoid similar restrictions in SLS.

Despite traces being proper syntactic objects, they are not first-class concepts in the theory: they are derived from focused OL_3 terms and interpreted as partial proofs. Because hereditary substitution, identity expansion, and focalization are only defined on complete OL_3 proofs, these theorems and operations only apply by analogy to the deductive fragment of SLS; they do not apply to traces. In joint work with Deng and Cervesato, we considered a presentation of logic that treats process states and traces as first-class concepts and reformulates the usual properties of cut and identity in terms of coinductive simulation relations on process states [DCS12]. We hope that this work will eventually lead to a better understanding of traces, but the gap remains quite large.

4.7.3 LF as a term language

The decision to use LF as a first-order domain of quantification rather than using a fully-dependent system is based on several considerations. First and foremost, this choice was sufficient for our purposes here. In fact, for the purposes of this dissertation, we could have used an even simpler term language of simply-typed LF [Pfe08]. Two other logic programming interpreters for SLS-like frameworks, Lollimon [LPPW05] and Ollibot [PS09], are in fact based on simply-typed term languages. Canonical LF and Spine Form LF are, at this point, sufficiently well understood that the additional overhead of fully dependently-typed terms is not a significant burden, and there are many examples beyond the scope of this dissertation where dependent types are useful.

On a theoretical level, it is a significant simplification when we restrict ourselves to *any* typed term language with a reasonable notion of equality and simultaneous substitution. The conceptual priority in this chapter is clear: Section 4.1 describes LF terms, Section 4.2 describes proof terms as a fragment of focused OL_3, and Section 4.3 describes a coarser equivalence on proof terms, concurrent equality. If the domain of first-order of quantification was SLS terms, these three considerations would be mutually dependent – we would need to characterize concurrent equality before presenting the logic itself. For the purposes of showing that a logical framework can be carved out from a focused logic – the central thesis of this and the previous two chapters – it is easiest to break this circular dependency. We conjecture that this complication is no great obstacle, but our approach avoids the issue.

On a practical level, there are advantages to using a well-understood term language. The SLS prototype implementation (Section 4.5) uses the mature type reconstruction engine of Twelf to reconstruct LF terms. Schack-Nielsen's implementation of type reconstruction for Celf is complicated by the requirements of dealing with type reconstruction for a substructural term language, a completely orthogonal consideration [SNS08].

Finally, it is not clear that the addition of full CLF-like dependency comes with great expressive benefit. In LF and Twelf, the ability to use full dependent types is critical in part because it allows us to express *metatheorems* – theorems about the programming languages and logics we have encoded, like progress and preservation for a programming language or cut admissibility for a logic. Substructural logical frameworks like LLF and CLF, in contrast, have not been successful in capturing metatheorems with dependent types. Instead, metatheorems about substructural logics have thus far generally been performed in logical frameworks based on persistent logics. Crary proved theorems about linear logics and languages in LF using the technique of explicit contexts [Cra10]. Reed was able to prove cut admissibility for linear logic and preservation for the LLF encoding of Mini-ML in HLF, a persistent extension to LF that uses an equational theory to capture the structure of substructural contexts [Ree09].

Bibliography

[AK99] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, 1999. 4.6.1

[And01] Jean-Marc Andreoli. Focussing and proof construction. *Annals of Pure and Applied Logic*, 107:131–163, 2001. 2.4, 2.6, 3, 4.6.1

[AP01] Pablo Armelín and David Pym. Bunched logic programming. In *Proceedings International Joint Conference on Automated Reasoning (IJCAR'01)*, pages 289–304. Springer LNCS 2083, 2001. 4.6.1

[BG96] Paola Bruscoli and Alessio Guglielmi. A linear logic view of Gamma style computations as proof searches. In Jean-Marc Andreoli, Chris Hankin, and Daniel Le Métayer, editors, *Coordination programming: mechanisms, models and semantics*, pages 249–273. Imperial College Press, 1996. 4.6.2

[BRF10] Hariolf Betz, Frank Raiser, and Thom Frühwirth. A complete and terminating execution model for constraint handling rules. *Theory and Practice of Logic Programming*, 10(4–6):597–610, 2010. 4.6.2

[CDE$^+$11] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *Maude Manual (Version 2.6).* University of Illinois, Urbana-Champaign, 2011. 1.3, 4.6.2

[CHP00] Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. *Theoretical Computer Science*, 232(1–2):133–163, 2000. 4.6.1

[Cla87] Keith L. Clark. Negation as failure. In *Readings in nonmonotonic logic*, pages 311–325. Morgan Kaufmann Publishers, Inc., 1987. 4.6.1

[CMS09] Kaustuv Chaudhuri, Dale Miller, and Alexis Saurin. Canonical sequent proofs via multi-focusing. In *Fifth IFIP International Conference on Theoretical Computer Science (TCS 2008, Track B)*, pages 383–396, 2009. 4.3.1, 4.3.1

[CP02] Iliano Cervesato and Frank Pfenning. A linear logical framework. *Information and Computation*, 179(1):19–75, 2002. 2.1, 3.3.3, 4.1, 4.1.1, 4.1.3, 4.4, 5.2, 9.1.3, 10

[CPP08] Kaustuv Chaudhuri, Frank Pfenning, and Greg Price. A logical characterization of forward and backward chaining in the inverse method. *Journal of Automated Reasoning*, 40(2–3):133–177, 2008. 4.6.1

[CPS$^+$12] Iliano Cervesato, Frank Pfenning, Jorge Luis Sacchini, Carsten Schürmann, and

Robert J. Simmons. Trace matching in a concurrent logical framework. In *Proceedings of the International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'12)*, Copenhagen, Denmark, 2012. 4.3, 4.3

[CPWW02] 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. 1.1, 2.1, 4.4, 5, 5.1, 7, 7.2, 7.2.2, B.5

[Cra10] Karl Crary. Higher-order representation of substructural logics. In *International Conference on Functional Programming (ICFP'10)*, pages 131–142. ACM, 2010. 3.8, 4.7.3

[CS09] Iliano Cervesato and Andre Scedrov. Relating state-based and process-based concurrency through linear logic (full-version). *Information and Computation*, 207(10):1044–1077, 2009. 1, 1.1, 2.1, 3, 4.6.2, 4.7.2

[DCS12] Yuxing Deng, Iliano Cervesato, and Robert J. Simmons. Relating reasoning methodologies in linear logic and process algebra. In *Second International Workshop on Linearity (LINEARITY'12)*, Tallinn, Estonia, 2012. 4.7.2, A

[Die90] Volker Diekert. *Combinatorics on Traces*. Springer LNCS 454, 1990. 4.3

[DP09] 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, Los Angelas, CA, 2009. 4.6.3

[FLHT01] Christian Fermüller, Alexander Leitsch, Ullrich Hustadt, and Tanel Tammet. Resolution decision procedures. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 2, chapter 25, pages 1791–1849. Elsevier Science and MIT Press, 2001. 4.6.2

[FM97] Matt Fairtlough and Michael Mendler. Propositional lax logic. *Information and Computation*, 137(1):1–33, 1997. 2.5.3, 3.1, 4.2

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993. 1.2, 2.2, 4.1, 6.3

[HL07] Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, 17(4–5):613–673, 2007. 2.2, 4.1, 4.1.2, 4.1.3, 4.1.3, 6.3

[HM94] Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994. 4.6.1

[Hoa71] C. A. R. Hoare. Proof of a program: FIND. *Communications of the ACM*, 14(1):39–45, 1971. 4.2.7

[JNS05] Rada Jagadeesan, Gopalan Nadathur, and Vijay Saraswat. Testing concurrent systems: An interpretation of intuitionistic logic. In *Foundations of Software Technology and Theoretical Computer Science*, pages 517–528. Springer LNCS 3821, 2005. 4.3, 4.6.2

[LPPW05] Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In *Principles and Practice of Declarative Programming (PPDP'05)*, pages 35–46. ACM, 2005. 4.6, 4.6.2, 4.7.3, 8.1

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51(1–2):125–157, 1991. 4.6.1

[Mur89] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989. 4.3

[NPP08] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *Transactions on Computational Logic*, 9(3), 2008. 3.1.1, 4.1, 4.1.3

[Pfe08] Frank Pfenning. Church and Curry: Combining intrinsic and extrinsic typing. In C.Benzmüller, C.Brown, J.Siekmann, and R.Statman, editors, *Reasoning in Simple Type Theory: Festschrift in Honor of Peter B. Andrews on His 70th Birthday*, volume 17 of *Studies in Logic*. College Publications, 2008. 3.3.3, 4.1.2, 4.7.3

[Pfe12] Frank Pfenning. Lecture notes on backtracking, February 2012. Lecture notes for 15-819K: Logic Programming at Carnegie Mellon University, available online: http://www.cs.cmu.edu/~fp/courses/lp/lectures/lp-all.pdf. 3

[Pol00] Jeff Polakow. Linear logic programming with ordered contexts. In *Principles and Practice of Declarative Programming (PPDP'00)*, pages 68–79. ACM, 2000. 4.6.1

[Pol01] Jeff Polakow. *Ordered Linear Logic and Applications*. PhD thesis, Carnegie Mellon University, 2001. 3.1, 4.1, 4.4, 4.6.1, 9.1.3

[PS99] Frank Pfenning and Carsten Schürmann. Algorithms for equality and unification in the presence of notational definitions. In T. Altenkirch, B. Reus, and W. Naraschewski, editors, *Types for Proofs and Programs (TYPES'98)*, pages 179–193. Springer LNCS 1657, 1999. 4.2.1

[PS09] 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. 1.2, 2.1, 2.5, 2.5.2, 2.5.3, 3.6.1, 4.7.3, 5, 5.1, 6.5, 6.5.3, 7.2, 7.2.2, 10.2

[Ree09] Jason Reed. *A Hybrid Logical Framework*. PhD thesis, Carnegie Mellon University, 2009. 4.1, 4.1.2, 4.7.3, 5.2, 10

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual Symposium on Logic in Computer Science (LICS'02)*, pages 55–74, 2002. 4.2.8

[Sau08] Alexis Saurin. Towards ludics programming: Interactive proof search. In *Proceedings of the International Conference on Logic Programming (ICLP'08)*, pages 253–268. Springer LNCS 5366, 2008. 4.6.2

[SN07] Anders Schack-Nielsen. Induction on concurrent terms. In *Proceedings of the International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'07)*, pages 37–51, Bremen, Germany, 2007. 4.4, 5.1, 6.1

[SN11] Anders Schack-Nielsen. *Implementing Substructural Logical Frameworks*. PhD thesis, IT University of Copenhagen, 2011. 4, 4.6

[SNS08] Anders Schack-Nielsen and Carsten Schürmann. Celf — a logical framework for deductive and concurrent systems (system description). In *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR'08)*, pages 320–326. Springer LNCS 5195, 2008. 3.2, 4.6, 4.7.3, B

[Vir99] Roberto Virga. *Higher-Order Rewriting with Dependent Types*. PhD thesis, Carnegie Mellon University, 1999. 4.1.3

[WCPW02] 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. 1.1, 1.1, 1.4, 2.2, 3.1, 3.3.3, 3.4.2, 3.8, 4, 4.1, 4.3