Chapter 6

Ordered abstract machines

This chapter centers around two transformations on logical specifications. Taken together, the operationalization transformation (Section 6.1) and the defunctionalization transformation (Section 6.2) allow us to establish the logical correspondence between the deductive SLS specification of a natural semantics and the concurrent SLS specification of an abstract machine.

Natural semantics specifications are common in the literature, and are also easy to encode in either the deductive fragment of SLS or in a purely deductive logical framework like LF. We will continue to use the natural semantics specification of call-by-value evaluation for the lambda calculus as a running example:

$$\frac{}{\lambda x.e \Downarrow \lambda x.e}\ \mathsf{ev/lam} \qquad \frac{e_1 \Downarrow \lambda x.e \quad e_2 \Downarrow v_2 \quad [v_2/x]e \Downarrow v}{e_1\ e_2 \Downarrow v}\ \mathsf{ev/app}$$

Natural semantics are a *big-step* semantics: the judgment $e \Downarrow v$ describes the relationship between an initial expression e and the value v to which it will eventually evaluate.

The alternative to a big-step semantics is a *small-step* semantics, which describes the relationship between one intermediate state of a computation and another intermediate state after a single transition. One form of small-step semantics is a structural operational semantics (SOS) specification [Plo04]. The SOS specification of call-by-value evaluation for the lambda calculus is specified in terms of two judgments: $v\ \mathsf{value}$, which expresses that v is a value that is not expected to make any more transitions, and $e_1 \mapsto e_2$, which expresses that e_1 transitions to e_2 by reducing a β-redex.

$$\frac{}{\lambda x.e\ \mathsf{value}} \qquad \frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2} \qquad \frac{e_1\ \mathsf{value} \quad e_2 \mapsto e_2'}{e_1\ e_2 \mapsto e_1\ e_2'} \qquad \frac{v\ \mathsf{value}}{(\lambda x.e)v \mapsto [v/x]e}$$

Abstract machine semantics are another important small-step semantics style. The most well-known abstract semantics is almost certainly Landin's SECD machine [Lan64], though our abstract machine presentation below is much closer to Danvy's SC machine from [Dan03] and Harper's $\mathcal{K}\{\mathsf{nat}{\rightharpoonup}\}$ system from [Har12, Chapter 27]. This abstract machine semantics is defined in terms of states s. The state $s = k \rhd e$ represents the expression e being evaluated on top of the stack k, and the state $s = k \lhd v$ represents the value v being returned to the stack k. Stacks k are have the form $((\ldots(\mathsf{halt}; f_1); \ldots); f_n)$ – they are left-associative sequences of

frames f terminated by halt, where $\mathsf{halt} \rhd e$ is the initial state in the evaluation of e and $\mathsf{halt} \lhd v$ is a final state that has completed evaluating to a value v. Each frame f either has the form $\Box\, e_2$ (an application frame waiting for an evaluated function to be returned to it) or the form $(\lambda x.e)\,\Box$ (an application frame with an evaluated function waiting for an evaluated value to be returned to it). Given states, stacks, and frames, we can define a "classical" abstract machine for call-by-value evaluation of the lambda calculus as a transition system with four transition rules:

$$\begin{aligned}
\mathsf{absmachine/lam}&: \; k \rhd \lambda x.e \;\mapsto\; k \lhd \lambda x.e \\
\mathsf{absmachine/app}&: \; k \rhd e_1\, e_2 \;\mapsto\; (k; \Box\, e_2) \rhd e_1 \\
\mathsf{absmachine/app1}&: \; (k; \Box\, e_2) \lhd \lambda x.e \;\mapsto\; (k; (\lambda x.e)\,\Box) \rhd e_2 \\
\mathsf{absmachine/app2}&: \; (k; (\lambda x.e)\,\Box) \lhd v_2 \;\mapsto\; k \rhd [v_2/x]e
\end{aligned}$$

The operational intuition for these rules is precisely the same as the operational intuition for the rewriting rules given in Section 1.2. This is not coincidental: the SLS specification from the introduction adequately encodes the transition system $s \mapsto s'$ defined above, a point that we will make precise in Section 6.3. The SLS specification from the introduction is *also* the result of applying the operationalization and defunctionalization transformations to the SLS encoding of the natural semantics given above. Therefore, these two transformations combined with the adequacy arguments at either end constitute a logical correspondence between natural semantics and abstract machines.

As discussed in Section 5.3, it is interesting to put existing specification styles into logical correspondence, but that is not our main reason for investigating logical correspondence in the context of this thesis. Rather, we are primarily interested in exploring the set of programming language features that can be modularly integrated into a transformed SLS specification and that could *not* be integrated into a natural semantics specification in a modular fashion. In Section 6.5 we explore a selection of these features, including mutable storage, call-by-need evaluation, and recoverable failure.

6.1 Logical transformation: operationalization

The intuition behind operationalization is rather simple: we examine the structure of backward chaining and then specify that computational process as an SLS specification. Before presenting the general transformation, we will motivate this transformation using our natural semantics specification of call-by-value evaluation.

The definition of $e \Downarrow v$ is moded with e as an input and v as an output, so it is meaningful to talk about being given e and using deductive computation to search for a v such that $e \Downarrow v$ is derivable. Consider a recursive search procedure implementing this particular deductive computation:

- If $e = \lambda x.e'$, it is possible to derive $\lambda x.e' \Downarrow \lambda x.e'$ with the rule $\mathsf{ev/lam}$.
- If $e = e_1\, e_2$, attempt to derive $e_1\, e_2 \Downarrow v$ using the rule $\mathsf{ev/app}$ by doing the following:
  1. Search for a v_1 such that $e_1 \Downarrow v_1$ is derivable.
  2. Assess whether $v_1 = \lambda x.e'$ for some e'; fail if it is not.

3. Search for a v_2 such that $e_2 \Downarrow v_2$ is derivable.
4. Compute $e' = [v_2/x]e$
5. Search for a v such that $e' \Downarrow v$ is derivable.

The goal of the operationalization transformation is to represent this deductive computation as a specification in the concurrent fragment of SLS. (To be sure, "concurrent" will seem like a strange word to use at first, as the specifications we write in the concurrent fragment of SLS will be completely sequential until Section 6.1.4.) The first step in this process, representing the syntax of expressions as LF terms of type exp, was discussed in Section 4.1.4. The second step is to introduce two new ordered atomic propositions. The proposition $\mathsf{eval}\ulcorner e \urcorner$ is the starting point, indicating that we want to search for a v such that $e \Downarrow v$, and the proposition $\mathsf{retn}\ulcorner v \urcorner$ indicates the successful completion of this procedure. Therefore, searching for a v such that $e \Downarrow v$ is derivable will be analogous to building a trace $T :: x_e{:}\langle \mathsf{eval}\ulcorner e \urcorner\rangle \rightsquigarrow^* x_v{:}\langle \mathsf{retn}\ulcorner v \urcorner\rangle$.

Representing the first case is straightforward: if we are evaluating $\lambda x.e$, then we have succeeded and can return $\lambda x.e$. This is encoded as the following proposition:

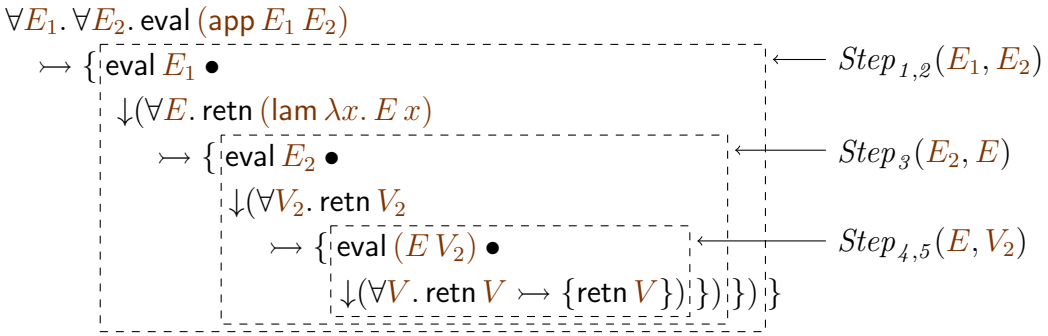
$$\forall E.\, \mathsf{eval}\ (\mathsf{lam}\ \lambda x.\, E\, x) \rightarrowtail \{\mathsf{retn}\ (\mathsf{lam}\ \lambda x.\, E\, x)\}$$

The natural deduction rule ev/app involves both recursion and multiple subgoals. The five steps in our informal search procedure are turned into three phases in SLS, corresponding to the three recursive calls to the search procedure – steps 1 and 2 are combined, as are steps 4 and 5. Whenever we make a recursive call to the search procedure, we leave a negative ordered proposition $A^-\ ord$ in the context that awaits the return of a proposition $\mathsf{retn}\ulcorner v' \urcorner$ to its left and then continues with the search procedure. Thus, each of the recursive calls to the search procedure will involve a sub-trace of the form

$$x_e{:}\langle \mathsf{eval}\ulcorner e' \urcorner\rangle, y{:}A^-\ ord, \Delta \rightsquigarrow^* x_v{:}\langle \mathsf{retn}\ulcorner v' \urcorner\rangle, y{:}A^-\ ord, \Delta$$

where A^- is a negative proposition that is prepared to interact with the subgoal's final $\mathsf{retn}\ulcorner v' \urcorner$ proposition to kickstart the rest of the computation. This negative proposition is, in effect, the calling procedure's continuation.

The nested rule for evaluating $e_1\ e_2$ to a value is the following proposition, where the three phases are indicated with dashed boxes:



Let's work backwards through this three-phase protocol. In the third phase, which corresponds to the fourth and fifth steps of our informal search procedure, we have found $\lambda x.\, E\, x = \lambda x.\ulcorner e \urcorner$

(where e potentially has x free) and $V_2 = \ulcorner v_2 \urcorner$. The recursive call is to $\mathsf{eval}\ulcorner[v_2/x]e\urcorner$, which is the same thing as $\mathsf{eval}\,(E\,V_2)$. If the recursive call successfully returns, the context will contain a suspended atomic proposition of the form $\mathsf{retn}\,V$ where $V = \ulcorner v \urcorner$, and the search procedure as a whole has been completed: the answer is v. Thus, the negative proposition that implements the continuation can be written as $(\forall V.\,\mathsf{retn}\,V \rightarrowtail \{\mathsf{retn}\,V\})$. (This continuation is the identity; we will show how to omit it when we discuss tail-recursion elimination in Section 6.1.3.) The positive proposition that will create this sub-computation can be written as follows:

$$\mathit{Step}_{4,5}(E, V_2) \equiv \mathsf{eval}\,(E\,V_2) \bullet \downarrow(\forall V.\,\mathsf{retn}\,V \rightarrowtail \{\mathsf{retn}\,V\})$$

Moving backwards, in the second phase (step 3 of the 5-step procedure) we have an expression $E_2 = \ulcorner e_2 \urcorner$ that we were given and $\lambda x.\,E\,x = \lambda x.\ulcorner e \urcorner$ that we have computed. The recursive call is to $\mathsf{eval}\ulcorner e_2 \urcorner$, and assuming that it completes, we need to begin the fourth step. The positive proposition that will create this sub-computation can be written as follows:

$$\mathit{Step}_3(E_2, E) \equiv \mathsf{eval}\,E_2 \bullet \downarrow(\forall V_2.\,\mathsf{retn}\,V_2 \rightarrowtail \{\mathit{Step}_{4,5}(E, V_2)\})$$

Finally, the first two steps, like the fourth and fifth steps, are handled together. We have $E_1 = \ulcorner e_1 \urcorner$ and $E_2 = \ulcorner e_2 \urcorner$; the recursive call is to $\mathsf{eval}\ulcorner e_1 \urcorner$. Once the recursive call completes, we enforce that the returned value has the form $\ulcorner \lambda x.e \urcorner$ before proceeding to the continuation.

$$\mathit{Step}_{1,2}(E_1, E_2) \equiv \mathsf{eval}\,E_1 \bullet \downarrow(\forall E.\,\mathsf{retn}\,(\mathsf{lam}\,\lambda x.\,E\,x) \rightarrowtail \{\mathit{Step}_3(E_2, E)\})$$

Thus, the rule implementing this entire portion of the search procedure is

$$\forall E_1.\,\forall E_2.\,\mathsf{eval}\,(\mathsf{app}\,E_1\,E_2) \rightarrowtail \{\mathit{Step}_{1,2}(E_1, E_2)\}$$

The SLS encoding of our example natural semantics is shown in Figure 6.1 alongside the transformed specification, which has the form of an ordered abstract machine semantics, though it is different than the ordered abstract machine semantics presented in the introduction. The specification in Figure 6.1 is *nested*, as ev/app is a rule that, when it participates in a transition, produces a new rule $(\forall E.\,\mathsf{retn}\,(\mathsf{lam}\,\lambda x.\,E\,x) \rightarrowtail \{\ldots\})$ that lives in the context. (In contrast, the ordered abstract machine semantics from the introduction was *flat*.) We discuss the defunctionalization transformation, which allows us to derive flat specifications from nested specifications, in Section 6.2 below.

The intuitive connection between natural semantics specifications and concurrent specifications has been explored previously and independently in the context of CLF by Schack-Nielsen [SN07] and by Cruz and Hou [CH12]; Schack-Nielsen proves the equivalence of the two specifications, whereas Cruz and Hou used the connection informally. The contribution of this section is to describe a general transformation (of which Figure 6.1 is one instance) and to prove the transformation correct in general. We have implemented the operationalization and defunctionalization transformations within the prototype implementation of SLS.

In Section 6.1.1 we will present the subset of specifications that our operationalization transformation handles, and in Section 6.1.2 we present the most basic form of the transformation. In Sections 6.1.3 and 6.1.4 we extend the basic transformation to be both tail-recursion optimizing and parallelism-enabling. Finally, in Section 6.1.5, we establish the correctness of the overall transformation.

```
#mode ev + -.
ev: exp -> exp -> prop.

ev/lam:
ev (lam \x. E x)
    (lam \x. E x).

ev/app:
ev (app E1 E2) V

 <- ev E1 (lam \x. E x)

 <- ev E2 V2

 <- ev (E V2) V.
```

```
eval: exp -> prop ord.
retn: exp -> prop ord.

ev/lam:
eval (lam \x. E x)
 >-> {retn (lam \x. E x)}.

ev/app:
eval (app E1 E2)
 >-> {eval E1 *
      (All E. retn (lam \x. E x)
        >-> {eval E2 *
             (All V2. retn V2
               >-> {eval (E V2) *
                    (All V. retn V
                      >-> {retn V})})})}.
```

Figure 6.1: Natural semantics (left) and ordered abstract machine (right) for CBV evaluation

6.1.1 Transformable signatures

The starting point for the operationalization transformation is a deductive signature that is well-moded in the sense described in Section 4.6.1. Every declared negative predicate will either remain defined by deductive proofs (we write the atomic propositions built with these predicates as p_d^-, d for deductive) or will be transformed into the concurrent fragment of SLS (we write these predicates as $\mathsf{a_c}$, $\mathsf{b_c}$ etc. and write the atomic propositions built with these predicates as p_c^-, c for concurrent).

For the purposes of describing and proving the correctness of the operationalization transformation, we will assume that all transformed atomic propositions p_c^- have two arguments where the first argument is moded as an input and the second is an output. That is, they are declared as follows:

$$\#\mathsf{mode}\ \mathsf{a_c}\ +\ -.$$
$$\mathsf{a_c} : \tau_1 \to \tau_2 \to \mathsf{prop}.$$

Without dependency, two-place relations are sufficient for describing n-place relations.[1] It should be possible to handle dependent predicates (that is, those with declarations of the form $\mathsf{a_c} : \Pi x{:}\tau_1.\, \tau_2(x) \to \mathsf{type}$), but we will not do so here.

The restriction on signatures furthermore enforces that all rules must be of the form $\mathsf{r} : C$ or $\mathsf{r} : D$, where C and D are refinements of the negative propositions of SLS that are defined as

[1] As an example, consider addition defined as a three-place relation $\mathsf{add}\ M\ N\ P$ (where add has kind $\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat} \to \mathsf{prop}$) with the usual mode ($\mathsf{add}\ +\ +\ -$). We can instead use a two-place relation $\mathsf{add}'\ (\mathsf{add\_inputs}\ M\ N)\ P$ with mode ($\mathsf{add}'\ +\ -$). The kind of add' is $\mathsf{add\_in} \to \mathsf{nat} \to \mathsf{type}$, where add_in is a new type with only one constructor $\mathsf{add\_inputs} : \mathsf{nat} \to \mathsf{nat} \to \mathsf{add\_in}$ that effectively pairs together the two natural numbers that are inputs.

follows:

$$C ::= p_c^- \mid \forall x{:}\tau.\, C \mid p_{pers}^+ \rightarrowtail C \mid !p_c^- \rightarrowtail C \mid !G \rightarrowtail C$$
$$D ::= p_d^- \mid \forall x{:}\tau.\, D \mid p_{pers}^+ \rightarrowtail D \mid !p_c^- \rightarrowtail D \mid !G \rightarrowtail C$$
$$G ::= p_d^- \mid \forall x{:}\tau.\, G \mid p_{pers}^+ \rightarrowtail G \mid !D \rightarrowtail G$$

For most of this chapter, we will restrict our attention to signatures where all atomic propositions have the form p_c^- and where all rules have the form C. This makes the classes p_d^-, D, and G irrelevant and effectively restricts rules to the Horn fragment. Propositions p_d^- that remain deductively defined by rules D will only be considered towards the end of this chapter in Section 6.6.2 when we consider various transformations on SOS specifications and in Section 6.6.3 when we consider transforming the natural semantics for Davies' $\lambda^{\bigcirc}$.

Note, however, that if a signature is well-formed given that a certain atomic proposition is assigned to the class p_c^- of transformed atomic propositions, the signature will *remain* well-formed if we instead assign that proposition to the class p_d^- of atomic propositions that get left in the deductive fragment. The only effect is that some rules that were previously of the form $\mathsf{r} : C$ will become rules of the form $\mathsf{r} : D$.[2] If we turn this dial all the way, we won't operationalize anything! If *all* atomic propositions are of the form p_d^- so that they remain deductive, then the propositions p_c^- and C are irrelevant, and the restriction above describes all persistent, deductive specifications – essentially, any signature that could be executed by the standard logic programming interpretation of LF [Pfe89]. The operationalization transformation will be the identity on such a specification.

All propositions C are furthermore equivalent (at the level of synthetic inference rules) to propositions of the form $\forall \overline{x_0} \ldots \forall \overline{x_n}.\, A_n^+ \rightarrowtail \ldots \rightarrowtail A_1^+ \rightarrowtail \mathsf{a_c}\, t_0\, t_{n+1}$, where the $\forall \overline{x_i}$ are shorthand for a series of universal quantifiers $\forall x_{i1}{:}\tau_{i1} \ldots \forall x_{ik}{:}\tau_{ik}$, and where each variable in $\overline{x_i}$ does not appear in t_0 (unless $i = 0$) nor in any A_j^+ with $j < i$ but does appear in A_i^+ (or t_0 if $i = 0$). Therefore, when we consider moded proof search, the variables bound in $\overline{x_0}$ are all fixed by the query and those bound in the other $\overline{x_i}$ are all fixed by the output position of the i^{th} subgoal.

Each premise A_i^+ either has the form p_{pers}^+, $!p_c^-$, or $!G$. The natural deduction rule $\mathsf{ev/app}$, which has three premises, can be represented in this standard form as follows:

$$\begin{array}{llll} \forall \overline{x_0}. & \forall \overline{x_1}.\forall \overline{x_2}.\forall \overline{x_3}.\, A_3^+ \rightarrowtail & A_2^+ \rightarrowtail & A_1^+ \rightarrowtail \quad \mathsf{a_c}\, t_0 \quad t_4 \\ \forall E_1.\, \forall E_2.\, \forall E.\, \forall V_2.\, \forall V.\, !(\mathsf{ev}\, (E\, V_2)\, V) & \rightarrowtail !(\mathsf{ev}\, E_2\, V_2) & \rightarrowtail !(\mathsf{ev}\, E_1\, (\mathsf{lam}\, \lambda x.\, E\, x)) & \rightarrowtail \mathsf{ev}\, (\mathsf{app}\, E_1\, E_2)\, V \end{array}$$

From here on out we will assume without loss of generality that any proposition C actually has this very specific form.

6.1.2 Basic transformation

The operationalization transformation $Op(\Sigma)$ operates on SLS signatures Σ that have the form described in the previous section. We will first give the transformation on signatures; the transformation of rule declarations $\mathsf{r} : C$ is the key case.

[2] The reverse does not hold: the proposition $!(\forall x{:}\tau.\, p_{pers}^+ \rightarrowtail p_d^-) \rightarrowtail p_d^-$ has the form D, but the proposition $!(\forall x{:}\tau.\, p_{pers}^+ \rightarrowtail p_c^-) \rightarrowtail p_c^-$ does *not* have the form C.

Each two-place predicate $\mathsf{a_c}$ gets associated with two one-place predicates eval_a and retn_a: both $\mathsf{eval\_a}\, t$ and $\mathsf{retn\_a}\, t$ are positive ordered atomic propositions. We will write $X^\dagger$ for the operation of substituting all occurrences of $p_c^- = \mathsf{a_c}\, t_1\, t_2$ with $(\mathsf{eval\_a}\, t_1 \rightarrowtail \{\mathsf{retn\_a}\, t_2\})$ in X. This substitution operation is used on propositions and contexts; it appears in the transformation of rules $\mathsf{r} : D$ below.

* $Op(\cdot) = \cdot$
* $Op(\Sigma, \mathsf{a_c} : \tau_1 \to \tau_2 \to \mathsf{prop}) = Op(\Sigma),\ \mathsf{eval\_a} : \tau_1 \to \mathsf{prop\ ord},\ \mathsf{retn\_a} : \tau_2 \to \mathsf{prop\ ord}$
* $Op(\Sigma, \mathsf{a_c} : \tau_1 \to \tau_2 \to \mathsf{prop}) = Op(\Sigma),\ \mathsf{eval\_a} : \tau_1 \to \mathsf{prop\ ord}$
  (if retn_a *is already defined)*
* $Op(\Sigma, \mathsf{b} : \kappa) = Op(\Sigma),\ \mathsf{b} : \kappa$ *(if* $\mathsf{b} \neq \mathsf{a_c}$*)*
* $Op(\Sigma, \mathsf{c} : \tau) = Op(\Sigma),\ \mathsf{c} : \tau$
* $Op(\Sigma, \mathsf{r} : C) = Op(\Sigma),\ \mathsf{r} : \forall \overline{x_0}.\, \mathsf{eval\_a}\, t_0 \rightarrowtail [\![A_1^+, \ldots, A_n^+]\!](\mathsf{a}, t_{n+1}, \mathsf{id})$
  (where $C = \forall \overline{x_0} \ldots \forall \overline{x_n}.\, A_n^+ \rightarrowtail \ldots \rightarrowtail A_1^+ \rightarrowtail \mathsf{a_c}\, t_0\, t_{n+1}$*)*
* $Op(\Sigma, \mathsf{r} : D) = Op(\Sigma),\ \mathsf{r} : D^\dagger$

The transformation of a proposition $C = \forall \overline{x_0} \ldots \forall \overline{x_n}.\, A_n^+ \rightarrowtail \ldots \rightarrowtail A_1^+ \rightarrowtail \mathsf{a_c}\, t_0\, t_{n+1}$ involves the definition $[\![A_i^+, \ldots, A_n^+]\!](\mathsf{a}, t_{n+1}, \sigma)$, where σ substitutes only for variables in $\overline{x_j}$ where $j < i$. The function is defined inductively on the length of the sequence $A_i^+, \ldots, A_n^+$.

* $[\![]\!](\mathsf{a}, t_{n+1}, \sigma) = \{\mathsf{retn\_a}\, (\sigma t_{n+1})\}$
* $[\![p_{pers}^+, A_{i+1}^+, \ldots, A_n^+]\!](\mathsf{a}, t_{n+1}, \sigma) = \forall \overline{x_i}.\, (\sigma p_{pers}^+) \rightarrowtail [\![A_{i+1}^+, \ldots, A_n^+]\!](\mathsf{a}, t_{n+1}, \sigma)$
* $[\![!p_c^-, A_{i+1}^+, \ldots, A_n^+]\!](\mathsf{a}, t_{n+1}, \sigma)$
  $= \{\mathsf{eval\_b}\, (\sigma t_i^{in}) \bullet (\forall \overline{x_i}.\, \mathsf{retn\_b}\, (\sigma t_i^{out}) \rightarrowtail [\![A_{i+1}^+, \ldots, A_n^+]\!](\mathsf{a}, t_{n+1}, \sigma))\}$
  (where p_c^- *is* $\mathsf{b_c}\, t_i^{in}\, t_i^{out}$*)*
* $[\![!G, A_{i+1}^+, \ldots, A_n^+]\!](\mathsf{a}, t_{n+1}, \sigma) = \forall \overline{x_i}.\, !(\sigma G^\dagger) \rightarrowtail [\![A_{i+1}^+, \ldots, A_n^+]\!](\mathsf{a}, t_{n+1}, \sigma)$

This operation is slightly more general than it needs to be to describe the transformation on signatures, where the substitution σ will always just be the identity substitution id. Non-identity substitutions arise during the proof of correctness, which is why we introduce them here.

Figure 6.1, relating the natural semantics to the encoding of the search procedure as an ordered abstract machine, is an instance of this transformation.

6.1.3 Tail-recursion

Consider again our motivating example, the procedure for that takes expressions e and searches for expressions v such that $e \Downarrow v$ is derivable. If we were to implement that procedure as a functional program, the procedure would be *tail-recursive*. In the procedure that handles the case when $e = e_1\, e_2$, the last step invokes the search procedure recursively. If and when that callee returns v, then the caller will also return v.

Tail-recursion is significant in functional programming because tail-recursive calls can be implemented without allocating a stack frame: when a compiler makes this more efficient choice, we say it is performing *tail-recursion optimization*.[3] An analogous opportunity for tail-recursion

[3] Or *tail-call optimization*, as a tail-recursive function call is just a specific instance of a tail call.

optimization arises in our logical compilation procedure. In our motivating example, the last step in the $e_1\ e_2$ case was operationalized as a positive proposition of the form $\mathsf{eval}\ (E\ V_2) \bullet (\forall V.\ \mathsf{retn}\ V \rightarrowtail \{\mathsf{retn}\ V\})$. In a successful search, the process state

$$x{:}\mathsf{eval}\ (E\ V_2), y{:}(\forall V.\ \mathsf{retn}\ V \rightarrowtail \{\mathsf{retn}\ V\})\ ord, \Delta$$

will evolve until the state

$$x'{:}\mathsf{retn}\ V', y{:}(\forall V.\ \mathsf{retn}\ V \rightarrowtail \{\mathsf{retn}\ V\})\ ord, \Delta$$

is reached, at which point the next step, focusing on y, takes us to the the process state

$$y'{:}\mathsf{retn}\ V', \Delta$$

If we operationalize the last step in the $e_1\ e_2$ case as $\mathsf{eval}\ (E\ V_2)$ instead of as $\mathsf{eval}\ (E\ V_2) \bullet (\forall V.\ \mathsf{retn}\ V \rightarrowtail \{\mathsf{retn}\ V\})$, we will reach the same final state with one fewer transition. The tail-recursion optimizing version of the operationalization transformation creates concurrent computations that avoid these useless steps.

We cannot perform tail recursion in general because the output of the last subgoal may be different from the output of the goal. For example, the rule $\mathsf{r} : \forall X.\ \forall Y.\ !\mathsf{a}\ X\ Y \rightarrowtail \mathsf{a}\ (\mathsf{c}\ X)\ (\mathsf{c}\ Y)$ will translate to

$$\mathsf{r} : \forall X.\ \mathsf{eval\_a}\ (\mathsf{c}\ X) \rightarrowtail \{\mathsf{eval\_a}\ X \bullet (\forall Y.\ \mathsf{retn\_a}\ Y \rightarrowtail \{\mathsf{retn\_a}\ (\mathsf{c}\ Y)\})\}$$

There is no opportunity for tail-recursion optimization, because the output of the last search procedure, $t_n^{out} = Y$, is different than the value returned down the stack, $t_{n+1} = \mathsf{c}\ Y$. This case corresponds to functional programs that cannot be tail-call optimized.

More subtly, we cannot even eliminate all cases where $t_n^{out} = t_{n+1}$ unless these terms are *fully general*. The rule $\mathsf{r} : \forall X.\ !\mathsf{test}\ X\ \mathsf{true} \rightarrowtail \mathsf{test}\ \mathsf{s}\ X\ \mathsf{true}$, for example, will translate to

$$\mathsf{r} : \forall X.\ \mathsf{eval\_a}\ \mathsf{s}\ X \rightarrowtail \{\mathsf{eval\_a}\ X \bullet (\mathsf{retn\_a}\ \mathsf{true} \rightarrowtail \mathsf{retn\_a}\ \mathsf{true})\}$$

It would invalid to tail-call optimize in this situation. Even though the proposition $\mathsf{retn\_a}\ \mathsf{true} \rightarrowtail \mathsf{retn\_a}\ \mathsf{true}$ is an identity, if the proposition $\mathsf{retn\_a}\ \mathsf{false}$ appears to its left, the process state will be unable to make a transition. This condition doesn't have an analogue in functional programming, because it corresponds to the possibility that moded deductive computation can perform pattern matching on *outputs* and fail if the pattern match fails.

We say that t_{n+1} with type τ is fully general if all of its free variables are in $\overline{x_n}$ (and therefore not fixed by the input of any other subgoal) and if, for any variable-free term t' of type τ, there exists a substitution σ such that $t = \sigma t_{n+1}$. The simplest way to ensure this is to require that $t_{n+1} = t_n^{out} = y$ where $y = \overline{x_n}$.[4]

The tail-recursive procedure can be described by adding a new case to the definition of $[\![A_i^+, \ldots, A_n^+]\!](\mathsf{a}, t_{n+1}, \sigma)$:

[4]It is also possible to have a fully general $t_{n+1} = \mathsf{c}\ y_1\ y_2$ if, for instance, c has type $\tau_1 \rightarrow \tau_2 \rightarrow \mathsf{foo}$ and there are no other constructors of type foo. However, we also have to check that there are no other first-order variables in Ψ with types like $\tau_3 \rightarrow \mathsf{foo}$ that could be used to make other terms of type foo.

```
ev/lam: eval (lam \x. E x) >-> {retn (lam \x. E x)}.

ev/app: eval (app E1 E2)
         >-> {eval E1 *
              (All E. retn (lam \x. E x)
                >-> {eval E2 *
                     (All V2. retn V2 >-> {eval (E V2)})})}.
```

Figure 6.2: Tail-recursion optimized semantics for CBV evaluation

- $*$ *...(four other cases from Section 6.1.2)* ...
- $*$ $[\![!\mathsf{b}\, t_n^{in}\, t_{n+1}]\!](\mathsf{a}, t_{n+1}, \sigma) = \{\mathsf{eval\_a}\ (\sigma t_n^{in})\}$
  (where t_{n+1} is fully general and $\mathsf{retn\_a} = \mathsf{retn\_b}$*)*

This case overlaps with the third case of the definition given in Section 6.1.2, which indicates that tail-recursion optimization can be applied or not in a nondeterministic manner.

Operationalizing the natural semantics from 6.1 with tail-recursion optimization gives us the ordered abstract machine in Figure 6.2.

6.1.4 Parallelism

Both the basic and the tail-recursive transformations are sequential: if $x{:}\mathsf{eval}\ulcorner e \urcorner \rightsquigarrow^* \Delta$, then the process state Δ contains at most one proposition $\mathsf{eval}\ulcorner e' \urcorner$ or $\mathsf{retn}\ulcorner v \urcorner$ that can potentially be a part of any further transition. Put differently, the first two versions of the operationalization transformation express deductive computation as a concurrent computation that does not exhibit any parallelism or concurrency (sequential computation being a special case of concurrent computation).

Sometimes, this is what we want: in Section 6.3 we will see that the sequential tail-recursion-optimized abstract machine adequately represents a traditional on-paper abstract machine for the call-by-value lambda calculus. In general, however, when distinct subgoals do not have input-output dependencies (that is, when none of subgoal i's outputs are inputs to subgoal $i+1$), deductive computation can search for subgoal i and $i+1$ simultaneously, and this can be represented in the operationalization transformation.

Parallelism will change the way we think about the structure of the ordered context: previously we were encoding a stack-like structure in the ordered context, and now we will encode a tree-like structure in the ordered context. It's really easy to encode a stack in an ordered context, as we have seen: we just write down the stack! Trees are only a little bit more complicated: we encode them in an ordered context by writing down an ordered tree traversal. Our translation uses a postfix traversal, so it is always possible to reconstruct a tree from the ordered context for the same reason that a postfix notations like Reverse Polish notation are unambiguous: there's always only one way to reconstruct the tree of subgoals.

In the previous transformations, our process states were structured such that every negative proposition A^- was waiting on a single retn to be computed to its left; at that point, the negative proposition could be focused upon, invoking the continuation stored in that negative proposition.

```
eval: exp -> prop ord.
retn: exp -> prop ord.

ev/lam: eval (lam \x. E x) >-> {retn (lam \x. E x)}.

ev/app: eval (app E1 E2)
         >-> {eval E1 * eval E2 *
               (All E. All V2. retn (lam \x. E x) * retn V2
                 >-> {eval (E V2)})}.
```

Figure 6.3: Parallel, tail-recursion optimized semantics for CBV evaluation

If we ignore the first-order structure of the concurrent computation, these intermediate states look like this:

$$\textit{(...subgoal 1...)} \qquad y{:}(\mathsf{retn} \rightarrowtail \mathit{cont})\ \mathit{ord}$$

Note that *subgoal 1* is intended to represent some nonempty sequence of ordered propositions, not a single proposition. With the parallelism-enabling transformation, subgoal 1 will be able to perform parallel search for its own subgoals:

$$\textit{(subgoal 1.1)} \qquad \textit{(subgoal 1.2)} \qquad y_1{:}(\mathsf{retn}_{1.1} \bullet \mathsf{retn}_{1.2} \rightarrowtail \mathit{cont}_1)\ \mathit{ord}, \qquad y{:}(\mathsf{retn} \rightarrowtail \mathit{cont})\ \mathit{ord}$$

The two subcomputations *(subgoal 1.1)* and *(subgoal 1.2)* are next to one another in the ordered context, but the postfix structure imposed on the process state ensures that the only way they can interact is if they both finish (becoming $z_{1.1}{:}\langle\mathsf{retn}_{1.1}\rangle$ and $z_{1.2}{:}\langle\mathsf{retn}_{1.2}\rangle$, respectively), which will allow us to focus on y_1 and begin working on the continuation cont_1.

To allow the transformed programs to enable parallel evaluation, we again add a new case to the function that transforms propositions C. The new case picks out $j-i$ premises $A_i, \ldots, A_j = !p^-_{ci}, \ldots, !p^-_{cj}$, requiring that those $j-i$ premises are *independent*. Each $p^-_{ck} = \mathsf{bk_c}\ t^{in}_k\ t^{out}_k$, where the term t^{out}_k is what determines the assignments for the variables in $\overline{x_k}$ when we perform moded proof search. Independence between premises requires that the free variables of t^{in}_k cannot include any variables in $\overline{x_m}$ for $i \leq m < k$; the well-modedness of the rule already ensures that t^{in}_k does not contain any variables in $\overline{x_m}$ for $k \leq m \leq j$.

- $*$ *...(four other cases from Section 6.1.2, one other case from Section 6.1.3)...*
- $*$ $[\![!p^-_{ci}, \ldots, !p^-_{cj}, A^+_{j+1}, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma)$
  $= \{\mathsf{eval\_bi}\ (\sigma t^{in}_i) \bullet \ldots \bullet \mathsf{eval\_bj}\ (\sigma t^{in}_j) \bullet$
  $(\forall \overline{x_i} \ldots \forall \overline{x_j}.\ \mathsf{retn\_bi}\ (\sigma t^{out}_i) \bullet \ldots \bullet \mathsf{retn\_bj}\ (\sigma t^{out}_j)$
  $\rightarrowtail [\![A^+_{j+1}, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma))\}$

  (where p^-_{ck} is $\mathsf{bk_c}\ t^{in}_k\ t^{out}_k$ and $\emptyset = FV(t^{in}_k) \cap (\overline{x_i} \cup \ldots \cup \overline{x_j})$ for $i \leq k \leq j$)

This new case subsumes the old case that dealt with sequences of the form $!p^-_c, A^+_{i+1}, \ldots, A^+_n$; that old case is now an instance of the general case where $i = j$. Specifically, the second side condition on the free variables, which is necessary if the resulting rule is to be well-scoped, is trivially satisfied in the sequential case where $i = j$.

The result of running the natural semantics from Figure 6.1 through the parallel and tail-recursion optimizing ordered abstract machine is shown in Figure 6.3; it shows that we can

search for the subgoals $e_1 \Downarrow \lambda x.e$ and $e_2 \Downarrow v_2$ in parallel. We cannot run either of these subgoals in parallel with the third subgoal $[v_2/x]e \Downarrow v$ because the input $[v_2/x]e$ mentions the outputs of both of the previous subgoals.

6.1.5 Correctness

We have presented, in three steps, a nondeterministic transformation. One reason for presenting a nondeterministic transformation is that the user can control this nondeterminism to operationalize with or without parallelism and with or without tail-call optimization. (The transformation as implemented in the SLS prototype only has one setting: it optimizes tail-calls but does not enable parallel evaluation.) The other reason for presenting a nondeterministic transformation is that we can prove the correctness of all the variants we have presented so far in one fell swoop by proving the correctness of the nondeterministic transformation.

Correctness is fundamentally the property that we have $\Psi; \Gamma \vdash_\Sigma \langle p_d^- \rangle$ if and only if we have $\Psi; \Gamma^\dagger \vdash_{Op(\Sigma)} \langle p_d^- \rangle$ and that we have $\Psi; \Gamma \vdash_\Sigma \langle \mathsf{a_c}\, t_1\, t_2 \rangle$ if and only if we have a trace $(\Psi; \Gamma, \mathsf{eval\_a}\, t_1) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Gamma, \mathsf{retn}\,(\mathsf{retn\_a}\, t_2))$. We label the forward direction "completeness" and the backward direction "soundness," but directional assignment is (as usual) somewhat arbitrary. Completeness is a corollary of Theorem 6.2, and soundness is a corollary of Theorem 6.4. We use Theorem 6.1 pervasively and usually without mention.

Theorem 6.1 (No effect on the LF fragment)**.** $\Psi \vdash_\Sigma t : \tau$ *if and only if* $\Psi \vdash_{Op(\Sigma)} t : \tau$.

Proof. Straightforward induction in both directions; the transformation leaves the LF-relevant part of the signature unchanged. □

Completeness

Theorem 6.2 (Completeness of operationalization)**.** *If all propositions in* Γ *have the form* $x{:}D$ *pers or* $z{:}\langle p^+_{pers} \rangle$*, then*

1. *If* $\Psi; \Gamma \vdash_\Sigma \langle p_d^- \rangle$*, then* $\Psi; \Gamma^\dagger \vdash_{Op(\Sigma)} \langle p_d^- \rangle$.
2. *If* $\Psi; \Gamma, [D] \vdash_\Sigma \langle p_d^- \rangle$*, then* $\Psi; \Gamma^\dagger, [D^\dagger] \vdash_{Op(\Sigma)} \langle p_d^- \rangle$.
3. *If* $\Psi; \Gamma \vdash_\Sigma G$*, then* $\Psi; \Gamma^\dagger \vdash_{Op(\Sigma)} G^\dagger$.
4. *If* Δ *matches* $\Theta\{\!\{\Gamma\}\!\}$ *and* $\Psi; \Gamma \vdash_\Sigma \langle p_c^- \rangle$ *(where* $p_c^- = \mathsf{a_c}\, t\, s$*),*
   then $(\Psi; \Theta^\dagger\{x{:}\langle \mathsf{eval\_a}\, t \rangle\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Theta^\dagger\{y{:}\langle \mathsf{retn\_a}\, s \rangle\})$.

Proof. Mutual induction on the structure of the input derivation.

The first three parts are straightforward. In part 1, we have $\Psi; \Gamma \vdash_\Sigma h \cdot Sp : \langle p_d^- \rangle$ where either $h = x$ and $x{:}D \in \Gamma$ or else $h = \mathsf{r}$ and $\mathsf{r} : D \in \Sigma$. In either case the necessary result is $h \cdot Sp'$, where we get Sp' from the induction hypothesis (part 2) on Sp.

In part 2, we proceed by case analysis on the proposition D in focus. The only interesting case is where $D = !p_c^- \rightarrowtail D'$

- If $D = p_d^-$, then $Sp = \textsc{nil}$ and $\textsc{nil}$ gives the desired result.

* If $D = \forall x{:}\tau.\, D'$ or $D = p^+_{pers} \rightarrowtail D'$, then $Sp = (t; Sp')$ or $Sp = (z; Sp')$ (respectively). The necessary result is $(t; Sp'')$ or $(z; Sp'')$ (respectively) where we get Sp'' from the induction hypothesis (part 2) on Sp'.
* If $D = !p^-_c \rightarrowtail D'$ and $p^-_c = \mathsf{a_c}\, t_1\, t_2$, then $Sp = (!N; Sp')$ and $D^\dagger = !(\mathsf{eval\_a}\, t_1 \rightarrowtail \bigcirc(\mathsf{retn\_a}\, t_2) \rightarrowtail D'^\dagger)$.[5]

  $\Psi; \Gamma \vdash_\Sigma N : \langle \mathsf{a_c}\, t_1\, t_2 \rangle$ (given)
  $\Psi; \Gamma, [D] \vdash_\Sigma Sp' : \langle p^-_d \rangle$ (given)
  $T :: (\Psi; \Gamma^\dagger, x{:}\langle \mathsf{eval\_a}\, t_1 \rangle) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Gamma^\dagger, y{:}\langle \mathsf{retn\_a}\, t_2 \rangle)$ (ind. hyp. (part 4) on N)
  $\Psi; \Gamma, [D'^\dagger] \vdash_{Op(\Sigma)} Sp'' : \langle p^-_d \rangle$ (ind. hyp. (part 2) on Sp')
  $\Psi; \Gamma^\dagger \vdash_{Op(\Sigma)} \lambda x.\, \{\mathsf{let}\, T\, \mathsf{in}\, y\} : \mathsf{eval\_a}\, t_1 \rightarrowtail \bigcirc(\mathsf{retn\_a}\, t_2)$ (construction)
  $\Psi; \Gamma^\dagger, [D^\dagger] \vdash_{Op(\Sigma)} !(\lambda x.\, \{\mathsf{let}\, T\, \mathsf{in}\, y\}); Sp' : \langle p^-_d \rangle$ (construction)

* If $D = !G \rightarrowtail D'$, then $Sp = !N; Sp'$. The necessary result is $!N'; Sp''$. We get N' from the induction hypothesis (part 3) on N and get Sp'' from the induction hypothesis (part 2) on Sp'.

The cases of part 3 are straightforward invocations of the induction hypothesis (part 1 or part 3). For instance, if $G = !D \rightarrowtail G'$ then we have a derivation of the form $\lambda x.N$ where $\Psi; \Gamma, x{:}D\ pers \vdash_\Sigma N : G'$. By the induction hypothesis (part 3) we have $\Psi; \Gamma^\dagger, x{:}D^\dagger\ pers \vdash_{Op(\Sigma)} N' : G'^\dagger$, and we conclude by constructing $\lambda x.N'$.

In part 4, we have $\Psi; \Gamma \vdash_\Sigma \mathsf{r} \cdot Sp : \langle p^-_d \rangle$, where $\mathsf{r}{:}C \in \Sigma$ and the proposition C is equivalent to $\forall \overline{x_0} \ldots \forall \overline{x_n}.\, A^+_n \rightarrowtail \ldots \rightarrowtail A^+_1 \rightarrowtail \mathsf{a_c}\, t_0\, t_{n+1}$ as described in Section 6.1.2. This means that, for each $0 \leq i \leq n$, we can decompose Sp to get $\sigma_i = (\overline{s_0}/\overline{x_0}, \ldots, \overline{s_i}/\overline{x_i})$ (for some terms $\overline{s_0} \ldots \overline{s_i}$ that correspond to the correct variables) and we have a value $\Psi; \Gamma \vdash_\Sigma V_i : [\sigma_i A^+_i]$. We also have $t = \sigma_0 t_0$ and $s = \sigma_n t_{n+1}$. It suffices to show that, for any $1 \leq i \leq n$, there is

* a spine Sp such that $\Psi; \Gamma^\dagger, [[\![A^+_i, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_0)] \vdash_{Op(\Sigma)} Sp : \langle \bigcirc C^+ \rangle$,
* a pattern+trace $\lambda P.T :: (\Psi; \Theta^\dagger\{C^+\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Theta^\dagger\{y{:}\mathsf{retn\_a}\, (\sigma_n t_{n+1})\})$.[6]

Once we prove this, the trace we need to show is $\{P\} \leftarrow \mathsf{r} \cdot (s_0; Sp); T$.

We will prove this by induction on the length of the sequence sequence $A_i, \ldots, A_n$, and proceed by case analysis on the definition of the operationalization transformation:

* $[\![]\!](\mathsf{a}, t_{n+1}, \sigma_n) = \bigcirc(\mathsf{retn\_a}\, (\sigma_n t_{n+1}))$

  This is a base case: let $Sp = \text{NIL}$, $P = y$, and $T = \diamond$, and we are done.

* $[\![!\mathsf{a_c}\, t^{in}_n\, t_{n+1}]\!](\mathsf{a}, t_{n+1}, \sigma_{n-1}) = \bigcirc(\mathsf{eval\_a}\, (\sigma_{n-1} t^{in}_n))$

  We are given a value $\Psi; \Gamma \vdash_\Sigma !N : [!\mathsf{a_c}\, (\sigma_n t^{in}_n)\, (\sigma_n t_{n+1})]$; observe that $\sigma_{n-1} t^{in}_n = \sigma_n t^{in}_n$.
  This is also a base case: let $Sp = \text{NIL}$, and let $P = x_n :: (\Psi; \Theta^\dagger\{\mathsf{eval\_a}\, (\sigma_n t^{in}_n)\}) \Longrightarrow_{Op(\Sigma)} (\Psi; \Theta^\dagger\{x_n{:}\langle \mathsf{eval\_a}\, (\sigma_n t^{in}_n) \rangle\})$. We need a trace $T :: (\Psi; \Theta^\dagger\{x_n{:}\langle \mathsf{eval\_a}\, (\sigma_n t^{in}_n) \rangle\}) \rightsquigarrow^*_{Op(\Sigma)}$

[5] Recall that $\bigcirc C^+$ and $\{C^+\}$ are synonyms for the internalization of the lax modality; we will use the $\bigcirc$ in the course of this proofs in this section.

[6] The derived pattern+trace form is discussed at the end of Section 4.2.6.

$(\Psi; \Theta^\dagger\{y{:}\langle \mathsf{retn\_a}\ (\sigma_n t_{n+1})\rangle\})$; this follows from the outer induction hypothesis (part 4) on N.

* $[\![p^+_{pers}, A^+_{i+1}, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1}) = \forall \overline{x_i}.\ \sigma_{i-1} p^+{}_{pers} \rightarrowtail [\![A^+_{i+1}, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1})$

$\Psi; \Gamma \vdash_\Sigma z : [\sigma_i p^+_{pers}]$ (given)
$\sigma_i = (\sigma_{i-1}, \overline{s_i}/\overline{x_i})$ (definition of σ_i)
$\Psi; \Gamma, [[\![A^+_{i+1}, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_i)] \vdash_{Op(\Sigma)} Sp' : \langle \bigcirc C^+ \rangle$ (by inner ind. hyp.)
$\lambda P.T :: (\Psi; \Theta^\dagger\{C^+\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Theta^\dagger\{y{:}\mathsf{retn\_a}\ (\sigma_n t_{n+1})\})$ ”
$\Psi; \Gamma, [\forall \overline{x_i}.\ (\sigma_{i-1} p^+{}_{pers}) \rightarrowtail [\![A^+_{i+1}, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1})] \vdash_{Op(\Sigma)} (\overline{s_i}; z; Sp') : \langle \bigcirc C^+ \rangle$ (construction)

We conclude by letting $Sp = \overline{s_i}; z; Sp'$.

* $[\![!p^-_{ci}, \ldots, !p^-_{cj}, A^+_{j+1}, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1})$
$$= \bigcirc \left(\begin{array}{l} \mathsf{eval\_bi}\ (\sigma_{i-1} t^{in}_i) \bullet \ldots \bullet \mathsf{eval\_bj}\ (\sigma_{i-1} t^{in}_j) \bullet \\ (\forall \overline{x_i} \ldots \forall \overline{x_j}.\ \mathsf{retn\_bi}\ (\sigma_{i-1} t^{out}_i) \bullet \ldots \bullet \mathsf{retn\_bj}\ (\sigma_{i-1} t^{out}_j) \\ \quad \rightarrowtail [\![A^+_{j+1}, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1}))) \end{array} \right)$$
(where p^-_{ck} is $\mathsf{bk_c}\ t^{in}_k\ t^{out}_k$ and $\emptyset = FV(t^{in}_k) \cap (\overline{x_i} \cup \ldots \cup \overline{x_j})$ for $i \leq k \leq j$)

Let $Sp = \text{NIL}$ and $P = y_i, \ldots, y_j, y_{ij}$. It suffices to show that there is a trace

$$\begin{array}{rl} T :: & (\Psi, \Theta^\dagger\{y_i{:}\langle \mathsf{eval\_bi}\ (\sigma_{i-1} t^{in}_i)\rangle, \ldots, y_j{:}\langle \mathsf{eval\_bj}\ (\sigma_{i-1} t^{in}_j)\rangle, \\ & \qquad y_{ij}{:}(\forall \overline{x_i} \ldots \forall \overline{x_j}.\ \mathsf{retn\_bi}\ (\sigma_{i-1} t^{out}_i) \bullet \ldots \bullet \mathsf{retn\_bj}\ (\sigma_{i-1} t^{out}_j) \\ & \qquad\qquad \rightarrowtail [\![A^+_{j+1}, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1}))\ ord\}) \\ & \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Theta^\dagger\{z{:}\langle \mathsf{retn\_a}\ (\sigma_n t_{n+1})\rangle\}) \end{array}$$

$\Psi; \Gamma \vdash_\Sigma !N_k : [!\mathsf{bk_c}\ (\sigma_k t^{in}_k)\ (\sigma_k t^{out}_k)] \quad (i \leq k \leq j)$ (given)
$\Psi; \Gamma \vdash_\Sigma !N_k : [!\mathsf{bk_c}\ (\sigma_{i-1} t^{in}_k)\ (\sigma_j t^{out}_k)] \ (i \leq k \leq j)$ (condition on translation, defn. of σ_k)
$T_0 :: (\Psi, \Theta^\dagger\{y_i{:}\langle \mathsf{eval\_bi}\ (\sigma_{i-1} t^{in}_i)\rangle, \ldots, y_j{:}\langle \mathsf{eval\_bj}\ (\sigma_{i-1} t^{in}_j)\rangle,$
$y_{ij}{:}(\forall \overline{x_i} \ldots \forall \overline{x_j}.\ \mathsf{retn\_bi}\ (\sigma_{i-1} t^{out}_i) \bullet \ldots \bullet \mathsf{retn\_bj}\ (\sigma_{i-1} t^{out}_j)$
$\rightarrowtail [\![A^+_{j+1}, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1}))\ ord\})$
$\rightsquigarrow^*_{Op(\Sigma)} (\Psi, \Theta^\dagger\{z_i{:}\langle \mathsf{retn\_bi}\ (\sigma_j t^{out}_i)\rangle, \ldots, z_j{:}\langle \mathsf{retn\_bj}\ (\sigma_j t^{out}_j)\rangle,$
$y_{ij}{:}(\forall \overline{x_i} \ldots \forall \overline{x_j}.\ \mathsf{retn\_bi}\ (\sigma_{i-1} t^{out}_i) \bullet \ldots \bullet \mathsf{retn\_bj}\ (\sigma_{i-1} t^{out}_j)$
$\rightarrowtail [\![A^+_{j+1}, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1}))\ ord\})$
(by outer ind. hyp. (part 4) on each of the N_k in turn)
$\Psi; \Gamma, [[\![A^+_{j+1}, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_j)] \vdash_{Op(\Sigma)} Sp' : \langle \bigcirc C^+ \rangle$ (by inner ind. hyp.)
$\lambda P'.T' :: (\Psi, \Theta^\dagger\{C^+\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi, \Theta^\dagger\{y{:}\langle \mathsf{retn\_a}\ s\rangle\})$ ”

We conclude by letting $T = (T_0; \{P'\} \leftarrow y_{ij} \cdot (\overline{s_i}; \ldots \overline{s_j}; (y_i \bullet \ldots \bullet y_j); Sp'); T')$.

$*$ $[\![!G, A^+_{i+1}, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1}) = \forall \overline{x_i}.\, !(\sigma_{i-1} G^\dagger) \rightarrowtail [\![A^+_i, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1})$

$\Psi; \Gamma \vdash_\Sigma\ !N : [!\sigma_i G]$ (given)
$\Psi; \Gamma^\dagger \vdash_{Op(\Sigma)} N' : \sigma_i G^\dagger$ (by outer ind. hyp. (part 3) on N)
$\sigma_i = (\sigma_{i-1}, \overline{s_i}/\overline{x_i})$. (definition of σ_i)
$\Psi; \Gamma, [[\![A^+_i, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_i)] \vdash_{Op(\Sigma)} Sp' : \langle \bigcirc C^+ \rangle$ (by inner ind. hyp.)
$\lambda P.T :: (\Psi; \Theta^\dagger\{C^+\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Theta^\dagger\{y{:}\mathsf{retn\_a}\, s\})$ ”
$\Psi; \Gamma, [\forall \overline{x_i}.\, !(\sigma_{i-1} G) \rightarrowtail [\![A^+_i, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1})] \vdash_{Op(\Sigma)} (\overline{s_i}; !N'; Sp') : \langle \bigcirc C^+ \rangle$ (construction)

We conclude by letting $Sp = \overline{s_i}; !N'; Sp'$.

This completes the inner induction in the fourth part, and hence the completeness proof. □

Soundness

Soundness under the parallel translation requires us to prove an inversion lemma like the one that we first encountered in the proof of preservation for adequacy (Theorem 4.7). To this end, we describe two new refinements of negative propositions that capture the structure of transformed concurrent rules.

$$R ::= \forall \overline{x}.\, \mathsf{retn\_b1}\, t_1 \bullet \ldots \bullet \mathsf{retn\_bn}\, t_n \rightarrowtail S$$
$$S ::= \forall x{:}\tau.\, S \mid p^+_{pers} \rightarrowtail S \mid\ !A^- \rightarrowtail S \mid \bigcirc(\mathsf{eval\_b1}\, t_1 \bullet \ldots \bullet \mathsf{eval\_bn}\, t_n \bullet \downarrow R) \mid \bigcirc(\mathsf{eval\_b}\, t)$$

Every concurrent rule in a transformed signature $Op(\Sigma)$ has the form $\mathsf{r} : \forall \overline{x}.\, \mathsf{eval\_b}\, t \rightarrowtail S$.

Theorem 6.3 (Rearrangement). *If Δ contains only atomic propositions, persistent propositions of the form D, and ordered propositions of the form R, and if Γ matches $z{:}\langle \mathsf{retn\_z}\, t_z \rangle$, then*

1. *If Δ matches $\Theta\{\!\{x_1{:}\langle \mathsf{retn\_b1}\, t_1 \rangle, \ldots, x_n{:}\langle \mathsf{retn\_bn}\, t_n \rangle, y{:}(\forall \overline{x}.\, \mathsf{retn\_b1}\, s_1 \bullet \ldots \bullet \mathsf{retn\_bn}\, s_n \rightarrowtail S)\}\!\}$ and $T :: (\Psi; \Delta) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Gamma)$, then $T = \{P\} \leftarrow y \cdot (\overline{u}; (x_1 \bullet \ldots \bullet x_n); Sp); T'$ where $(\overline{u}/\overline{x}) s_i = t_i$ for $1 \le i \le n$.*
2. *If Δ matches $\Theta\{\!\{y{:}\langle \mathsf{eval\_b}\, t \rangle\}\!\}$ and $T :: (\Psi; \Delta) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Gamma)$, then $T = \{P\} \leftarrow \mathsf{r} \cdot (\overline{u}; y; Sp); T'$ where $\mathsf{r} : \forall \overline{x}.\, \mathsf{eval\_b}\, s \rightarrowtail S \in Op(\Sigma)$.*

Proof. In both cases, the proof is by induction on the structure of T and case analysis on the first steps in T. If the first step does not proceed by focusing on y (part 1) or focusing on a rule in the signature and consuming y (part 2), then we proceed by induction on the smaller trace to move the relevant step to the front. We have to check that the first step doesn't output any variables that are input variables of the relevant step. This is immediate from the structure of R, S, and transformed signatures. □

The main soundness theorem is Theorem 6.4. The first three cases of Theorem 6.4 are straightforward transformations from deductive proofs to deductive proofs, and the last two cases are the key. In the last two cases, we take a trace that, by its type, must contain the information needed to reconstruct a deductive proof.

Theorem 6.4 (Soundness of operationalization). *If all propositions in* Γ *have the form* $x{:}D$ *pers or* $z{:}\langle p^+_{pers}\rangle$*, and all propositions in* Δ *have the form* $x{:}D$ *pers,* $z{:}\langle p^+_{pers}\rangle$*,* $x{:}\langle \mathsf{eval\_b}\, t\rangle$*,* $x{:}\langle \mathsf{retn\_b}\, t\rangle$*, or* $x{:}R$ *ord, then*

1. *If* $\Psi; \Gamma^\dagger \vdash_{Op(\Sigma)} \langle p^-_d\rangle$*, then* $\Psi; \Gamma \vdash_\Sigma \langle p^-_d\rangle$.
2. *If* $\Psi; \Gamma^\dagger, [D^\dagger] \vdash_{Op(\Sigma)} \langle p^-_d\rangle$*, then* $\Psi; \Gamma, [D] \vdash_\Sigma \langle p^-_d\rangle$.
3. *If* $\Psi; \Gamma^\dagger \vdash_{Op(\Sigma)} G^\dagger$*, then* $\Psi; \Gamma \vdash_\Sigma G$.
4. *If* Δ *matches* $\Theta^\dagger\{\!\{\Gamma^\dagger\}\!\}$*,* Γ_z *matches* $z{:}\langle \mathsf{retn\_z}\, t_z\rangle$*,*
   and $(\Psi; \Theta^\dagger\{\!\{\Gamma, x{:}\langle \mathsf{eval\_a}\, t\rangle\}\!\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Gamma_z)$*,*
   then there exists an s *s.t.* $\Psi; \Gamma \vdash_\Sigma \langle \mathsf{a_c}\, t\, s\rangle$ *and* $(\Psi; \Theta^\dagger\{y{:}\langle \mathsf{retn\_a}\, s\rangle\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Gamma_z)$.
5. *If* Δ *matches* $\Theta^\dagger\{\!\{\Gamma^\dagger\}\!\}$*,* Γ_z *matches* $z{:}\langle \mathsf{retn\_z}\, t_z\rangle$*,*
   $\Psi; \Gamma^\dagger, [[\![A_i, \ldots, A_n]\!](\mathsf{a}, t_{n+1}, \sigma_i)] \vdash_{Op(\Sigma)} \langle \bigcirc C^+\rangle$*, and* $(\Psi; \Theta^\dagger\{C^+\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Gamma_z)$*,*
   then there exists a $\sigma \supseteq \sigma_i$ *such that* $\Psi; \Gamma \vdash_\Sigma [\sigma A^+_j]$ *for* $i \le j \le n$
   and $(\Psi; \Theta^\dagger\{y{:}\langle \mathsf{retn\_a}\, (\sigma t_{n+1})\rangle\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Delta_z)$.

Proof. By induction on the structure of the given derivation or given trace. Parts 1 and 3 exactly match the structure of the completeness proof (Theorem 6.2); the only difference in Part 2 is the case where $D = !p_c \rightarrowtail D'$. In his case, we reason as follows:

$D = !p_c \rightarrowtail D'$, where $p_c = \mathsf{a_c}\, t_1\, t_2$	(given)
$D^\dagger = !(\mathsf{eval\_a}\, t_1 \rightarrowtail \bigcirc(\mathsf{retn\_a}\, t_2)) \rightarrowtail D'^\dagger$	(definition of $D^\dagger$)
$\Psi; \Gamma, [!(\mathsf{eval\_a}\, t_1 \rightarrowtail \bigcirc(\mathsf{retn\_a}\, t_2)) \rightarrowtail D'^\dagger] \vdash_{Op(\Sigma)} Sp : \langle p^-_d\rangle$	(given)
$Sp = !(\lambda x.\, \{\mathsf{let}\, T\, \mathsf{in}\, y\}); Sp'$, where	(inversion on the type of Sp)
$\quad T :: (\Psi; \Gamma, x{:}\langle \mathsf{eval\_a}\, t_1\rangle) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Gamma_z)$,	”
$\quad \Gamma_z$ matches $y{:}\langle \mathsf{retn\_a}\, t_2\rangle$, and	”
$\quad \Psi; \Gamma, [D'^\dagger] \vdash_{Op(\Sigma)} Sp' : \langle p^-_d\rangle$	”
$\Psi; \Gamma, [D'] \vdash_\Sigma Sp'' : \langle p^-_d\rangle$	(ind. hyp. (part 2) on Sp')
$\Psi; \Gamma \vdash_\Sigma N : \langle \mathsf{a_c}\, t_1\, s\rangle$	(ind. hyp. (part 4) on T)
$T' :: (\Psi; \Gamma, y'{:}\langle \mathsf{retn\_a}\, s\rangle) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Gamma_z)$	”
$T' = \diamond$, $y' = y$, and $t_2 = s$	(case analysis on the structure of T')
$\Psi; \Gamma, [!p_c \rightarrowtail D'] \vdash_\Sigma !N; Sp' : \langle p^-_d\rangle$	(construction)

Part 4 we let $\sigma_0 = (\overline{u}/\overline{x})$.

$T :: (\Psi; \Theta^\dagger\{\!\{x{:}\langle \mathsf{eval\_a}\, t\rangle\}\!\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Gamma_z)$	(given)
$T = \{P\} \leftarrow \mathsf{r} \cdot (\overline{u}; x; Sp); T'$	(Theorem 6.3 on T)
$\quad \mathsf{r} : \forall \overline{x_0}.\, \mathsf{eval\_a}\, t_0 \rightarrowtail [\![A^+_1, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \mathsf{id}) \in Op(\Sigma)$	”
$\quad t = \sigma_0 t_0$	”
$\quad \Psi; \Gamma, [[\![A^+_1, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_0)] \vdash_{Op(\Sigma)} Sp : \langle \bigcirc C^+\rangle$	”
$\quad \lambda P.T' :: (\Psi; \Theta^\dagger\{C^+\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Gamma_z)$	”
$\Psi; \Gamma \vdash_\Sigma V_i : [\sigma A^+_i]$ for $1 \le j \le n$	(ind. hyp. (part 5) on Sp and T')
$T'' :: (\Psi; \Theta^\dagger\{y{:}\langle \mathsf{retn\_a}\, (\sigma t_{n+1})\rangle\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Gamma_z)$	”

We needed to show a derivation and a trace: the trace T'' is precisely the latter thing. For the derivation of $\mathsf{a_c}\, t\, s$ (for some s), we observe that $\mathsf{r} \in \Sigma$ has a type equivalent to $\forall \overline{x_0} \ldots \forall \overline{x_n}.\, A^+_n \rightarrowtail \ldots \rightarrowtail A^+_0 \rightarrowtail \mathsf{a_c}\, t_0\, t_{n+1}$. Therefore, by letting $s = \sigma t_{n+1}$ and using the V_i from the induction

hypothesis (part 5) above, we can construct a derivation of $\Psi; \Gamma \vdash_\Sigma \langle \mathsf{a_c}\, t\, s \rangle$ by focusing on r, which is the other thing we needed to show.

The step above where we assume that the head of r involves the predicate $\mathsf{a_c}$ is the only point in the correctness proofs where we rely on the fact that each transformed predicate eval_a is associated with exactly one original-signature predicate $\mathsf{a_c}$ – if this were not the case and eval_a were also associated with an original-signature predicate b, then we might get the "wrong" derivation back from this step. We do not have to similarly rely on retn_a being similarly uniquely associated with $\mathsf{a_c}$.

Part 5 We are given a spine $\Psi; \Gamma^\dagger, [[\![A_i, \ldots, A_n]\!](\mathsf{a}, t_{n+1}, \sigma_i)] \vdash_{Op(\Sigma)} Sp : \langle \bigcirc C^+ \rangle$ and a trace $\lambda P.T :: (\Psi; \Theta^\dagger\{C^+\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Gamma_z)$. We need to show a $\sigma \supseteq \sigma_i$, an $\Psi; \Gamma \vdash_{Op(\Sigma)} N : [\sigma A_k^+]$ for $i \leq j \leq n$, and $T' :: (\Psi; \Theta^\dagger\{y{:}\langle \mathsf{retn\_a}\, (\sigma t_{n+1}) \rangle\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Delta_z)$.

We proceed by case analysis on the definition of the operationalization transformation:

* $[\ = \bigcirc(\mathsf{retn\_a}\, (\sigma_n t_{n+1}))$

  This is a base case: there are no values to construct. By inversion on the type of P we know it has the form $y :: (\Psi; \Theta^\dagger\{\mathsf{retn\_a}\, (\sigma_n t_{n+1})\}) \Longrightarrow_{Op(\Sigma)} (\Psi; \Theta^\dagger\{y{:}\langle \mathsf{retn\_a}\, (\sigma_n t_{n+1}) \rangle\})$. Let $\sigma = \sigma_n$ and we are done; the trace T is the necessary trace.

  Because $(\Psi; \Theta^\dagger\{\mathsf{retn\_a}\, (\sigma_n t_{n+1})\})$ decomposes to $(\Psi; \Theta^\dagger\{y{:}\langle \mathsf{retn\_a}\, (\sigma_n t_{n+1}) \rangle\})$, we let $\sigma = \sigma_n$ and we are done.

* $[\![!\mathsf{a_c}\, t_n^{in}\, t_{n+1}]\!](\mathsf{a}, t_{n+1}, \sigma_{n-1}) = \bigcirc(\mathsf{eval\_a}\, (\sigma_{n-1} t_n^{in}))$

  This is also a base case: we have one value of type $!\mathsf{a_c}\, (\sigma t_n^{in})\, (\sigma t_{n+1})$ to construct, where $\sigma \supseteq \sigma_{n-1}$. By inversion on the type of P we know that the pattern has the form $y :: (\Psi; \Theta^\dagger\{\mathsf{eval\_a}\, (\sigma_n t_n^{in})\}) \Longrightarrow_{Op(\Sigma)} (\Psi; \Theta^\dagger\{y{:}\langle \mathsf{eval\_a}\, (\sigma_n t_n^{in}) \rangle\})$. This means we also have that $T :: (\Psi; \Theta^\dagger\{y{:}\langle \mathsf{eval\_a}\, (\sigma_n t_n^{in}) \rangle\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Delta_z)$.

  By the induction hypothesis (part 4) on T, we have an s such that $\Psi; \Gamma \vdash_\Sigma N : \langle \mathsf{a_c}\, t\, s \rangle$ and a trace $T' :: (\Psi; \Theta^\dagger\{y{:}\langle \mathsf{retn\_a}\, s \rangle\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Gamma_z)$.

  We can only apply tail-recursion optimization when t_{n+1} is fully general, which means we can construct a $\sigma \supseteq \sigma_{n-1}$ such that $\sigma t_{n+1} = s$. The value we needed to construct is just $!N$, and the trace T' is in the form we need, so we are done.

* $[\![p^+_{pers}, A^+_{i+1}, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1}) = \forall \overline{x_i}.\, \sigma_{i-1} p^+{}_{pers} \rightarrowtail [\![A^+_{i+1}, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1})$

  By type inversion, the spine $Sp = \overline{u_i}; z; Sp'$. Let $\sigma_i = (\sigma_{i-1}, \overline{u_i}/\overline{x_i})$. The induction hypothesis (part 5) on Sp' and T gives $\sigma \supseteq \sigma_i$, values σA^+_j for $i < j \leq n$, and a trace $T' :: (\Psi; \Theta^\dagger\{y{:}\langle \mathsf{retn\_a}\, (\sigma t_{n+1}) \rangle\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Delta_z)$. The remaining value of $\sigma A_i = \sigma p^+_{pers}$ is just z.

* $[\![!p^-_{ci}, \ldots, !p^-_{cj}, A^+_{j+1}, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1})$

$$= \bigcirc \left(\begin{array}{l} \mathsf{eval\_bi}\, (\sigma_{i-1} t_i^{in}) \bullet \ldots \bullet \mathsf{eval\_bj}\, (\sigma_{i-1} t_j^{in}) \bullet \\ (\forall \overline{x_i} \ldots \forall \overline{x_j}.\, \mathsf{retn\_bi}\, (\sigma_{i-1} t_i^{out}) \bullet \ldots \bullet \mathsf{retn\_bj}\, (\sigma_{i-1} t_j^{out}) \\ \quad \rightarrowtail [\![A^+_{j+1}, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1})) \end{array} \right)$$

  (where p^-_{ck} is $\mathsf{bk_c}\, t_k^{in}\, t_k^{out}$ and $FV(t_k^{in}) \notin (\overline{x_i} \cup \ldots \cup \overline{x_j})$ for $i \leq k \leq j$)

Let $R = \forall \overline{x_i} \ldots \forall \overline{x_j}.\, \mathsf{retn\_bi}\,(\sigma_{i-1} t_i^{out}) \bullet \ldots \bullet \mathsf{retn\_bj}\,(\sigma_{i-1} t_j^{out}) \rightarrowtail [\![A_{j+1}^+, \ldots, A_n^+]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1})$

By inversion on the type of P, we know it has the form $y_i, \ldots, y_j, y$, and we know that $T :: (\Psi; \Theta^\dagger\{y_i{:}\langle \mathsf{eval\_bi}\,(\sigma_{i-1} t_i^{in})\rangle, \ldots, y_j{:}\langle \mathsf{eval\_bj}\,(\sigma_{i-1} t_j^{in})\rangle, y{:}R\ ord\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi, \Delta_z)$.

By $j - i$ applications of the induction hypothesis (part 4) starting with T, we obtain for $i \le k \le j$ a derivation of $\Psi; \Gamma \vdash_\Sigma N_k : \langle \mathsf{bk_c}\,(\sigma_{i-1} t_k^{in})\ s_k\rangle$ and a smaller trace $T'' :: (\Psi; \Theta^\dagger\{z_i{:}\langle \mathsf{retn\_bi}\ s_i\rangle, \ldots, z_j{:}\langle \mathsf{retn\_bj}\ s_j\rangle, y{:}R\ ord\}) \rightsquigarrow^+_{Op(\Sigma)} (\Psi; \Delta_z)$.

By Theorem 6.3 we get that $T'' = \{P\} \leftarrow y \cdot (\overline{u_i}; \ldots; \overline{u_j}; (z_i \bullet \ldots \bullet z_j); Sp'); T'''$.

Let $\sigma_j = (\sigma_{i-1}, \overline{u_i}/\overline{x_i}, \ldots, \overline{u_j}/\overline{x_j})$. Then we have that $s_k = \sigma_j t_k^{out}$ for $i \le k \le j$ and $\Psi; \Gamma^\dagger, [[\![A_{j+1}^+, \ldots, A_n^+]\!](\mathsf{a}, t_{n+1}, \sigma_j)] \vdash_{Op(\Sigma)} Sp' : \langle \bigcirc C^+\rangle$.*

The induction hypothesis (part 5) on Sp' and T''' gives $\sigma \supseteq \sigma_j$, values σA_k^+ for $j < k \le n$, and a trace $T' :: (\Psi; \Theta^\dagger\{y{:}\langle \mathsf{retn\_a}\,(\sigma t_{n+1})\rangle\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Delta_z)$. The remaining values of type $\sigma A_k = !(\sigma p_{ck}^-)$ for $i \le k \le j$ all have the form $!N_k$ (where the N_k were constructed above by invoking part 4 of the induction hypothesis).

* $[\![!G, A_{i+1}^+, \ldots, A_n^+]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1}) = \forall \overline{x_i}.\, !\sigma_{i-1} G^\dagger \rightarrowtail [\![A_i^+, \ldots, A_n^+]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1})$

By type inversion, the spine $Sp = \overline{u_i}; !N; Sp'$. Let $\sigma_i = (\sigma_{i-1}, \overline{u_i}/\overline{x_i})$ The induction hypothesis (part 5) on Sp' and T gives $\sigma \supseteq \sigma_i$, values σA_j^+ for $i < j \le n$, and a trace $T' :: (\Psi; \Theta^\dagger\{y{:}\langle \mathsf{retn\_a}\,(\sigma t_{n+1})\rangle\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Delta_z)$. The remaining value of type $\sigma A_i = !(\sigma G)$ is $!N'$, where we get $\Psi; \Gamma \vdash_\Sigma N' : \sigma G$ from the induction hypothesis (part 3) on N.

This completes the proof. □

6.2 Logical transformation: defunctionalization

Defunctionalization is a procedure for turning nested SLS specifications into flat SLS specifications. The key idea is that a nested rule can always be simulated by a distinguished atomic proposition by inserting a rule into the signature that teaches the atomic proposition how to act like the nested rule. (Or, looking at it the other way around, the new atomic propositions act like triggers for the additional rules.) The correctness of defunctionalization follows from a simple lock-step bisimulation argument – if a trigger can cause a rule in the defunctionalized signature to fire, then that trigger corresponds to a nested rule in the context that can fire immediately in the non-defunctionalized process state, and vice-versa.

The defunctionalization procedure implemented in the SLS prototype is actually three transformations. The first is properly a defunctionalization transformation (Section 6.2.1), the second is an uncurrying transformation (Section 6.2.2), and the third is a refactoring that transforms a family of cont-like predicates into a single cont predicate and a family of frames (Section 6.2.3). We will explain each of these transformations in turn; one example of the full three-part defunctionalization transformation on a propositional signature is given in Figure 6.4.

$$
\begin{array}{lcl}
\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d} : \mathsf{prop}\,\mathsf{ord}, & & \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d} : \mathsf{prop}\,\mathsf{ord}, \\
 & & \mathsf{frame} : \mathsf{type}, \\
 & & \mathsf{cont} : \mathsf{frame} \to \mathsf{type}. \\
 & & \mathsf{frameA1} : \mathsf{frame}, \\
\mathsf{ruleA} : \mathsf{a} \rightarrowtail \{\mathsf{b} \bullet \downarrow(\mathsf{c} \rightarrowtail \{\mathsf{d}\})\}, & & \mathsf{ruleA} : \mathsf{a} \rightarrowtail \{\mathsf{b} \bullet \mathsf{cont}\,\mathsf{frameA1}\}, \\
 & \Longrightarrow & \mathsf{ruleA1} : \mathsf{c} \bullet \mathsf{cont}\,\mathsf{frameA1} \rightarrowtail \{\mathsf{d}\}, \\
 & & \mathsf{frameB1} : \mathsf{frame}, \\
 & & \mathsf{frameB2} : \mathsf{frame}, \\
\mathsf{ruleB} : \mathsf{c} \rightarrowtail \{\downarrow(\mathsf{d} \rightarrowtail \{\downarrow(\mathsf{a} \bullet \mathsf{a} \rightarrowtail \{\mathsf{b}\})\})\} & & \mathsf{ruleB} : \mathsf{c} \rightarrowtail \{\mathsf{cont}\,\mathsf{frameB1}\}, \\
 & & \mathsf{ruleB1} : \mathsf{d} \bullet \mathsf{cont}\,\mathsf{frameB1} \rightarrowtail \{\mathsf{cont}\,\mathsf{frameB2}\}, \\
 & & \mathsf{ruleB2} : \mathsf{a} \bullet \mathsf{a} \bullet \mathsf{cont}\,\mathsf{frameB2} \rightarrowtail \{\mathsf{b}\}
\end{array}
$$

Figure 6.4: Defunctionalization on a nested SLS signature

6.2.1 Defunctionalization

Defunctionalization is based on the following intuitions: if A^- is a closed negative proposition and we have a single-step transition $\{P\} \leftarrow y \cdot Sp :: (\Psi; \Theta\{y{:}A^-\ ord\}) \rightsquigarrow_\Sigma (\Psi; \Delta')$, then we can define an augmented signature

$$
\begin{array}{rcl}
\Sigma' = & \Sigma, & \\
 & \mathsf{cont} : \mathsf{prop}\,\mathsf{ord}, & \\
 & \mathsf{run\_cont} : \mathsf{cont} \rightarrowtail A^- &
\end{array}
$$

and it is the case that $(\Psi; \Theta\{y{:}\langle \mathsf{cont} \rangle\}) \rightsquigarrow_{\Sigma'} (\Psi; \Delta')$ as well. Whenever the step $\{P\} \leftarrow y \cdot Sp$ is possible under Σ, the step $\{P\} \leftarrow \mathsf{run\_cont} \cdot (y; Sp)$ will be possible under Σ', and vice versa. (It would work just as well for run_cont to be $\mathsf{cont} \twoheadrightarrow A^-$ – the persistent propositions $A^+ \rightarrowtail B^-$ and $A^+ \twoheadrightarrow B^-$ are indistinguishable in ordered logic.)

Because rules in the signature must be closed negative propositions, this strategy won't work for a transition that mentions free variables from the context. However, every open proposition $\Psi \vdash_\Sigma A^-\ \mathsf{type}^-$ can be refactored as an open proposition $a_1{:}\tau_1, \ldots, a_n{:}\tau_n \vdash B^-\ \mathsf{type}^-$ and a substitution $\sigma = (t_1/a_1, \ldots, t_n/a_n)$ such that $\Psi \vdash_\Sigma \sigma : a_1{:}\tau_1, \ldots, a_n{:}\tau_n$ and $\sigma B^- = A^-$. Then, we can augment the signature in this more general form:

$$
\begin{array}{rcl}
\Sigma'' = & \Sigma, & \\
 & \mathsf{cont} : \Pi a_1{:}\tau_1 \ldots \Pi a_n{:}\tau_n.\ \mathsf{prop}\,\mathsf{ord} & \\
 & \mathsf{run\_cont} : \forall a_1{:}\tau_1 \ldots \forall a_n{:}\tau_n.\ \mathsf{cont}\, a_1 \ldots a_n \rightarrowtail B^- &
\end{array}
$$

As before, whenever the step $\{P\} \leftarrow y \cdot Sp :: (\Psi; \Theta\{y : A^-\ ord\}) \rightsquigarrow^*_\Sigma (\Psi'; \Delta')$ is possible under Σ, the step $\{P\} \leftarrow \mathsf{run\_cont} \cdot (t_1; \ldots; t_n; y; Sp) :: (\Psi; \Theta\{y : \langle \mathsf{cont}\, t_1 \ldots t_n \rangle\}) \rightsquigarrow^*_{\Sigma'} (\Psi'; \Delta')$ will be possible under Σ', and vice versa.

Taking this a step further, if we have a signature $\Sigma_1 = \Sigma, \mathsf{rule} : \ldots \rightarrowtail \bigcirc(\ldots \bullet \downarrow B^- \bullet \ldots)$ where the variables $a_1 \ldots a_n$ are free in B^-, then any trace in this signature will be in lock-step bisimulation with a trace in this signature:

$$\begin{aligned}\Sigma_2 = \; & \Sigma, \\ & \mathsf{cont} : \Pi a_1{:}\tau_1 \ldots \Pi a_n{:}\tau_n.\, \mathsf{prop\ ord} \\ & \mathsf{rule} : \ldots \rightarrowtail \bigcirc(\ldots \bullet (\mathsf{cont}\ a_1 \ldots a_n) \bullet \ldots) \\ & \mathsf{run\_cont} : \forall a_1{:}\tau_1 \ldots \forall a_n{:}\tau_n.\, \mathsf{cont}\ a_1 \ldots a_n \rightarrowtail B^-\end{aligned}$$

Specifically, say that $(\Psi; \Delta_1)\, \mathcal{R}\, (\Psi; \Delta_2)$ when Δ_1 is Δ_2 where all variable declarations of the form $y{:}\langle \mathsf{cont}\ t_1 \ldots t_n \rangle\ ord$ in Δ_2 have been replaced by $y{:}(t_1/a_1 \ldots t_n/a_n)B^-\ ord$ in Δ_1. (Note that if $\vdash_{\Sigma_2} (\Psi; \Delta_2)\, \mathsf{state}$ holds, then $\vdash_{\Sigma_1} (\Psi; \Delta_1)\, \mathsf{state}$ holds as well.) It is the case that if $S_1 :: (\Psi; \Delta_1) \rightsquigarrow^*_{\Sigma_1} (\Psi'; \Delta_1')$ then $S_2 :: (\Psi; \Delta_2) \rightsquigarrow^*_{\Sigma_2} (\Psi'; \Delta_2')$ where $(\Psi'; \Delta_1')\, \mathcal{R}\, (\Psi'; \Delta_2')$. The opposite also holds: if $S_2 :: (\Psi; \Delta_2) \rightsquigarrow^*_{\Sigma_2} (\Psi'; \Delta_2')$ then $S_1 :: (\Psi; \Delta_1) \rightsquigarrow^*_{\Sigma_1} (\Psi'; \Delta_1')$ where $(\Psi'; \Delta_1')\, \mathcal{R}\, (\Psi'; \Delta_2')$. For transitions not involving rule or run_cont in Σ_2 this is immediate, for transitions involving rule we observe that the propositions introduced by inversion preserve the simulation, and for transitions involving run_cont we use aforementioned fact that the atomic term $y \cdot Sp$ in Σ_1 is equivalent to the atomic term $\mathsf{run\_cont} \cdot (t_1; \ldots; t_n; y; Sp)$ in Σ_2.

We can iterate this defunctionalization procedure on the nested ev/app rule from Figure 6.2:

$$\begin{aligned}&\mathsf{ev/app} : \forall E_1.\forall E_2.\, \mathsf{eval}\ (\mathsf{app}\ E_1\ E_2) \\ &\qquad \rightarrowtail \{\mathsf{eval}\ E_1 \bullet \\ &\qquad\quad \downarrow(\forall E.\, \mathsf{retn}\ (\mathsf{lam}\ \lambda x.E\ x) \\ &\qquad\qquad \rightarrowtail \{\mathsf{eval}\ E_2 \bullet \downarrow(\forall V_2.\, \mathsf{retn}\ V_2 \rightarrowtail \{\mathsf{eval}\ (E\ V_2)\})\})\}.\end{aligned}$$

The outermost nested rule only has the variable E_2 free, so the first continuation we introduce, cont_app1, has one argument.

$$\begin{aligned}&\mathsf{cont\_app1} : \mathsf{exp} \rightarrow \mathsf{prop\ ord}, \\ &\mathsf{ev/app} : \forall E_1.\forall E_2.\, \mathsf{eval}\ (\mathsf{app}\ E_1\ E_2) \rightarrowtail \{\mathsf{eval}\ E_1 \bullet \mathsf{cont\_app1}\ E_2\} \\ &\mathsf{ev/app1} : \forall E_2.\, \mathsf{cont\_app1}\ E_2 \rightarrowtail \forall E.\, \mathsf{retn}\ (\mathsf{lam}\ \lambda x.E\ x) \\ &\qquad \rightarrowtail \{\mathsf{eval}\ E_2 \bullet \downarrow(\forall V_2.\, \mathsf{retn}\ V_2 \rightarrowtail \{\mathsf{eval}\ (E\ V_2)\})\}.\end{aligned}$$

This step turns ev/app into a flat rule; we repeat defunctionalization on ev/app1 to get a completely flat specification. This introduces a new proposition cont_app2 that keeps track of the free variable E with type exp $\rightarrow$ exp.

$$\begin{aligned}&\mathsf{cont\_app1} : \mathsf{exp} \rightarrow \mathsf{prop\ ord}, \\ &\mathsf{cont\_app2} : (\mathsf{exp} \rightarrow \mathsf{exp}) \rightarrow \mathsf{prop\ ord}, \\ &\mathsf{ev/app} : \forall E_1.\forall E_2.\, \mathsf{eval}\ (\mathsf{app}\ E_1\ E_2) \rightarrowtail \{\mathsf{eval}\ E_1 \bullet \mathsf{cont\_app1}\ E_2\} \\ &\mathsf{ev/app1} : \forall E_2.\, \mathsf{cont\_app1}\ E_2 \rightarrowtail \forall E.\, \mathsf{retn}\ (\mathsf{lam}\ \lambda x.E\ x) \rightarrowtail \{\mathsf{eval}\ E_2 \bullet \mathsf{cont\_app2}\ (\lambda x.E\ x)\}. \\ &\mathsf{ev/app2} : \forall E.\, \mathsf{cont\_app2}\ (\lambda x.E\ x) \rightarrowtail \forall V_2.\, \mathsf{retn}\ V_2 \rightarrowtail \{\mathsf{eval}\ (E\ V_2)\}.\end{aligned}$$

```
eval: exp -> prop ord.
retn: exp -> prop ord.
cont_app1: exp -> prop ord.
cont_app2: (exp -> exp) -> prop ord.

ev/lam:  eval (lam \x. E x) >-> {retn (lam \x. E x)}.

ev/app:  eval (app E1 E2) >-> {eval E1 * cont_app1 E2}.

ev/app1: retn (lam \x. E x) * cont_app1 E2
          >-> {eval E2 * cont_app2 (\x. E x)}.

ev/app2: retn V2 * cont_app2 (\x. E x) >-> {eval (E V2)}.
```

Figure 6.5: Uncurried call-by-value evaluation

6.2.2 Uncurrying

The first arrow in a rule can be freely switched between left and right ordered implication: the rules $A^+ \rightarrowtail \{B^+\}$ and $A^+ \twoheadrightarrow \{B^+\}$ are equivalent, for instance. Pfenning used $A^+ \multimap \{B^+\}$ as a generic form of ordered implication for this reason in [Pfe04]. This observation only holds because rules act like persistent resources, however! It does not seem to be possible to treat $\multimap$ as a real connective in ordered logic with well-behaved left and right rules that satisfy cut admissibility, and the observation applies only to the first arrow: while the rule $\mathsf{rule1} : A^+ \rightarrowtail B^+ \rightarrowtail \{C^+\}$ is equivalent to the rule $\mathsf{rule2} : A^+ \twoheadrightarrow B^+ \rightarrowtail \{C^+\}$, these two rules are *not* equivalent to the rule $\mathsf{rule3} : A^+ \rightarrowtail B^+ \twoheadrightarrow \{C^+\}$.

Uncurrying tries to rewrite a rule so that the only arrow is the first one, taking an awkward rule like $A^+ \rightarrowtail B^+ \twoheadrightarrow C^+ \rightarrowtail D^+ \twoheadrightarrow \{E^+\}$ to the more readable flat rule $C^+ \bullet A^+ \bullet B^+ \bullet D^+ \rightarrowtail \{E^+\}$. Uncurrying can only be performed on persistent or linear propositions (or rules in the signature): there is no A^+ that makes the variable declaration $x{:}(p^+ \rightarrowtail q^+ \twoheadrightarrow \{r^+\})\ \mathit{ord}$, equivalent to $x{:}(A^+ \twoheadrightarrow \{r^+\})\ \mathit{ord}$ or $x{:}(A^+ \rightarrowtail \{r^+\})\ \mathit{ord}$ for any A^+.

Thus, defunctionalization and uncurrying work well together: if we replace the variable declaration $x{:}(p^+ \rightarrowtail q^+ \twoheadrightarrow \{r^+\})\ \mathit{ord}$ with the suspended ordered proposition $x{:}\langle \mathsf{cont} \rangle\ \mathit{ord}$ and add a rule $\mathsf{run\_cont} : \mathsf{cont} \rightarrowtail p^+ \rightarrowtail q^+ \twoheadrightarrow \{r^+\}$, that rule can then be uncurried to get the equivalent rule $p^+ \bullet \mathsf{cont} \bullet q^+ \rightarrowtail \{r^+\}$.

If we uncurry the defunctionalized specification for CBV evaluation from the previous section, we get the SLS specification shown in Figure 6.5. This flat specification closely and adequately represents the abstract machine semantics from the beginning of this chapter, but before proving adequacy in Section 6.3, we will make one more change to the semantics.

6.2.3 From many predicates to many frames

This last change we make appears to be largely cosmetic, but it will facilitate, in Section 6.5.4 below, the modular extension of our semantics with recoverable failure.

```
eval: exp -> prop ord.
retn: exp -> prop ord.
cont: frame -> prop ord.

app1: exp -> frame.
app2: (exp -> exp) -> frame.

ev/lam:  eval (lam \x. E x) >-> {retn (lam \x. E x)}.

ev/app:  eval (app E1 E2)  >-> {eval E1 * cont (app1 E2)}.

ev/app1: retn (lam \x. E x) * cont (app1 E2)
          >-> {eval E2 * cont (app2 \x. E x)}.

ev/app2: retn V2 * cont (app2 \x. E x) >-> {eval (E V2)}.
```

Figure 6.6: A first-order ordered abstract machine semantics for CBV evaluation

The defunctionalization procedure introduces many new atomic propositions. The two predicates introduced in the call-by-value specification were called cont_app1 and cont_app2, and a larger specification will, in general, introduce many more. The one last twist we make is to observe that, instead of introducing a new ordered atomic proposition $\mathsf{cont}\,\bar{t}$ for each iteration of the defunctionalization procedure, it is possible to introduce a single type $(\mathsf{frame} : \mathsf{type})$ and a single atomic proposition $(\mathsf{cont} : \mathsf{frame} \to \mathsf{prop}\,\mathsf{ord})$.

With this change, each iteration of the defunctionalization procedure adds a new constant with type $\Pi y_1{:}\tau_1 \ldots \Pi y_m{:}\tau_m.\,\mathsf{frame}$ to the signature instead of a new atomic proposition with kind $\Pi y_1{:}\tau_1 \ldots \Pi y_m{:}\tau_m.\,\mathsf{prop}\,\mathsf{ord}$. Operationally, these two approaches are equivalent, but it facilitates the addition of control features when we can modularly talk about all of the atomic propositions introduced by defunctionalization as having the form $\mathsf{cont}\,F$ for some term F of type frame.

The ordered abstract machine resulting from this version of defunctionalization and uncurrying is shown in Figure 6.6; this specification can be compared to the one in Figure 6.5.

6.3 Adequacy with abstract machines

The four-rule abstract machine specification given at the beginning of this chapter is adequately represented by the derived SLS specification in Figure 6.6. For terms and for deductive computations, adequacy is a well-understood concept: we know what it means to define an adequate encoding function $\ulcorner e \urcorner = t$ from "on-paper" terms e with (potentially) variables $x_1, \ldots, x_n$ free to LF terms t where $x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp} \vdash t : \mathsf{exp}$, and we know what it means to adequately encode the judgment $e \Downarrow v$ as a negative atomic SLS proposition $\mathsf{ev}\ulcorner e \urcorner \ulcorner v \urcorner$ and to encode derivations of this judgment to SLS terms N where $\cdot;\cdot \vdash_\Sigma N : \langle \mathsf{ev}\ulcorner e \urcorner \ulcorner v \urcorner \rangle$ [HHP93, HL07]. In Section 4.4 we discussed the methodology of adequacy and applied it to the very simple push-down automata from the introduction. In this section, we will repeat this development for Figure 6.6. The gen-

```
value: exp -> prop.
value/lam: value (lam \x. E x).

gen: prop ord.
gen/eval: gen >-> {eval E}.
gen/retn: gen * !value V >-> {retn V}.
gen/cont: gen >-> {gen * cont F}.
```

Figure 6.7: The generative signature Σ_{Gen} describing states Δ that equal $\ulcorner s \urcorner$ for some s

erative signature itself has a slightly different character, but beyond that our discussion closely follows the contours of the adequacy argument from Section 4.4.

Recall the definition of states s, frames f, and stacks k from the beginning of this chapter. Our first step will be to define an interpretation function $\ulcorner s \urcorner = \Delta$ from abstract machine states s to process states Δ so that, for example, the state

$$((\ldots(\mathsf{halt}; \Box\, e_1)\ldots); (\lambda x.e_n)\,\Box) \lhd v$$

is interpreted as the process state

$$y{:}\langle \mathsf{retn}\ulcorner v\urcorner\rangle,\quad x_n{:}\langle \mathsf{cont}\,(\mathsf{app2}\,\lambda x.\ulcorner e_n\urcorner)\rangle,\quad \ldots,\quad x_1{:}\langle \mathsf{cont}\,(\mathsf{app1}\ulcorner e_1\urcorner)\rangle,$$

We also define a generative signature that precisely captures the set of process states in the image of this translation. Having done so, we prove that the property that encoded abstract machine states $\ulcorner s \urcorner \rightsquigarrow^*_{\Sigma_{CBV}} \Delta'$, where Σ_{CBV} stands for the signature in Figure 6.6, only when $\Delta' = \ulcorner s' \urcorner$ for some abstract machine state s'. Then, the main adequacy result, that the interpretation of state s steps to the interpretation of state s' if and only if $s \mapsto s'$, follows by case analysis.

6.3.1 Encoding states

Our first goal is to describe a signature Σgen with the property that if $x{:}\langle \mathsf{gen}\rangle \rightsquigarrow^*_{\Sigma_{Gen}} \Delta$ and $\Delta \not\rightsquigarrow_{\Sigma_{CBV}}$, then Δ encodes an abstract machine state s. A well-formed process state that represents an abstract machine state $((\ldots(\mathsf{halt}; f_1); \ldots); f_n) \rhd e$ has the form

$$y{:}\langle \mathsf{eval}\ulcorner e\urcorner\rangle,\quad x_n{:}\langle \mathsf{cont}\ulcorner f_n\urcorner\rangle,\quad \ldots,\quad x_1{:}\langle \mathsf{cont}\ulcorner f_1\urcorner\rangle$$

where $\ulcorner \Box\, e_2 \urcorner = \mathsf{app1}\ulcorner e_2\urcorner$ and $\ulcorner (\lambda x.e)\,\Box\urcorner = \mathsf{app2}\,(\lambda x.\ulcorner e\urcorner)$. A well-formed process state representing a state $k \lhd v$ has the same form, but with $\mathsf{retn}\ulcorner v\urcorner$ instead of $\mathsf{eval}\ulcorner e\urcorner$. We also stipulate that $k \lhd v$ is only well-formed if v is actually a value – in our current specifications, the only values are functions $\ulcorner \lambda x.e \urcorner = \mathsf{lam}\,\lambda x.\ulcorner e\urcorner$.

The simplest SLS signature that encodes well-formed states has the structure of a context-free grammar like the signature that encoded well-formed PDA states in Figure 4.15. The judgment $\mathsf{value}\ulcorner e\urcorner$ captures the refinement of expressions e that are values. In addition to the four declarations above, the full signature Σgen includes all the type, proposition, and constant declarations from Figure 6.6, but none of the rules.

Note that this specification cannot reasonably be run as a logic program, because the variables E, V and F appear to be invented out of thin air. Rather than traces in these signatures being produced by some concurrent computation (such as forward chaining), they are produced and manipulated by the constructive content of theorems like the ones in this section.

Theorem 6.5 (Encoding)**.** *Up to variable renaming, there is a bijective correspondence between abstract machine states s and process states Δ such that $T :: (x{:}\langle\mathsf{gen}\rangle\ ord) \leadsto^*_{\Sigma_{Gen}} \Delta$ and $\Delta \not\leadsto_{\Sigma_{PDA}}$.*

The proof of Theorem 6.5 follows the structure of Theorem 4.6: we first define context encoding functions $\ulcorner s \urcorner$ and $\ulcorner k \urcorner$.

* $\ulcorner k \rhd e \urcorner = y{:}\langle\mathsf{eval}\ulcorner e\urcorner\rangle,\ \ulcorner k \urcorner$
* $\ulcorner k \lhd v \urcorner = y{:}\langle\mathsf{retn}\ulcorner v\urcorner\rangle,\ \ulcorner k \urcorner$
* $\ulcorner \mathsf{halt} \urcorner = \cdot$
* $\ulcorner k; f \urcorner = x_i{:}\langle\mathsf{cont}\ulcorner f\urcorner\rangle,\ \ulcorner k \urcorner$

It is simple to observe that the encoding function is injective (that $\ulcorner s \urcorner = \ulcorner s' \urcorner$ if and only if $s = s'$), so injectivity boils down to showing that every state s can be generated as a trace $\ulcorner s \urcorner = T :: (x{:}\langle gen\rangle) \leadsto^*_{\Sigma_{Gen}} \ulcorner s \urcorner$. Surjectivity requires us to do induction on the structure of T and case analysis on the first steps in T. Both steps require a lemma that the notion of value expressed by the predicate value in Σ_{Gen} matches the notion of values v used to define well-formed states $k \lhd v$.

6.3.2 Preservation and adequacy

Generated world preservation proceeds as in Section 4.4.3 and Theorem 4.7; this theorem has a form that we will consider further in Chapter 9.

Theorem 6.6 (Preservation)**.** *If $T_1 :: (x{:}\langle\mathsf{gen}\rangle) \leadsto^*_{\Sigma Gen} \Delta_1$, $\Delta_1 \not\leadsto_{\Sigma_{CBV}}$, and $S :: \Delta_1 \leadsto_{\Sigma_{CBV}} \Delta_2$, then $T_2 :: (x{:}\langle\mathsf{gen}\rangle) \leadsto^*_{\Sigma_{Gen}} \Delta_2$.*

The proof proceeds by enumerating the synthetic transitions possible under Σ_{CBV}, performing inversion on the structure of the trace T_1, and using the results to construct the necessary result. This is the most interesting part of the adequacy proof, and a generalization of this preservation proof is carefully considered in Section 9.2 along with a more detailed discussion of the relevant inversion principles. With this property established, the final step is a straightforward enumeration as Theorem 4.8 in Section 4.4.4 was.

Theorem 6.7 (Adequacy of the transition system)**.** $\ulcorner s \urcorner \leadsto_{\Sigma_{CBV}} \ulcorner s' \urcorner$ *if and only if* $s \mapsto s'$.

As in Section 4.4.4, the proof is a straightforward enumeration. An immediate corollary of Theorems 6.5-6.7 is the stronger adequacy result that if $\ulcorner s \urcorner \leadsto_{\Sigma_{CBV}} \Delta$ then $\Delta = \ulcorner s' \urcorner$ for some s' such that $s \mapsto s'$.

$$\frac{[\mathsf{fix}\, x.e/x]e \Downarrow v}{\mathsf{fix}\, x.e \Downarrow v}$$

```
ev/fix:   eval (fix (\x. E x))
           >-> {eval (E (fix (\x. E x)))}.
```

$$\frac{}{\langle\rangle \Downarrow \langle\rangle}$$

```
ev/unit:  eval unit >-> {retn unit}.
```

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{\langle e_1, e_2\rangle \Downarrow \langle v_1, v_2\rangle}$$

```
ev/pair:  eval (pair E1 E2)
           >-> {eval E1 * eval E2 * cont pair1}.
ev/pair1: retn V1 * retn V2 * cont pair1
           >-> {retn (pair V1 V2)}.
```

$$\frac{e \Downarrow \langle v_1, v_2\rangle}{e.1 \Downarrow v_1}$$

```
ev/fst:   eval (fst E) >-> {eval E * cont fst1}.
ev/fst1:  retn (pair V1 V2) * cont fst1 >-> {retn V1}.
```

$$\frac{e \Downarrow \langle v_1, v_2\rangle}{e.2 \Downarrow v_2}$$

```
ev/snd:   eval (snd E) >-> {eval E * cont snd1}.
ev/snd1:  retn (pair V1 V2) * cont snd1 >-> {retn V2}.
```

$$\frac{}{\mathsf{z} \Downarrow \mathsf{z}}$$

```
ev/zero:  eval zero >-> {retn zero}.
```

$$\frac{e \Downarrow v}{\mathsf{s}\, e \Downarrow \mathsf{s}\, v}$$

```
ev/succ:  eval (succ E) >-> {eval E * cont succ1}.
ev/succ1: retn V * cont succ1 >-> {retn (succ V)}.
```

Figure 6.8: Semantics of some pure functional features

6.4 Exploring the image of operationalization

The examples given in the previous section all deal with call-by-value semantics for the untyped lambda calculus, which has the property that any expression will either evaluate forever or will eventually evaluate to a value $\lambda x.e$. We now want to discuss ordered abstract machines with traces that might get *stuck*. One way to raise the possibility of stuck states is to add values besides $\lambda x.e$. In Figure 6.8 we present an extension to Figure 6.6 with some of the features of a pure "Mini-ML" functional programming language: fixed-point recursion ($\ulcorner \mathsf{fix}\, x.e \urcorner = \mathsf{fix}\, \lambda x.\ulcorner e \urcorner$), units and pairs ($\ulcorner \langle\rangle \urcorner = \mathsf{unit}$, $\ulcorner \langle e_1, e_2\rangle \urcorner = \mathsf{pair} \ulcorner e_1 \urcorner \ulcorner e_2 \urcorner$), projections ($\ulcorner e.1 \urcorner = \mathsf{fst} \ulcorner e \urcorner$), $\ulcorner e.2 \urcorner = \mathsf{snd} \ulcorner e \urcorner$), and natural numbers ($\ulcorner \mathsf{z} \urcorner = \mathsf{zero}$, $\ulcorner \mathsf{s}\, e \urcorner = \mathsf{succ} \ulcorner e \urcorner$). The natural semantics is given on the left-hand side of that figure, and the operationalized and defunctionalized ordered abstract machine that arises from (an SLS encoding of) that natural semantics is given on the right.

Note that, facilitated by the nondeterminism inherent in operationalization, we chose parallel evaluation of pairs even though the execution of functions is sequential in Figure 6.6. Traditional abstract machine semantics are syntactic and do not handle parallel evaluation; therefore, it is not possible to show that this ordered abstract machine adequately encodes a traditional abstract machine presentation of Mini-ML. Nevertheless, the SSOS specification as a whole adapts seamlessly to the presence of this new feature.

As we discussed in Chapter 4, when we treat a natural semantics specification as an inductive definition, only the behavior of terminating computations can be observed, and it is not possible to distinguish a non-terminating term like $\mathsf{fix}\, x.x$ from a stuck term like $\mathsf{z}.1$ without relying on

$$\frac{e_1 \Downarrow v}{e_1 \,?\, e_2 \Downarrow v}$$

```
ev/choose1: eval (choose E1 E2)
              >-> {eval E1}.
```

$$\frac{e_2 \Downarrow v}{e_1 \,?\, e_2 \Downarrow v}$$

```
ev/choose2: eval (choose E1 E2)
              >-> {eval E2}.
```

Figure 6.9: Semantics of nondeterministic choice

a characterization of partial proofs. This is one of the problems that Leroy and Grall sought to overcome in their presentation of coinductive big-step operational semantics [LG09]. They defined the judgment $e \Uparrow^\infty$ coinductively; therefore it is easy to express the difference between non-terminating terms ($\mathsf{fix}\, x.x \Uparrow^\infty$) and stuck ones (there is no v such that $\mathsf{z.1} \Downarrow v$ or $\mathsf{z.1} \Uparrow^\infty$).

The translation of natural semantics into ordered abstract machines also allows us to distinguish $\mathsf{fix}\, x.x$ from $\mathsf{z.1}$. The former expression generates a trace that can always be extended:

$$x_1{:}\langle \mathsf{eval}\ (\mathsf{fix}\ \lambda x.x)\rangle \rightsquigarrow x_2{:}\langle \mathsf{eval}\ (\mathsf{fix}\ \lambda x.x)\rangle \rightsquigarrow x_3{:}\langle \mathsf{eval}\ (\mathsf{fix}\ \lambda x.x)\rangle \rightsquigarrow \ldots$$

whereas the latter gets stuck and can make no more transitions:

$$x_1{:}\langle \mathsf{eval}\ (\mathsf{fst}\ \mathsf{zero})\rangle \rightsquigarrow x_2{:}\langle \mathsf{eval}\ \mathsf{zero}\rangle, y{:}\langle \mathsf{cont}\ \mathsf{fst1}\rangle \rightsquigarrow x_3{:}\langle \mathsf{retn}\ \mathsf{zero}\rangle, y{:}\langle \mathsf{cont}\ \mathsf{fst1}\rangle \not\rightsquigarrow$$

Because $(x_3{:}\langle \mathsf{retn}\ \mathsf{zero}\rangle, y{:}\langle \mathsf{cont}\ \mathsf{fst1}\rangle)$ is not a final state – only states consisting of a single $\mathsf{retn}\ulcorner e \urcorner$ proposition are final – this is a stuck state and not a completed computation.

Thus, for deterministic semantics, both coinductive big-step operational semantics and the operationalization transformation represent ways of reasoning about the difference between non-termination and failure in a natural semantics. Our approach has the advantage of being automatic rather than requiring the definition of a new coinductive relation $e \Uparrow^\infty$, though it would presumably be possible to consider synthesizing the definition of $e \Uparrow^\infty$ from the definition of $e \Downarrow v$ by an analogue of our operationalization transformation.

In Section 6.4.1, we discuss the advantages that operationalization has in dealing with nondeterministic language features. These advantages do come with a cost when we consider natural semantics specifications that make deterministic choices, which we discuss in Section 6.4.2.

6.4.1 Arbitrary choice and failure

For the purposes of illustration, we will extend the language of expressions with a nondeterministic choice operator ($\ulcorner e_1 \,?\, e_2 \urcorner = \mathsf{choose}\ulcorner e_1 \urcorner\ulcorner e_2 \urcorner$). The two natural semantics rules for this extension and their (tail-recursion optimized) operationalization are shown in Figure 6.9.

We need to think about the desired semantics of expressions like $(\lambda y.\, y) \,?\, \mathsf{z.1}$ – if the first subexpression $(\lambda y.\, y)$ is chosen for evaluation, then the expression evaluates to a value, but if the second subexpression $\mathsf{z.1}$ is chosen, then the evaluation gets stuck. Small-step intuitions about language safety say that this is a possibility we should be able to express, if only to exclude it with an appropriately designed type system and type safety proof. The ordered abstract machine semantics allows us to produce traces where $x{:}\langle \mathsf{eval}\ulcorner(\lambda y.\, y) \,?\, \mathsf{z.1}\urcorner\rangle \rightsquigarrow^* y{:}\langle \mathsf{eval}\ (\mathsf{lam}\ \lambda y.y)\rangle$ and

where $x{:}\langle \mathsf{eval}\ulcorner(\lambda y.\, y)\ ⁇\ \mathsf{z}.1\urcorner\rangle \rightsquigarrow^* (x'{:}\langle \mathsf{retn}\ \mathsf{zero}\rangle, y{:}\langle \mathsf{cont}\ \mathsf{fst1}\rangle) \not\rightsquigarrow$ as we would hope. Natural semantics specifications (including coinductive big step operational semantics) merely conclude that $(\lambda y.\, y\, y)\ ⁇\ \mathsf{z}.1 \Downarrow (\lambda y.\, y\, y)$. Capturing the stuck behavior in this situation would require defining an extra inductive judgment capturing all the situations where e can get stuck, which is verbose and error-prone [Har12, Section 7.3].

Our ability to reason about evaluations that go wrong is an artifact of the fact that SLS allows us to talk about traces T that represent the process of incomplete proof search in addition to talking about complete proofs. A trace that reaches a state that is not a final $\mathsf{retn}\ulcorner v\urcorner$ state but that cannot step further, like $T :: (x{:}\langle \mathsf{eval}\ulcorner(\lambda y.\, y)\ ⁇\ \mathsf{z}.1\urcorner\rangle) \rightsquigarrow^* (x'{:}\langle \mathsf{retn}\ \mathsf{zero}\rangle, y{:}\langle \mathsf{cont}\ \mathsf{fst1}\rangle)$, corresponds to a point at which backward chaining proof search must backtrack (in a backtracking interpreter) or immediately fail (in a flat resolution interpreter). The trace above corresponds semantically to a failing or going-wrong evaluation, implying that backtracking is not the correct choice. Such an evaluation *ought* to fail, and therefore faithfully capturing the semantics of nondeterministic choice $e_1\ ⁇\ e_2$ with a natural semantics requires us to use a particular operational interpretation of the natural semantics that is based on non-backtracking backward chaining (flat resolution). The operationalization transformation allows us to concretize this particular operational strategy with traces.

6.4.2 Conditionals and factoring

It is great that we're able to reason about nondeterministic specifications in the output of the operationalization transformation! However, a complication arises if we try to encode a Mini-ML feature that was conspicuously missing from Figure 6.8: the elimination form for natural numbers $\ulcorner \mathsf{case}\ e\ \mathsf{of}\ \mathsf{z} \Rightarrow e_z \mid \mathsf{s}\, x \Rightarrow e_s \urcorner = \mathsf{case}\ulcorner e\urcorner\ulcorner e_z\urcorner(\lambda x.\ulcorner e_s\urcorner)$. The usual natural semantics for case analysis look like this:

$$\frac{e \Downarrow \mathsf{z} \quad e_z \Downarrow v}{(\mathsf{case}\ e\ \mathsf{of}\ \mathsf{z} \Rightarrow e_z \mid \mathsf{s}\, x \Rightarrow e_s) \Downarrow v}\ \mathsf{ev/casez} \qquad \frac{e \Downarrow \mathsf{s}\, v' \quad [v'/x]e_s \Downarrow v}{(\mathsf{case}\ e\ \mathsf{of}\ \mathsf{z} \Rightarrow e_z \mid \mathsf{s}\, x \Rightarrow e_s) \Downarrow v}\ \mathsf{ev/cases}$$

If we operationalize this specification directly, we get an ordered abstract machine shown in Figure 6.10 before defunctionalization and in Figure 6.11 after defunctionalization. This specification is nondeterministic much as the specification of $e_1\ ⁇\ e_2$ was: we can evaluate a case expression either with rule ev/casez, which effectively predicts that the answer will be zero, or with rule ev/cases, which effectively predicts that the answer will be the successor of some value. But this means that it is possible to get stuck while executing an intuitively type-safe expression if we predict the wrong branch:

$$\begin{aligned} & x_1{:}\langle \mathsf{eval}\ (\mathsf{case}\ \mathsf{zero}\ e_z\ \lambda x.\, e_s)\rangle \\ \rightsquigarrow\ & x_2{:}\langle \mathsf{eval}\ \mathsf{zero}\rangle, y_2{:}\langle \mathsf{cont}\ (\mathsf{cases}\ \lambda x.\, e_s)\rangle \\ \rightsquigarrow\ & x_3{:}\langle \mathsf{retn}\ \mathsf{zero}\rangle, y_2{:}\langle \mathsf{cont}\ (\mathsf{cases}\ \lambda x.\, e_s)\rangle \\ \not\rightsquigarrow\ & (!!!) \end{aligned}$$

This is a special case of a well-known general problem: in order for us to interpret the usual natural semantics specification (rules ev/casez and ev/cases above) as an operational specification, we need backtracking. With backtracking, if we try to evaluate e using one of the rules and

```
ev/casez: eval (case E Ez (\x. Es x))
           >-> {eval E * (retn zero >-> {eval Ez})}.

ev/cases: eval (case E Ez (\x. Es x))
           >-> {eval E * (All V'. retn (succ V') >-> {eval (Es V')})}.
```

Figure 6.10: Problematic semantics of case analysis (not defunctionalized)

```
ev/casez:  eval (case E Ez (\x. Es x))
            >-> {eval E * cont (casez Ez)}.
ev/casez1: retn zero * cont (casez Ez)
            >-> {eval Ez}.

ev/cases:  eval (case E Ez (\x. Es x))
            >-> {eval E * cont (cases \x. Es x)}.
ev/cases1: retn (succ V) * cont (cases \x. Es x)
            >-> {eval (Es V)}.
```

Figure 6.11: Problematic semantics of case analysis (defunctionalized)

fail, a backtracking semantics means that we will apply the other rule, *re-evaluating* the scrutinee e to a value. Backtracking is therefore necessary for a correct interpretation of the standard rules above, even though it is incompatible with a faithful account of nondeterministic choice! Something must give: we can either give up on interpreting nondeterministic choice correctly or we can change the natural semantics for case analysis. Luckily, the second option is both possible and straightforward.

It is possible to modify the natural semantics for case analysis to avoid backtracking by a transformation called *factoring*. Factoring has been expressed by Poswolsky and Schürmann as a transformation on functional programs in a variant of the Delphin programming language [PS03]. It can also be seen as a generally-correct *logical* transformation on Prolog, λProlog, or Twelf specifications, though this appears to be a folk theorem. We factor this specification by creating a new judgment $(v', e_z, x.e_s) \Downarrow^? v$ that is mutually recursive with the definition of $e \Downarrow v$.

$$\frac{e \Downarrow v' \quad (v', e_z, x.e_s) \Downarrow^? v}{(\mathsf{case}\, e\, \mathsf{of}\, \mathsf{z} \Rightarrow e_z \mid \mathsf{s}\, x \Rightarrow e_s) \Downarrow v}\ \mathsf{ev/case}$$

$$\frac{e_z \Downarrow v}{(\mathsf{z}, e_z, x.e_s) \Downarrow^? v}\ \mathsf{casen/z} \qquad \frac{[v'/x]e_s \Downarrow v}{(\mathsf{s}\, v', e_z, x.e_s) \Downarrow^? v}\ \mathsf{casen/s}$$

This natural semantics specifications is provably equivalent to the previous one we gave when rules are interpreted as inductive definitions. Not only are the same judgments $e \Downarrow v$ derivable, the set of possible derivations are isomorphic, and it is possible to use the existing metatheoretic machinery of Twelf to verify this fact. In fact, the two specifications are also equivalent if we understand natural semantics in terms of the success or failure backtracking proof search, though the factored presentation avoids redundantly re-evaluating the scrutinee. It is only when

```
ev/case:  eval (case E Ez (\x. Es x))
           >-> {eval E *
                (All V'. retn V' >-> {casen V' Ez (\x. Es x)})}.

casen/z:  casen zero Ez (\x. Es x) >-> {eval Ez}.
casen/s:  casen (succ V') Ez (\x. Es x) >-> {eval (Es V')}.
```

Figure 6.12: Revised semantics of case analysis (not defunctionalized)

```
ev/case:  eval (case E Ez (\x. Es x))
           >-> {eval E * cont (case1 Ez (\x. Es x))}.

ev/case1: retn V' * cont (case1 Ez (\x. Es x))
           >-> {casen V' Ez (\x. Es x)}.

casen/z:  casen zero Ez (\x. Es x) >-> {eval Ez}.
casen/s:  casen (succ V') Ez (\x. Es x) >-> {eval (Es V')}.
```

Figure 6.13: Revised semantics of case analysis (defunctionalized)

we interpret the natural semantics specification through the lens of non-backtracking backward chaining (also called flat resolution) that the specifications differ.

The operationalization of these rules is shown in Figure 6.12 before defunctionalization and in Figure 6.13 after defunctionalization. In those figures, the standard evaluation judgment $e \Downarrow v$ is given the now-familiar evaluation and return predicates eval and retn. The judgment $(v', e_z, x.e_s) \Downarrow^? v$ is given the evaluation predicate casen, and *shares* the return predicate retn with the judgment $e \Downarrow v$. This is a new aspect of operationalization. It is critical for us to assign each predicate $\mathsf{a_c}$ uniquely to an evaluation predicate eval_a – without this condition, soundness (Theorem 6.4) would fail to hold. However, we never rely on $\mathsf{a_c}$ being uniquely assigned a return predicate. When return predicates that have the same type are allowed to overlap, it enables the tail-call optimization described in Section 6.1.3 to apply even when the tail call is to a different procedure. This, in turn, greatly simplifies Figures 6.12 and 6.13.

6.4.3 Operationalization and computation

When we described the semantics of nondeterministic choice $e_1 \mathbin{?} e_2$, our operational intuition was to search either for a value such that $e_1 \Downarrow v$ or a value such that $e_2 \Downarrow v$. This implies an operational interpretation of natural semantics as flat resolution as opposed to backward chaining with backtracking. Maintaining this non-backtracking intuition means that some natural semantics specifications, such as those for case analysis, need to be revised. It is a folk theorem that such revisions are always possible by factoring [PS03]; therefore, we can conclude that *in the context of natural semantics specifications* the form of deductive computation (Section 4.6.1) that we are most interested in is flat resolution.

Under the operationalization transformation, traces represent the internal structure of proof

search, and a non-extendable (and non-final) trace represents a situation in which backward chaining search backtracks and where flat resolution search gives up. If we search for a trace $(x{:}\langle \mathsf{eval}\ulcorner e\urcorner\rangle) \leadsto^* (y{:}\langle \mathsf{retn}\ulcorner v\urcorner\rangle)$ in an operationalized specification using committed-choice forward chaining, the operational behavior will coincide with the behavior of flat resolution in the original specification. Alternatively, if we take the exhaustive search interpretation of an operationalized specification and attempt to answer, one way or the other, whether a trace of the form $(x{:}\langle \mathsf{eval}\ulcorner e\urcorner\rangle) \leadsto^* (y{:}\langle \mathsf{retn}\ulcorner v\urcorner\rangle)$ can be constructed, then the operational behavior of the interpreter will coincide with the behavior of backtracking backward chaining in the original specification.

Therefore, operationalization can be said to connect backward chaining in the deductive fragment of SLS to forward chaining in the concurrent fragment of SLS. More precisely, operationalization both connects flat resolution to committed-choice forward chaining and connects backtracking backward chaining to the exhaustive search interpretation of forward chaining.

6.5 Exploring the richer fragment

Work by Danvy et al. on the functional correspondence has generally been concerned with exploring tight correspondences between different styles of specification. However, as we discussed in Section 5.3, one of the main reasons the logical correspondence in SLS is interesting is because, once we translate from a less expressive style (natural semantics) to a more expressive style (ordered abstract machine semantics), we can consider new modular extensions in the more expressive style that were not possible in the less expressive style. As we discussed in Chapter 1, extending a natural semantics with state requires us to revise every existing rule, whereas a SSOS specification can be extended in a modular fashion: we just insert new rules that deal with state. The opportunities for modular extension are part of what distinguishes the logical correspondence we have presented from the work by Hannan and Miller [HM92] and Ager [Age04]. Both of those papers translated natural semantics into a syntactic specification of abstract machines; such specifications are not modularly extensible to the degree that concurrent SLS specifications are.

The ordered abstract machine style of specification facilitates modular extension with features that involve *state* and *parallel evaluation*. We have already seem examples of the latter: the operationalization translation (as extended in Section 6.1.4) can put a natural semantics specification into logical correspondence with either a sequential ordered abstract machine semantics or a parallel ordered abstract machine semantics, and our running example evaluates pairs in parallel. In this section, we will consider some other extensions, focusing on stateful features like mutable storage (Section 6.5.1) and call-by-need evaluation (Section 6.5.2). We will also discuss the semantics of recoverable failure in Section 6.5.4. The presentation of recoverable failure will lead us to consider a point of non-modularity: if we want to extend our language flexibly with non-local control features like recoverable failure, the parallel operationalization translation will make this difficult or impossible. A more modular semantics of parallel evaluation will be presented in Section 7.2.1.

This section will present extensions to the sequential, flat abstract machine for parallel evaluation presented in Figure 6.6. We first presented most of these specifications in [PS09].

```
cell: mutable_loc -> exp -> prop lin.

ev/loc:  eval (loc L)
          >-> {retn (loc L)}.

ev/ref:  eval (ref E)
          >-> {eval E * cont ref1}.
ev/ref1: retn V * cont ref1
          >-> {Exists l. $cell l V * retn (loc l)}.

ev/get:  eval (get E)
          >-> {eval E * cont get1}.
ev/get1: retn (loc L) * cont get1 * $cell L V
          >-> {retn V * $cell L V}.

ev/set:  eval (set E1 E2)
          >-> {eval E1 * cont (set1 E2)}.
ev/set1: retn (loc L) * cont (set1 E2)
          >-> {eval E2 * cont (set2 L)}.
ev/set2: retn V2 * cont (set2 L) * $cell L _
          >-> {retn unit * $cell L V2}.
```

Figure 6.14: Semantics of mutable storage

6.5.1 Mutable storage

Classic stateful programming languages feature mutable storage, which forms the basis of imperative algorithms. We will consider ML-style references, which add four new syntax forms to the language. The first three create ($\ulcorner \mathsf{ref}\, e \urcorner = \mathsf{ref} \ulcorner e \urcorner$), dereference ($\ulcorner !e \urcorner = \mathsf{get} \ulcorner e \urcorner$), and update ($\ulcorner e_1 := e_2 \urcorner = \mathsf{set} \ulcorner e_1 \urcorner \ulcorner e_2 \urcorner$) dynamically allocated cells in the heap. The fourth, $\mathsf{loc}\, l$, is a value that represents pointers to allocated memory. The term l is of a type mutable_loc that has no constructors; locations l can only be allocated at runtime. We also introduce a new *linear* atomic proposition $\mathsf{cell}\, l\, v$ representing a piece of allocated memory (at location l) and its contents (the value v). (Recall from Section 4.5 that, in the ASCII notation for SLS, these linear propositions are written `$cell L V`, as `$` is used as the ASCII representation of the mobile modality ¡A.)

A collection of linear propositions acts much like one of Jeannin and Kozen's *capsules* [JK12], but unlike capsule formulations (and most other existing specification frameworks), we can introduce state in this way without revising any of the rules introduced in the previous section. Without mutable state or parallel computation, specifications such as the one in Figure 6.8 maintain the invariant that the process state Δ is made up of either a $\mathsf{eval}\, e$ proposition or a $\mathsf{retn}\, v$ proposition to the left of some number of $\mathsf{cont}\, f$ propositions. Once we add mutable state, the first-order (or LF) context Ψ, which has been empty in the SSOS semantics we have considered so far, becomes non-empty. We maintain the invariant that the context has one mutable location for each allocated cell:

$$(l_1{:}\mathsf{mutable\_loc}, \ldots, l_n{:}\mathsf{mutable\_loc};\ \ x_1{:}\langle \mathsf{cell}\, l_1\, v_1 \rangle, \ldots, x_n{:}\langle \mathsf{cell}\, l_n\, v_n \rangle, \Delta)$$

where Δ has the form described before. Because the cell $l_i\ v_i$ propositions are mobile, the order of cells is irrelevant, as is the placement of these cells relative to the control structure encoded in the context Δ.

The semantics of mutable storage are presented in Figure 6.14. Rule ev/get in that figure takes an expression get E and evaluates E to a value of the form loc L. After that, rule ev/get1 takes the (unique, by invariant) cell associated with the location L, reads its value, and restores the cell to the context. The synthetic transition associated with ev/get1 is as follows:

$$(\Psi;\ \ \Theta\{\!\{x{:}\langle \mathsf{retn}\ (\mathsf{loc}\ l)\rangle,\ \ y{:}\langle \mathsf{cont}\ \mathsf{get1}\rangle,\ \ z{:}\langle \mathsf{cell}\ l\ v\rangle\}\!\}) \\ \leadsto (\Psi;\ \ \Theta\{w{:}\langle \mathsf{retn}\ v\rangle\ \mathit{ord},\ \ z'{:}\langle \mathsf{cell}\ l\ v\rangle\ \mathit{eph}\})$$

Again, it is critical, when reading this transition, to account for the fact that retn and cont are ordered predicates but cell is a mobile predicate.

The set rules are similar, except that we also evaluate a new value v_2 and restore that value to the process state instead of the value previously contained in the cell. We mention cell l _ in the premise of ev/set2 only to consume the old cell associated with l before we replace it with something new. If multiple parts of a process state are trying to consume the same resource in order to set the value of a cell concurrently, we have a race condition; this possibility is discussed below.

Finally, the ref rules evaluate the subexpression to a value v and then, in rule ev/ref1, allocate a new cell to hold that value. This new cell, according to our context invariant, needs to be associated with a new variable l, which we generate with existential quantification in the head of the ev/ref1 rule. The synthetic transition associated with ev/ref1 therefore has an extended first-order context after the transition:

$$(\Psi;\ \ \Theta\{\!\{x{:}\langle \mathsf{retn}\ v\rangle,\ \ y{:}\langle \mathsf{cont}\ \mathsf{ref1}\rangle\}\!\}) \\ \leadsto (\Psi, l{:}\mathsf{mutable\_loc};\ \ \Theta\{w{:}\langle \mathsf{retn}\ (\mathsf{loc}\ l)\rangle\ \mathit{ord},\ \ z{:}\langle \mathsf{cell}\ l\ v\rangle\ \mathit{eph}\})$$

Existential angst

Our semantics of mutable storage uses existential quantification as a symbol generator to conjure up new locations. However, it is important to remember that LF variables in Ψ are defined by substitution, so if there is a step $(\Psi, l_1{:}\mathsf{loc}, l_2{:}\mathsf{loc}; \Delta) \leadsto (\Psi, l_1{:}\mathsf{loc}, l_2{:}\mathsf{loc}; \Delta')$, it must also be the case that $(\Psi, l_1{:}\mathsf{loc}; [l_1/l_2]\Delta) \leadsto (\Psi, l_1{:}\mathsf{loc}; [l_1/l_2]\Delta')$. Therefore, there can be no SLS proposition or synthetic transition that holds only if two variables are distinct, since by definition the same proposition or synthetic transition would hold when we unified the two variables. This, in turn, means our specification of Mini-ML with mutable references *cannot* be further extended to include the tests for reference equality that languages like Standard ML or OCaml have, since locations are described only as variables.

One workaround to this problem is to maintain an association between distinct variables l and distinct concrete terms (usually distinct natural numbers) in the process state. It is possible to use generative signatures to enforce that all well-formed states associate distinct variables with distinct concrete terms, as described in Section 9.4.4. In such a specification, we can use inequality of the concrete terms as a proxy for inequality of the variables. It will still be the case

that unifying distinct variables preserves transitions, but we can ensure that any process state obtained by unifying distinct variables is not well-formed according to the generative invariant.

I believe that a substructural treatment of nominal quantification could be incorporated into SLS and would allow for locations to be handled in a more satisfying way along the lines of proposals by Cheney and Harper [Che12, Har12]. This extension to the SLS framework is beyond the scope of this thesis, however. Luckily, aside from being unable to elegantly represent tests for pointer inequality or the entirety of Harper's Modernized Algol [Har12, Chapter 35], we will not miss name generation facilities much in the context of this thesis. One of the most important use cases of name generation and nominal abstraction is in reasoning *about* logical specifications within a uniform logic [GMN11], and this thesis does not consider a uniform *meta*logic for SLS specifications.

Race conditions

Because ordered abstract machine semantics allow us to add both state and parallelism to specifications, the issue of race conditions, which arise whenever there is both concurrency and state, should be briefly addressed. Fundamentally, SLS and SSOS specifications have no notion of atomicity beyond the one provided by focusing and synthetic inference rules, and so race conditions can arise.

$$
\begin{array}{ll}
(l : \mathsf{mutable\_loc};\ & x_1{:}\langle \mathsf{cell}\ l\ \mathsf{zero}\rangle\ \mathit{eph},\\
& x_2{:}\langle \mathsf{retn}\ (\mathsf{succ}\ \mathsf{zero})\rangle\ \mathit{ord},\quad x_3{:}\langle \mathsf{cont}\ (\mathsf{set2}\ l)\rangle\ \mathit{ord},\\
& x_4{:}\langle \mathsf{retn}\ (\mathsf{succ}\ (\mathsf{succ}\ \mathsf{zero}))\rangle\ \mathit{ord},\quad x_5{:}\langle \mathsf{cont}\ (\mathsf{set2}\ l)\rangle\ \mathit{ord},\\
& x_6{:}\langle \mathsf{cont}\ \mathsf{pair1}\rangle\ \mathit{ord})
\end{array}
$$

Figure 6.15: A racy process state

A process state that starts out containing only $\mathsf{eval}\ulcorner(\lambda x.\langle \mathsf{set}\ x\ (\mathsf{s}\ \mathsf{z}), \mathsf{set}\ x\ (\mathsf{s}(\mathsf{s}\ \mathsf{z}))\rangle)(\mathsf{ref}\ \mathsf{z})\urcorner$, for example, can evaluate to the process state in Figure 6.15. Two different transitions out of this state can both manipulate the data associated with the mutable location l – this is a race condition. Reasoning about race conditions (and possibly precluding them from well-formed specifications) is not within the scope of this thesis. The applicability of generative invariants discussed in Chapter 9 to race conditions is, however, certainly an interesting topic for future work. If we set up the semantics such that a situation like the one above could nondeterministically transition to an ill-formed error state, then it would not be possible to prove the preservation of the generative invariant unless the well-formedness criteria expressed by that invariant precluded the existence of race conditions.

Let us consider what happens when we operationalize a natural semantics that uses state. Recall that the addition of an imperative counter to the natural semantics for semantics for CBV evaluation was presented as a non-modular extension, because we had to revise the existing rules

for functions and application as follows:

$$\frac{}{(\lambda x.e, \underline{n}) \Downarrow (\lambda x.e, \underline{n})} \qquad \frac{(e_1, \underline{n}) \Downarrow (\lambda x.e, \underline{n}_1) \quad (e_2, \underline{n}_1) \Downarrow (v_2, \underline{n}_2) \quad ([v_2/x]e_2, \underline{n}_2) \Downarrow (v, \underline{n'})}{(e_1\, e_2, \underline{n}) \Downarrow (v, \underline{n'})}$$

In the pure CBV specification, the two premises $e_1 \Downarrow \lambda x.e$ and $e_2 \Downarrow v_2$ could be treated as independent and could be made parallel by the operationalization transformation, but the two premises $(e_1, \underline{n}) \Downarrow (\lambda x.e, \underline{n}_1)$ and $(e_2, \underline{n}_1) \Downarrow (v_2, \underline{n}_2)$ are not: the first premise binds $\underline{n}_1$, which appears as an input argument to the second premise. Therefore, from the perspective of operationalization, parallel evaluation and state are simply incompatible: having state in the original specification will preclude parallel evaluation (and therefore race conditions) in the operationalized specification.

Ordered abstract machine semantics allow for the modular composition of mutable storage and parallel evaluation, in that the original specifications can be simply composed to give a semantically meaningful result. However, composing mutable storage and parallel evaluation leads to the possibility of race conditions (which can be represented in SLS), indicating that this composition is not always a good idea if we want to avoid race conditions. Adding state to a natural semantics specification, on the other hand, will force operationalization to produce an ordered abstract machine without parallelism.

6.5.2 Call-by-need evaluation

Mutable references were an obvious use of ambient state, and we were able to extend the ordered abstract machine obtained from the operationalization transformation by simply adding new rules for mutable references (though this did introduce the possibility of race conditions). Another completely modular extension to our (now stateful) Mini-ML language is *call-by-need* evaluation. The basic idea in call-by-need evaluation is that an expression is not evaluated eagerly; rather, instead, it is stored until the value of that expression is demanded. Once a value is needed, it is computed and the value of that computation is memoized; therefore, a suspended expression will be computed at most once.

This section considers two rather different implementations of by-need evaluation: the first, recursive suspensions, presents itself to the programmer as a different sort of fixed-point operator, and the second, lazy call-by-need, presents itself to the programmer as a different sort of function. Both approaches to lazy evaluation are based on Harper's presentation [Har12, Chapter 37].

Recursive suspensions

Recursive suspensions (Figure 6.16) replace the fixed-point operator $\mathsf{fix}\, x.e$ with a thunked expression $\ulcorner \mathsf{thunk}\, x.e \urcorner = \mathsf{thunk}\, (\lambda x.\ulcorner e \urcorner)$. Whereas the fixed-point operator returns a value (or fails to terminate), thunked expressions always immediately return a value $\mathsf{issusp}\, l$, where l is a location of type bind_loc. This location is initially associated with a linear atomic proposition $\mathsf{susp}\, l\, (\lambda x.\ulcorner e \urcorner)$ (rule ev/thunk).

When we apply the force operator to an expression that returns $\mathsf{issusp}\, l$ for the first time, the location l stops being associated with a linear atomic proposition of the form $\mathsf{susp}\, l\, (\lambda x.\ulcorner e \urcorner)$ and

```
susp: bind_loc -> (exp -> exp) -> prop lin.
blackhole: bind_loc -> prop lin.
bind: bind_loc -> exp -> prop pers.

ev/susp:    eval (issusp L) >-> {retn (issusp L)}.

ev/thunk:   eval (thunk \x. E x)
             >-> {Exists l. $susp l (\x. E x) * retn (issusp l)}.

ev/force:   eval (force E)
             >-> {eval E * cont force1}.

ev/force1a: retn (issusp L) * cont force1 * $susp L (\x. E' x)
             >-> {eval (E' (issusp L)) * cont (bind1 L) *
                  $blackhole L}.

ev/force2a: retn V * cont (bind1 L) * $blackhole L
             >-> {retn V * !bind L V}.

#| STUCK -  retn (issusp L) * cont force1 * $blackhole L >-> ??? |#

ev/force1b: retn (issusp L) * cont force1 * !bind L V
             >-> {retn V}.
```

Figure 6.16: Semantics of call-by-need recursive suspensions

becomes associated with a linear atomic proposition of the form blackhole l (rule ev/force1a). This blackhole l proposition can be used to detect when an expression tries to directly reference its own value in the process of computing to a value. In this example, such a computation will end up stuck, but the comparison with fix $x.x$ suggests that the semantically correct option is failing to terminate instead. A rule with the premise retn (issusp l) • cont force1 • blackhole l could instead be used to loop endlessly or signal failure. This possibility is represented in Figure 6.16 by the commented-out rule fragment – `#| this is comment syntax |#`.

Once a suspended expression has been fully evaluated (rule ev/force2b), the black hole is removed and the location l is persistently associated with the value v; future attempts to force the same suspended expression will trigger rule ev/force1b instead of ev/force1a and will immediately return the memoized value.

The last four rules in Figure 6.16 (and the one commented-out rule fragment) are all part of one multi-stage protocol. It may be enlightening to consider the *re*functionalization of Figure 6.16 presented in Figure 6.17. This rule has a conjunctive continuation (using the additive conjunction connective $A^- \mathbin{\&} B^-$) with one conjunct for two of the three atomic propositions a bind_loc can be associated with: the linear proposition susp l $(\lambda x.e)$, the linear proposition blackhole l (which cannot be handled by the continuation and so will result in a stuck state), and the persistent proposition bind l v.

```
ev/force: eval (force E)
           >-> {eval E *
                ((All L. All E'.
                  retn (issusp L) * $susp L (\x. E' x)
                   >-> {eval (E' (issusp L)) * $blackhole L *
                        (All V. retn V * $blackhole L
                          >-> {retn V * !bind L V})})
     #| STUCK -  & (All L. retn (issusp L) * $blackhole L >-> ???) |#
                & (All L. All V.
                   retn (issusp L) * !bind L V >-> {retn V}))}.
```

Figure 6.17: Semantics of call-by-need recursive suspensions, refunctionalized

```
susp': exp -> exp -> prop lin.
blackhole': exp -> prop lin.
bind': exp -> exp -> prop pers.

ev/lazylam: eval (lazylam \x. E x) >-> {retn (lazylam \x. E x)}.

ev/applazy: retn (lazylam \x. E x) * cont (app1 E2)
             >-> {Exists x:exp. eval (E x) * $susp' x E2}.

ev/susp':   eval X * susp' X E
             >-> {$blackhole' X * eval E * cont (bind1' E)}.

ev/susp1':  retn V * cont (bind1' X) * $blackhole' X
             >-> {retn V * !bind' X V}.

ev/bind':   eval X * !bind' X V >-> {retn V}.
```

Figure 6.18: Semantics of lazy call-by-need functions

Lazy evaluation

Recursive suspensions must be forced explicitly; an alternative to recursive suspensions, which uses very similar specification machinery but that presents a different interface to the programmer, is lazy call-by-need function evaluation. Lazy evaluation better matches the semantics of popular call-by-need languages like Haskell. For this semantics, we will not create a new abstract location type like mutable_loc or bind_loc; instead, we will associate suspended expressions (and black holes and memoized values) with free expression variables of type exp.

We can treat lazy call-by-need functions $(\mathsf{lazylam}\ \lambda x.e)$ as an extension to the language that already includes call-by-value functions $(\mathsf{lam}\ \lambda x.e)$ and application; this extension is described in Figure 6.18. Lazy functions are values (rule ev/lazylam), but when a lazy function is returned to a frame $\ulcorner \Box\, e_2 \urcorner = \mathsf{app1} \ulcorner e_2 \urcorner$, we do not evaluate $\ulcorner e_2 \urcorner$ to a value immediately. Instead, we create a free variable x of type exp and substitute that into the lazy function.

Free variables x are now part of the language of expressions that get evaluated, though they

```
ev/envlam:  eval (envlam \x. E x) >-> {retn (envlam \x. E x)}.

ev/appenv1: retn (envlam \x. E x) * cont (app1 E2)
             >-> {Exists x. eval E2 * cont (app2' x (E x))}.

ev/appenv2: retn V2 * cont (app2' X E)
             >-> {eval E * !bind' X V2}.
```

Figure 6.19: Environment semantics for call-by-value functions

are not in the language of values that get returned. Therefore, we need some way of evaluating free variables. This is handled by the three rules ev/susp′, ev/susp1′, and ev/bind′ as before. Each free expression variable x is either associated with a unique linear atomic proposition susp′ $x\, e_2$, a black hole blackhole′ x, or a persistent binding bind′ $x\, v$.

As a final note, call-by-need evaluation is semantically equivalent to call-by-name evaluation in a language that does not otherwise use state. Unevaluated suspensions can therefore be ignored if they are not needed. For this reason, if SLS were extended with an affine modality, it would be quite reasonable to view susp and susp′ as affine atomic propositions instead of linear atomic propositions. However, as long as we restrict ourselves to considering traces rather than complete derivations, the differences between affine and linear propositions are irrelevant.

6.5.3 Environment semantics

Toninho, Caires, and Pfenning have observed that call-by-need and call-by-value can both be seen in a larger family of *sharing* evaluation strategies (if and when the argument to $\lambda x.e$ is evaluated, the work of evaluating that argument to a value is shared across all occurrences of x). Call-by-name, in contrast, is called a *copying* evaluation strategy, since the unevaluated argument of $\lambda x.e$ is copied to all occurrences of x [TCP12]. This relationship between the lazy call-by-need semantics from Figure 6.18 and call-by-value is better presented by giving a variant of what we called an *environment semantics* in [PS09].

As with the lazy call-by-name semantics, we introduce the environment semantics by creating a new function value envlam $(\lambda x.\, e)$ instead of reinterpreting the existing function expression lam $(\lambda x.\, e)$. When a value of the form lazylam $(\lambda x.\, e)$ was returned to a frame app1 e_2 in rule ev/applazy from Figure 6.18, we immediately created a new expression variable x, suspended the argument e_2, and scheduled the function body for evaluation. When a value of the form envlam $(\lambda x.\, e)$ is returned to a frame app1 e_2 frame in rule ev/appenv1 in Figure 6.19, we likewise create the new expression variable x, but we suspend the *function body* in a frame app2′ $x\, e$ that also records the new expression variable x and schedule the *argument* for evaluation. Immediately evaluating the argument is, of course, exactly how call-by-value evaluation is performed; this is what makes environment semantics equivalent to call-by-value semantics. Then, when the evaluated function argument v_2 is returned to that frame (rule ev/appenv2), we create the same persistent binding bind $x\, v_2$ that was generated by rule ev/susp1′ in Figure 6.18 and proceed to evaluate the function body. Upon encountering the free variable x in the course of evaluation, the same rule ev/bind′ from Figure 6.18 will return the right value.

```
error: prop ord.
handle: exp -> prop ord.

ev/fail:   eval fail >-> {error}.
ev/error:  error * cont F >-> {error}.

ev/catch:  eval (catch E1 E2) >-> {eval E1 * handle E2}.
ev/catcha: retn V * handle _ >-> {retn V}.
ev/catchb: error * handle E2 >-> {eval E2}.
```

Figure 6.20: Semantics of recoverable failure

This presentation of the environment semantics is designed to look like call-by-need, and so it creates the free variable x early, in rule ev/appenv1. It would be equally reasonable to create the free variable x later, in rule ev/appenv2, which would result in a specification that resembles Figure 6.6 more closely. This is what was done in [PS09] and [SP11], and we will use a similar specification (Figure 8.3) as the basis for deriving a control flow analysis in Section 8.4.

6.5.4 Recoverable failure

In a standard abstract machine presentation, recoverable failure can be introduced by adding a new state $s = k\blacktriangleleft$ to the existing two ($s = k \rhd e$ and $s = k \lhd v$) [Har12, Chapter 28]. Whereas $k \lhd v$ represents a value being returned to the stack, $k\blacktriangleleft$ represents *failure* being returned to the stack; failure is signaled by the expression fail[7] and can be handled by the expression $\ulcorner \mathsf{try}\ e_1\ \mathsf{ow}\ e_2 \urcorner = \mathsf{catch} \ulcorner e_1 \urcorner \ulcorner e_2 \urcorner$.

We can extend *sequential* ordered abstract machines with exceptions in a modular way, as shown in Figure 6.20. Recall that a sequential ordered abstract machine specification is one where there is only one ordered $\mathsf{eval}\, e$ or $\mathsf{retn}\, v$ proposition in the process state to the right of a series of ordered $\mathsf{cont}\, f$ propositions – process states with $\mathsf{eval} \ulcorner e \urcorner$ correspond to states $k \rhd e$ and process states with $\mathsf{retn} \ulcorner v \urcorner$ correspond to states $k \lhd v$.

We introduce two new ordered atomic propositions. The first, error, is introduced by rule ev/fail. A state with an error proposition corresponds to a state $k\blacktriangleleft$ in traditional ordered abstract machine specifications. Errors eat away at any $\mathsf{cont}\, f$ propositions to their right (rule ev/error). The only thing that stops the inexorable march of an error is the special ordered atomic proposition $\mathsf{handle}\, e$ that is introduced in rule ev/catch when we evaluate the handler.

This is one case where the use of defunctionalized specifications – and, in particular, our decision to defunctionalize with a single $\mathsf{cont}\, f$ proposition instead of inventing a new ordered atomic proposition at every step (Section 6.2.3) – gives us a lot of expressive power. If we wanted to add exceptions to the higher-order specification of Mini-ML, we would have to include the possibility of an exceptional outcome in every individual rule. For instance, this would be the rule for evaluating $\ulcorner \mathsf{s}\, e \urcorner = \mathsf{succ} \ulcorner e \urcorner$:

[7] Failure could also be introduced by actions like as dividing by zero or encountering the black hole in the commented-out case of Figure 6.16.

```
ev/succ: eval (succ E)
          >-> {eval E *
               ((All V. retn V >-> {retn (succ V)})
                & (error >-> {error}))}.
```

Instead, this case is handled generically by rule ev/error in Figure 6.20, though the above specification is what we would get if we were to refunctionalize the defunctionalized specification of core Mini-ML from Figure 6.8 extended with rule ev/error.

Failures and the parallel translation

Our semantics of recoverable failure composes reasonably well with stateful features, though arguably in call-by-need evaluation it is undesirable that forcing a thunk can lead to errors being raised in a non-local fashion. However, recoverable failure does not compose well with parallel semantics as we have described them.

We assume, in rule ev/error, that we can blindly eliminate cont f frames with the ev/error rule. If we eliminate the cont pair1 frame from Figure 6.8 in this way, it breaks the invariant that the ordered propositions represent a branching tree written down in postfix. Recall that, without exceptions, a piece of process state in the process of evaluating $\ulcorner\langle e_1, e_2\rangle\urcorner = \mathsf{pair}\ulcorner e_1\urcorner\ulcorner e_2\urcorner$ has the following form:

$$(\text{subgoal: evaluating } e_1), (\text{subgoal: evaluating } e_2), y{:}\langle \mathsf{cont}\ \mathsf{pair1}\rangle$$

If the second subgoal evaluating e_2 signals an error, that error will immediately propagate to the right, orphaning the first subgoal. Conversely, if the first subgoal signals an error, that error will have to wait until the first subgoal completes: SLS specifications are local, and there is no local way for the first subgoal to talk about its continuation $y{:}\langle \mathsf{cont}\ \mathsf{pair1}\rangle$ because an arbitrary amount of stuff (the representation of the second subgoal) is in the way. This seems to force us into treating parallel evaluations asymmetrically: if e_{raise} signals failure and e_{loop} loops forever, then the two Mini-ML pair expressions $\langle e_{raise}, e_{loop}\rangle$ and $\langle e_{loop}, e_{raise}\rangle$ are observably different. That is arguably bad, though if we switch the relative position of e_1 and e_2 in the context, it can also be seen as an implementation of the sequential semantics for exceptions followed by Manticore [FRR08].

A fix is to modify defunctionalization to group similar propositions rather than grouping all propositions into the single proposition cont. Specifically, we defunctionalize *sequential* propositions of the form $\forall\overline{x}.\ \mathsf{retn}\ v \rightarrowtail \{\ldots\}$ using one ordered atomic proposition cont f and defunctionalize *parallel* propositions of the form $\forall\overline{x}.\ \mathsf{retn}\ v_1 \bullet \mathsf{retn}\ v_2 \rightarrowtail \{\ldots\}$ using a different ordered atomic proposition cont2 f. This lets us write rules that treat parallel continuations generically and that only return errors when *both* sub-computations have completed and at least one has signaled an error:

```
ev/errerr:  error * error * cont2 _ >-> {error}.
ev/errretn: error * retn _ * cont2 _ >-> {error}.
ev/retnerr: retn _ * error * cont2 _ >-> {error}.
```

This is a big improvement, because parallel pairs are again treated symmetrically. But it's not the way we necessarily wanted to restore symmetry: the evaluation $\langle e_{raise}, e_{loop}\rangle$ and $\langle e_{loop}, e_{raise}\rangle$

will both loop forever, but we might wish for both of them to signal failure. The latter alternative is not expressible in an ordered abstract machine specification.

Part of the problem is that recoverable failure is fundamentally a *control* feature and not a stateful or parallel programming language feature. As a result, it is not easy to handle at the level of ordered abstract machines, because ordered abstract machines do not give the specification author enough access to the control structure. The destination-passing style we consider in the next chapter, on the other hand, will give us sufficient access to control structure.

6.5.5 Looking back at natural semantics

Mutable storage, call-by-need evaluation, and the environment semantics are all modular extensions to the call-by-value specification in Figure 6.6. The extensions are modular because they make essential use of the ambient context available to concurrent SLS specifications, introducing new linear and persistent ordered atomic propositions that can be added to the context (and, in the linear case, removed as well).

For extensions to sequential ordered abstract machines that are only based on extending the state, we can consider what it would mean to reverse-engineer a natural semantics formalism that is as extensible as the resulting ordered abstract machine. The primary judgment of such a specification is not $e \Downarrow v$ as before; rather, the primary judgment becomes $\{e \| \mu\}_\Psi \Downarrow \{v \| \mu\}_{\Psi'}$.[8] The variable contexts Ψ and Ψ' are the same variable contexts that appear in our process states $(\Psi; \Delta)$ and our specifications are expected to maintain the invariant that $\Psi \subseteq \Psi'$. The objects e and v remain syntactic objects adequately encodable in LF, as before, whereas μ is an extensible bag of judgments $\mu = J_1 \otimes \ldots \otimes J_n$ that correspond to the propositions in our linear and persistent context; we treat $\otimes$ as an associative and commutative operator (just like conjunction of linear logic contexts). A new judgment in an SLS specification can be treated as a new member of the syntactic class J. For instance, lazy call-by-need functions as defined in Figure 6.18 use three judgments: $x \hookleftarrow e$ (corresponding to $\mathsf{susp}'\, x\, e$), $x \hookleftarrow \bullet$ (corresponding to $\mathsf{blackhole}'\, x$), and $x \rightarrowtriangle v$ (corresponding to $\mathsf{bind}'\, x\, v$). We can give a statefully-modular natural semantics for call-by-need lazy functions as follows:

$$\frac{}{\{\lambda x.e \| \mu\}_\Psi \Downarrow \{\lambda x.e \| \mu\}_\Psi}$$

$$\frac{\{e_1 \| \mu\}_\Psi \Downarrow \{\lambda x.e \| \mu'\}_{\Psi'} \quad \{e \| x \hookleftarrow e_2 \otimes \mu'\}_{\Psi',x} \Downarrow \{v \| \mu''\}_{\Psi''}}{\{e_1\, e_2 \| \mu\}_\Psi \Downarrow \{v \| \mu''\}_{\Psi''}}$$

$$\frac{\{e \| x \hookleftarrow \bullet \otimes \mu\}_{\Psi,x} \Downarrow \{v \| x \hookleftarrow \bullet \otimes \mu'\}_{\Psi',x}}{\{x \| x \hookleftarrow e \otimes \mu\}_{\Psi,x} \Downarrow \{v \| x \rightarrowtriangle v \otimes \mu'\}_{\Psi',x}}$$

$$\frac{}{\{x \| x \rightarrowtriangle v \otimes \mu\}_{\Psi,x} \Downarrow \{v \| x \rightarrowtriangle v \otimes \mu\}_{\Psi,x}}$$

While the definition of $e \Downarrow v$ could be directly encoded as a deductive SLS specification, the definition of $\{e \| \mu\}_\Psi \Downarrow \{v \| \mu'\}_{\Psi'}$ cannot. Nevertheless, the example above suggests that

[8] The notation $\{e \| \mu\}_\Psi$ is intended to evoke Harper's notation $\nu\Sigma\{e \| \mu\}$, which is used to describe mutable references and lazy evaluation in Chapters 36 and 37 of [Har12]. The critical semantic distinction is that our Ψ contains variables whereas Σ contains proper symbols that are not available in SLS, as discussed in Section 6.5.1.

```
#mode inc + -.
inc: nat -> nat -> prop.

inc/eps: inc eps (c eps b1).
inc/b0: inc (c N b0) (c N b1).
inc/b1: inc (c N b1) (c R b0) <- inc N R.

#mode plus + + -.
plus: nat -> nat -> nat -> prop.
plus/eN: plus eps N N.
plus/Ne: plus N eps N.
plus/b00: plus (c M b0) (c N b0) (c R b0) <- plus M N R.
plus/b01: plus (c M b0) (c N b1) (c R b1) <- plus M N R.
plus/b10: plus (c M b1) (c N b0) (c R b1) <- plus M N R.
plus/b11: plus (c M b1) (c N b1) (c R b0) <- plus M N K <- inc K R.
```

Figure 6.21: Backward-chaining logic program for binary addition

a carefully-defined formalism for statefully-modular natural semantics specifications could be similarly compiled into (or defined in terms of) the operationalization of specifications into SLS.

There is a great deal of work on special-purpose formalisms for the specification and modular extension of operational semantics. The specification above follows Harper's development in [Har12], and Mosses's Modular Structural Operational Semantics (MSOS) is a similar development [Mos04]. Previous work is primarily interested in the modular extension of small-step structural operational semantics specifications rather than big-step natural semantics, though Mosses does discuss the latter. The operationalization transformation applies to SOS specifications (as discussed below in Section 6.6.2), but the result is something besides an ordered abstract machine semantics.

The functional correspondence connects structural operational semantics and abstract machines [Dan08]. A logical correspondence between SOS specifications and ordered abstract machines in SLS might give us insight into a modular formalism for SOS that is defined in terms of concurrent SLS specifications, but this is left for future work.

6.6 Other applications of transformation

Thus far, we have only discussed the application of the operationalization and defunctionalization transformations to natural semantics specifications. However, both transformations are general and can be applied to many different specifications.

In this section, we will consider the meaning of operationalization on three other types of deductive SLS specifications: an algorithmic specification of addition by Pfenning, small-step structural operational semantics specifications, and the natural semantics of Davies' staged computation language $\lambda^{\bigcirc}$. The last two transformations explore the use of *partial* operationalization in which we use the generality of operationalization to transform only some of the predicates in a program.

```
inc: nat -> prop ord.
plus: nat -> nat -> prop ord.
retn: nat -> prop ord.
cont: frame -> prop ord.

inc/eps:    inc eps >-> {retn (c eps b1)}.
inc/b0:     inc (c N b0) >-> {retn (c N b1)}.
inc/b1:     inc (c N b1) >-> {inc N * cont append0}.

plus/eN:    plus eps N >-> {retn N}.
plus/Ne:    plus N eps >-> {retn N}.

plus/b00:   plus (c M b0) (c N b0) >-> {plus M N * cont append0}.
plus/b01:   plus (c M b0) (c N b1) >-> {plus M N * cont append1}.
plus/b10:   plus (c M b1) (c N b0) >-> {plus M N * cont append1}.
plus/b11:   plus (c M b1) (c N b1) >-> {plus M N * cont carry}.
plus/carry: retn K * cont carry >-> {inc K * cont append0}.

cont/0:     retn R * cont append0 >-> {retn (c R b0)}.
cont/1:     retn R * cont append1 >-> {retn (c R b1)}.
```

Figure 6.22: Forward-chaining logic program for binary addition

6.6.1 Binary addition

In the notes for his Spring 2012 course on Linear Logic, Pfenning gave two algorithmic specifications of binary addition as logic programs; in both cases, binary numbers are represented as lists of bits, either $\ulcorner \epsilon \urcorner = \mathsf{eps}$, $\ulcorner n0 \urcorner = \mathsf{c} \ulcorner n \urcorner \mathsf{b0}$, or $\ulcorner n1 \urcorner = \mathsf{c} \ulcorner n \urcorner \mathsf{b1}$. The first specification was given as a forward-chaining ordered abstract machine [Pfe12b], and the second specification was given as a backward chaining logic program [Pfe12a]. Because these operational specifications were developed independently of operationalization, this provides an interesting and relatively simple test-case for operationalization, defunctionalization, and their implementation in the SLS prototype.

The backward-chaining logic program for binary addition is presented in Figure 6.21; the three-place relation plus depends on a two-place relation inc that handles carry bits. We operationalize this specification by giving plus and inc the evaluation predicates plus and inc, respectively, and giving them the same return predicate, retn.

In the direct operationalization of Figure 6.21, we can observe that there are three separate continuation frames (associated with the rules inc/b1, plus/b00, and plus/b11) that do the exact same thing: cause the bit 0 to be appended to the end of the returned number. With this observation, we can consolidate these three frames and the rules associated with them into one frame append0 and one rule cont/0 in Figure 6.22. Similarly, continuation frames associated with rules plus/b01 and plus/b10 both append the bit 1, and can be consolidated into the frame append1 and the rule cont/1 in Figure 6.22. (The only remaining frame is associated with the rule plus/b11 and invokes the increment procedure inc to handle the carry bit.) With the exception of the cre-

```
#mode value +.
value: exp -> prop.
value/lam: value (lam \x. E x).

#mode step + -.
step: exp -> exp -> prop.

step/app1:   step (app E1 E2) (app E1' E2)
              <- step E1 E1'.

step/app2:   step (app E1 E2) (app E1 E2') <- value E1
              <- step E2 E2'.

step/appred: step (app (lam \x. E x) V) (E V) <- value V.

#mode evsos + -.
evsos: exp -> exp -> prop.
evsos/steps: evsos E V <- step E E' <- evsos E' V.
evsos/value: evsos V V <- value V.
```

Figure 6.23: SOS evaluation

ation of redundant continuations, which could certainly be addressed by giving a more robust implementation of defunctionalization, Figure 6.22 can be seen as a direct operationalization of the deductive procedure in Figure 6.21.

Unfortunately, Figure 6.22 is not quite the same as Pfenning's ordered abstract machine for addition in [Pfe12b], but the difference is rather minor. In Pfenning's version of addition, the rule we call plus/carry in Figure 6.22 does not generate the conclusion cont append0. Instead, that frame is generated earlier by the rule we called plus/b11, which in Pfenning's formulation is $\forall M.\, \forall N.\, \mathsf{plus}\ (\mathsf{c}\ M\ \mathsf{b1})\ (\mathsf{c}\ N\ \mathsf{b1}) \rightarrowtail \{\mathsf{plus}\ M\ N \bullet \mathsf{cont}\ \mathsf{carry} \bullet \mathsf{cont}\ \mathsf{append0}\}$.

Relating specifications that differ only in the order with which certain continuation frames are generated seems to be a pervasive pattern. For example, Ian Zerny observed a very similar phenomenon when using operationalization to replay the correspondence between natural semantics and abstract machines presented in [DMMZ12]. Characterizing this observation more precisely is left for future work.

6.6.2 Operationalizing SOS specifications

We have thus far considered big-step operational semantics and abstract machines, mostly neglecting another great tradition in operational semantics, *structural operational semantics* (SOS) specifications [Plo04], though we did define the small-step judgment $\ulcorner e \mapsto e' \urcorner = \mathsf{step} \ulcorner e \urcorner \ulcorner e' \urcorner$ for call-by-value evaluation in the beginning of this chapter. The SOS specification from that discussion is encoded as an SLS specification in Figure 6.23. The figure also defines the judgment $\ulcorner e_1 \mapsto^* v \urcorner = \mathsf{evsos} \ulcorner e \urcorner \ulcorner v \urcorner$ that implements big-step evaluation in terms of the small-step SOS specification.

```
eval_sos: exp -> prop ord.
retn_sos: exp -> prop ord.
evsos/steps: eval_sos E * !step E E'  >-> {eval_sos E'}.
evsos/value: eval_sos V * !value V >-> {retn_sos V}.
```

Figure 6.24: The operationalization of evsos from Figure 6.23

There are several ways that we can contemplate operationalizing the SOS specification in Figure 6.23. If we operationalize only the evsos predicate, making the evaluation predicate eval_sos and the return predicate retn_sos, then we get what may be the most boring possible substructural operational semantics specification, shown in Figure 6.24. The specification is fully tail-recursive and there are no continuation frames, just an expression transitioning according to the rules of the small-step evaluation relation for an indefinite number of steps as we extend the trace. While the specification is almost trivial, it still captures something of the essence of an SSOS specification – atomic transitions are interpreted as steps (by way the inductively-defined relation $\mathsf{step}\ulcorner e\urcorner\ulcorner e'\urcorner$) and potentially nonterminating or failing computations are interpreted as traces. This specification is also the first case where we have performed operationalization on only part of a specification. In the terminology of Section 6.1.1, Figure 6.24 implies that the rules value/lam, step/app1, step/app2, and step/appred have been assigned to the category D of rules that remain in the deductive fragment while the rules evsos/steps and evsos/value were assigned to the category C of rules that end up being transformed.

In the other direction, we can consider operationalizing only the predicate step, which implies that the rules value/lam, and evsos/steps and evsos/value to the category D and placing step/app1, step/app2, and step/appred in the category C of rules that end up being transformed into the concurrent fragment. The result of this transformation is shown in Figure 6.25. The first subgoal of ev/steps, the proposition $!(\mathsf{decomp}\ E \rightarrowtail \{\mathsf{plug}\ E'\})$, is the first time we have actually encountered the effect of the $D^\dagger$ operation discussed in Section 6.1.2.

Instead of eval, we have decomp in Figure 6.25, since the relevant action is to decompose the expression looking for an applicable β-reduction, and instead of retn we have plug, since the relevant action is to plug the reduced expression back into the larger term. When we operationalized natural semantics, the structure of the suspended cont f propositions was analogous to the control stacks k of abstract machine specifications. In our operationalized SOS specification, the structure of the cont f propositions is analogous to *evaluation contexts*, often written as $E[]$.

$$E[] ::= E[]\ e \mid v\ E[] \mid []$$

The names *decomp* and *plug* are taken from the treatment of evaluation contexts in the functional correspondence [Dan08].

As we foreshadowed in Section 6.4.3, the right computational interpretation of Figure 6.25 is *not* committed-choice forward chaining; the concurrent rules we generate can get stuck without states being stuck, and factoring does not seem to provide a way out. Consider terms of type $\mathsf{eval}\ulcorner(\lambda x.x)\ e\urcorner \rightarrowtail \{\mathsf{retn}\ulcorner(\lambda x.x)\ e'\urcorner\}$ where $e \mapsto e'$ and consequently $\Theta\{x{:}\langle\mathsf{eval}\ulcorner e\urcorner\rangle\} \rightsquigarrow^* \Theta\{y{:}\langle\mathsf{retn}\ulcorner e'\urcorner\rangle\}$. It is entirely possible to use rule step/app1 to derive the following:

$$(x_1{:}\langle\mathsf{eval}\ulcorner(\lambda x.x)\ e\urcorner\rangle) \quad \rightsquigarrow \quad (x_2{:}\langle\mathsf{eval}\ulcorner\lambda x.x\urcorner\rangle,\ \ y_2{:}\langle\mathsf{cont}\ (\mathsf{ap1}\ulcorner e\urcorner)\rangle) \quad \not\rightsquigarrow$$

```
decomp: exp -> prop ord.
plug: exp -> prop ord.

#| Reduction rules |#
step/appred: decomp (app (lam \x. E x) V) * !value V
              >-> {plug (E V)}.

#| Decomposing a term into an evaluation context |#
step/app1:   decomp (app E1 E2)
              >-> {decomp E1 * cont (ap1 E2)}.
step/app2:   decomp (app V1 E2) * !value V1
              >-> {decomp E2 * cont (ap2 V1)}.

#| Reconstituting a term from an evaluation context |#
step/app1/1: plug E1' * cont (ap1 E2)
              >-> {plug (app E1' E2)}.
step/app2/1: plug E2' * cont (ap2 E1)
              >-> {plug (app E1 E2')}.

#mode evsos + -.
evsos: exp -> exp -> prop.

evsos/steps: evsos E V
              <- (decomp E >-> {plug E'})
              <- evsos E' V.

evsos/value: evsos V V <- value V.
```

Figure 6.25: This transformation of Figure 6.23 evokes an evaluation context semantics

While stuck states in abstract machines raised alarm bells about language safety, the stuck state above is not a concern – we merely should have applied rule step/app2 to x_1 instead of rule step/app1. This corresponds to the fact that small-step SOS specifications and specifications that use evaluation contexts map most naturally to the backtracking search behavior generally associated with backward chaining.

6.6.3 Partial evaluation in $\lambda^{\bigcirc}$

As a final example, we present two SLS specifications of Davies' $\lambda^{\bigcirc}$, a logically-motivated type system and natural semantics for partial evaluation [Dav96]. Partial evaluation is not a modular language extension, either on paper or in SLS. On paper, we have to generalize the judgment $e \Downarrow v$ to have free variables; we write $e \Downarrow_\Psi v$ where Ψ contains free expression variables.

$$\frac{}{\lambda x.e \Downarrow_\Psi \lambda x.e} \qquad \frac{e_1 \Downarrow_\Psi \lambda x.e \quad e_2 \Downarrow_\Psi v_2 \quad [v_2/x]e \Downarrow_\Psi v}{e_1\, e_2 \Downarrow_\Psi v}$$

```
#mode fvar -.
fvar: exp -> prop.

#mode evn + + -.
evn: nat -> exp -> exp -> prop.

evn/var:  evn N X X <- fvar X.
evn/lam:  evn N (lam \x. E x) (lam \x. E' x)
           <- (All x. fvar x -> evn N (E x) (E' x)).
evn/app:  evn N (app E1 E2) (app E1' E2')
           <- evn N E1 E1'
           <- evn N E2 E2'.
```

Figure 6.26: Semantics of partial evaluation for $\lambda^{\bigcirc}$ (lambda calculus fragment)

In SLS, this does not actually require us to change the judgment $\mathsf{ev}\, e\, v$ from Figure 6.1, since the specification itself does not specify the context of LF. However, $\lambda^{\bigcirc}$ also requires a separate judgment $e \Downarrow^n_\Psi e'$ for partially evaluating expressions that will be fully evaluated not now but n partial evaluation stages in the future. On the fragment of the logic that deals with functions and applications, this judgment does nothing but induct over the structure of expressions:

$$\frac{x \in \Psi}{x \Downarrow^n_\Psi x} \qquad \frac{e \Downarrow^n_{\Psi,x} e'}{\lambda x.e \Downarrow^n_\Psi \lambda x.e'} \qquad \frac{e_1 \Downarrow^n_\Psi e_1' \quad e_2 \Downarrow^n_\Psi e_2'}{e_1\, e_2 \Downarrow^n_\Psi e_1'\, e_2'}$$

Note that the partial evaluation rule for $\lambda x.e$ extends the variable context Ψ. The SLS encoding of the judgment $e \Downarrow^n_\Psi e'$ is given in Figure 6.26, which also introduces an auxiliary fvar judgment that tracks all the variables in Ψ.

The evaluation judgment $e \Downarrow_\Psi v$ and the partial evaluation judgment $e \Downarrow^n_\Psi e'$ only interact in $\lambda^{\bigcirc}$ through the temporal fragment, which mediates between the two judgments by way of two expressions. The first, $\mathsf{next}\, e$, says that the enclosed expression e should be evaluated one time step later than the surrounding expression. The second, $\mathsf{prev}\, e$, says that the enclosed expression should be evaluated one time step *before* the surrounding expression. When we evaluate $\mathsf{prev}\, e$ at time 1 it is necessary that e evaluates to $\mathsf{next}\, e'$, as $\mathsf{prev}\, (\mathsf{next}\, e')$ at time step 1 will reduce to e'.

$$\frac{e \Downarrow^1_\Psi e'}{\mathsf{next}\, e \Downarrow_\Psi \mathsf{next}\, e'} \qquad \frac{e \Downarrow^{n+1}_\Psi e'}{\mathsf{next}\, e \Downarrow^n_\Psi \mathsf{next}\, e'} \qquad \frac{e \Downarrow_\Psi \mathsf{next}\, e'}{\mathsf{prev}\, e \Downarrow^1_\Psi e'} \qquad \frac{e \Downarrow^{n+1}_\Psi e'}{\mathsf{prev}\, e \Downarrow^{n+2}_\Psi \mathsf{prev}\, e'}$$

This natural semantics specification is represented on the left-hand side of Figure 6.27. Due to the structure of the evn/lam rule, we *cannot* operationalize the evn predicate: it does not have the structure of a C proposition as described in Section 6.1.1. Rule evn/lam does, however, have the structure of a D proposition if we assign evn to the class of predicates that are not operationalized. Therefore, it is possible to operationalize the ev predicate without operationalizing the evn predicate. This leaves the rules in Figure 6.26 completely unchanged; the right-hand side of Figure 6.27 contains the transformed temporal fragment, where evn/next and evn/prev rules are similarly unchanged. The ev/next rule, however, contains a subgoal $!\mathsf{evn}\, (\mathsf{s}\, \mathsf{z})\, E\, V$ which uses a deductive derivation to build a concurrent step. Conversely, the ev/prev rule contains a subgoal

```
ev/next:
ev (next E) (next E')
 <- evn (s z) E E'.

evn/next:
evn N (next E) (next E')
 <- evn (s N) E E'.

ev/prev:
evn (s z) (prev E) E'
 <- ev E (next E').

evn/prev:
evn (s (s N)) (prev E) (prev E')
 <- evn (s N) E E'.
```

```
ev/next:
eval (next E) * !evn (s z) E E'
 >-> {retn (next E')}.

evn/next:
evn N (next E) (next E')
 <- evn (s N) E E'.

ev/prev:
evn (s z) (prev E) E'
 <- (eval E >-> {retn (next E')}).

evn/prev:
evn (s (s N)) (prev E) (prev E')
 <- evn (s N) E E'.
```

Figure 6.27: Semantics for $\lambda^{\bigcirc}$ (temporal fragment)

of eval $E \rightarrowtail \{\mathsf{retn}\,(\mathsf{next}\,V)\}$ that uses a concurrent derivation to create a deductive derivation. This makes the right-hand side of Figure 6.27 the only SLS specification in this dissertation that exhibits an arbitrarily nested dependency between concurrent and deductive reasoning.

The natural semantics of $\lambda^{\bigcirc}$ are not, on a superficial level, significantly more complex than other natural semantics. It turns out, though, that the usual set of techniques for adding state to a operational semantics break down for $\lambda^{\bigcirc}$. Discussing a $\lambda^{\bigcirc}$-like logic with state remained a challenge for many years, though a full solution has recently been given by Kameyama et al. using delimited control operators [KKS11]. Our discussion of operationalization gives a perspective on why this task is difficult, as the specification is far outside of the image of the extended natural semantics we considered in Section 6.5.5. We normally add state to ordered abstract machine specifications by manipulating and extending the set of ambient linear and persistent resources. If we tried to add state to $\lambda^{\bigcirc}$ the same way we added it in Section 6.5.1, the entire store would effectively leave scope whenever computation considered the subterm e of $\mathsf{next}\,e$.

I conjecture that the nominal generalization of ordered linear lax logic alluded to in the discussion of locations and existential name generation (Section 6.5.1) could support operationalizing predicates like $\mathsf{evn}\,n\,e\,e'$. This might, in turn, make it possible to add state to an SSOS specification of $\lambda^{\bigcirc}$, but that is left for future work.

Bibliography

[Age04] Mads Sig Ager. From natural semantics to abstract machines. In *Logic Based Program Synthesis and Transformation (LOPSTR'04)*, pages 245–261. Springer LNCS 3573, 2004. 5.2, 6.5

[CH12] Flávio Cruz and Kuen-Bang (Favonia) Hou. Parallel programming and cost semantics, May 2012. Unpublished note. 6.1

[Che12] James Cheney. A dependent nominal type theory. *Logical Methods in Computer Science*, 8:1–29, 2012. 6.5.1

[Dan03] Olivier Danvy. A rational deconstruction of Landin's SECD machine. Technical Report RS-03-33, BRICS, October 2003. 6

[Dan08] Olivier Danvy. Defunctionalized interpreters for programming languages. In *International Conference on Functional Programming (ICFP'08)*, pages 131–142. ACM, 2008. Invited talk. 5.2, 5.2, 6.5.5, 6.6.2

[Dav96] Rowan Davies. A temporal-logic approach to binding-time analysis. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science (LICS'96)*, pages 184–195, New Brunswick, New Jersey, 1996. 6.6.3

[DMMZ12] Olivier Danvy, Kevin Millikin, Johan Munk, and Ian Zerny. On inter-deriving small-step and big-step semantics: A case study for storeless call-by-need evaluation. *Theoretical Computer Science*, 435:21–42, 2012. 5.2, 6.6.1

[FRR08] Matthew Fluet, Mike Rainey, and John Reppy. A scheduling framework for general-purpose parallel languages. In *International Conference on Functional Programming (ICFP'08)*, pages 241–252. ACM, 2008. 6.5.4, B.4.2

[GMN11] Andrew Gacek, Dale Miller, and Gopalan Nadathur. Nominal abstraction. *Information and Computation*, 209(1):48–73, 2011. 6.5.1

[Har12] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012. 3.3.4, 6, 6.4.1, 6.5.1, 6.5.2, 6.5.4, 8, 6.5.5, 8.5.1, 7, 9.3.2, 9.4

[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

[HM92] John Hannan and Dale Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2(4):415–459, 1992. 5.2, 6.5

[JK12] Jean-Baptiste Jeannin and Dexter Kozen. Computing with capsules. In *14th International Workshop on Descriptional Complexity of Formal Systems (DCFS'12)*, pages 1–19. Springer LNCS 7386, 2012. 6.5.1

[KKS11] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-Chieh Shan. Shifting the stage: Staging with delimited control. *Journal of Functional Programming*, 21(6):617–662, 2011. 6.6.3

[Lan64] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964. 6

[LG09] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009. 5.2, 2, 6.4

[Mos04] Peter D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:195–228, 2004. 6.5.5

[Pfe89] Frank Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Proceedings of the 4th Annual Symposium on Logic in Computer Science (LICS'89)*, pages 313–322, Pacific Grove, California, 1989. 6.1.1

[Pfe04] Frank Pfenning. Substructural operational semantics and linear destination-passing style. In *Programming Languages and Systems*, page 196. Springer LNCS 3302, 2004. Abstract of invited talk. 1.2, 2.1, 5, 5.1, 6.2.2, 7.2

[Pfe12a] Frank Pfenning. Lecture notes on backward chaining, March 2012. Lecture notes for 15-816: Linear Logic at Carnegie Mellon University, available online: http://www.cs.cmu.edu/~fp/courses/15816-s12/Lectures/17-bwdchaining.pdf. 6.6.1

[Pfe12b] Frank Pfenning. Lecture notes on ordered forward chaining, March 2012. Lecture notes for 15-816: Linear Logic at Carnegie Mellon University, available online: http://www.cs.cmu.edu/~fp/courses/15816-s12/Lectures/16-ordchain.pdf. 6.6.1

[Plo04] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004. Reprinted with corrections from Aarhus University technical report DAIMI FN-19. 2, 5.1, 6, 6.6.2

[PS03] Adam Poswolsky and Carsten Schürmann. Factoring report. Technical Report YALEU/DCS/TR-1256, Department of Computer Science, Yale University, November 2003. 6.4.2, 6.4.3

[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

[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

[SP11] Robert J. Simmons and Frank Pfenning. Logical approximation for program analysis. *Higher-Order and Symbolic Computation*, 24(1–2):41–80, 2011. 1.2, 2.5.3, 5.1, 6.5.3, 7, 7, 7.1, 1, 7.1, 7.2, 8, 8.3, 8.3, 8.4.2

[TCP12] Bernardo Toninho, Luis Caires, and Frank Pfenning. Functions as session-typed processes. In *Foundations of Software Science and Computational Structures (FOSSACS'12)*, pages 346–360. Springer LNCS 7213, 2012. 6.5.3