Chapter 7

Destination-passing

The natural notion of ordering provided by ordered linear logic is quite convenient for encoding evolving systems that perform local manipulations to a stack-like structure. This was demonstrated by the push-down automaton for generic bracket matching discussed in the introduction. We can now present that specification in Figure 7.1 as an SLS specification.

```
hd: prop ord.
left: tok -> prop ord.
right: tok -> prop ord.
stack: tok -> prop ord.

push: hd * left X >-> {stack X * hd}.
pop:  stack X * hd * right X >-> {hd}.
```

Figure 7.1: Ordered SLS specification of a PDA for parenthesis matching

Tree structures were reasonably straightforward to encode in the ordered context as well, as we saw from the SSOS specification for parallel pairs in Chapter 6.

At some point, however, the simple data structures that can be naturally encoded in an ordered context become too limiting. When we reach this point, we turn to *destinations*, which allow us to glue control flow together in much more flexible ways. Destinations (terms of type dest) are a bit like the locations l introduced in the specification of mutable storage in Section 6.5.1. They have no constructors: they are only introduced as variables by existential quantification, which means they can freely be subject to unification when the conclusion of a rule declares them to be equal (as described in Section 4.2). Destinations allow us to encode very expressive structures in the linear context of SLS. Instead of using order to capture the local relationships between different propositions, we use destinations.

Linear logic alone is able to express any (flat, concurrent) specifications that can be expressed using ordered atomic propositions. In other words, we did not ever *need* order, it was just a more pleasant way to capture simple control structures. We will demonstrate that fact in this chapter by describing a transformation, *destination-adding*, from specifications with ordered atomic propositions to specifications that only include linear and persistent atomic propositions. This destination-adding transformation, which we originally presented in [SP11], turns all ordered

atomic propositions into linear atomic propositions and tags them with two new arguments (the destinations of the destination-adding transformation). These extra destinations serve as a link between a formerly-ordered atomic proposition and its two former neighbors in the ordered context. When we perform the destination-adding transformation on the specification in Figure 7.1, we get the specification in Figure 7.2.

```
hd: dest -> dest -> prop lin.
left: tok -> dest -> dest -> prop lin.
right: tok -> dest -> dest -> prop lin.
stack: tok -> dest -> dest -> prop lin.

push: hd L M * left X M R >-> {Exists m. stack X L m * hd m R}.
pop:  stack X L M1 * hd M1 M2 * right X M2 R >-> {hd L R}.
```

Figure 7.2: Linear SLS specification of a PDA for parenthesis matching

The specification in Figure 7.2, like every other specification that results from destination-adding, has no occurrences of $\downarrow A^-$ (the transformation has not been adapted to nested rules) and no ordered atomic propositions (these are specifically removed by the transformation). As a result, we write `hd L M` instead of `$hd L M`, omitting the optional linearity indicator `$` on the linear atomic propositions as discussed in Section 4.5. Additionally, by the discussion in Section 3.7, we would be justified in viewing this specification as a linear logical specification (or a CLF specification) instead of a ordered logical specification in SLS. This would not impact the structure of the derivations significantly; essentially, it just means that we would write $A_1^+ \multimap \{A_2^+\}$ instead of $A_1^+ \rightarrowtail \{A_2^+\}$. This reinterpretation was used in [SP11], but we will stick with the notation of ordered logic for consistency, while recognizing that there is nothing ordered about specifications like the one in Figure 7.2.

When the destination-adding translation is applied to ordered abstract machine SSOS specifications, the result is a style of SSOS specification called *destination-passing*. Destination-passing specifications were the original style of SSOS specification proposed in the CLF technical reports [CPWW02]. Whereas the operationalization transformation exposed the structure of natural semantics proofs so that they could be modularly extended with stateful features, the destination-adding translation exposes the control structure of specifications, allowing the language to be modularly extended with control effects and effects like synchronization.

7.1 Logical transformation: destination-adding

The translation we define operates on rules the form $\forall \overline{x}.S_1 \rightarrowtail \{S_2\}$, where S_1 must contain at least one ordered atomic proposition. The syntactic category S is a refinement of the positive types A^+ defined by the following grammar:

$$S ::= p^+_{pers} \mid p^+_{eph} \mid p^+ \mid \mathbf{1} \mid t \doteq s \mid S_1 \bullet S_2 \mid \exists x{:}\tau.S$$

The translation of a rule $\forall \overline{x}.S_1 \rightarrowtail \{S_2\}$ is then $\forall \overline{x}.\, \forall d_L{:}\mathsf{dest}.\, \forall d_R{:}\mathsf{dest}.\, [\![S_1]\!]^{d_L}_{d_R} \rightarrowtail \{[\![S_2]\!]^{d_L}_{d_R}\}$, where $[\![S]\!]^{d_L}_{d_R}$ is defined in Figure 7.3. It is also necessary to transform all ordered predicates with

$$
\begin{aligned}
[\![p^+]\!]^{d_L}_{d_R} &= \mathsf{a}\, t_1 \ldots t_n\, d_L\, d_R \quad (\text{where } p^+ = \mathsf{a}\, t_1 \ldots t_n) \\
[\![p^+_{eph}]\!]^{d_L}_{d_R} &= p^+_{eph} \bullet d_L \doteq d_R \\
[\![p^+_{pers}]\!]^{d_L}_{d_R} &= p^+_{pers} \bullet d_L \doteq d_R \\
[\![\mathbf{1}]\!]^{d_L}_{d_R} &= d_L \doteq d_R \\
[\![t \doteq s]\!]^{d_L}_{d_R} &= t \doteq s \bullet d_L \doteq d_R \\
[\![S_1 \bullet S_2]\!]^{d_L}_{d_R} &= \exists d_M{:}\mathsf{dest}.\, [\![S_1]\!]^{d_L}_{d_M} \bullet [\![S_2]\!]^{d_M}_{d_R} \\
[\![\exists x{:}\tau.\, S]\!]^{d_L}_{d_R} &:= \exists x{:}\tau.\, [\![S]\!]^{d_L}_{d_R}
\end{aligned}
$$

Figure 7.3: Destination-adding transformation

kind $\Pi.x_1{:}\tau_1 \ldots \Pi.x_n{:}\tau_n.\, \mathsf{prop}\, \mathsf{ord}$ that are declared in the signature into predicates with kind $\Pi.x_1{:}\tau_1 \ldots \Pi.x_n{:}\tau_n.\, \mathsf{dest} \to \mathsf{dest} \to \mathsf{prop}\, \mathsf{ord}$ in order for the translation of an ordered atomic proposition p^+ to remain well-formed in the transformed signature.

The destination-adding translation presented here is the same as the one presented in [SP11], except that the transformation operated on rules of the form $\forall \overline{x}.S_1 \twoheadrightarrow \{S_2\}$ and ours will operate over rules of the form $\forall \overline{x}.S_1 \rightarrowtail \{S_2\}$.[1] As discussed in Section 6.2.2, the difference between $\twoheadrightarrow$ and $\rightarrowtail$ is irrelevant in this situation. The restriction to flat specifications, on the other hand, is an actual limitation. We conjecture that the translation presented here, and the correctness proof presented in [SP11], would extend to nested SLS specifications. However, the detailed correctness proofs in that work are already quite tedious (though our explicit notation for patterns as partial derivations can simplify the proof somewhat) and the limited transformation described by Figure 7.3 is sufficient for our purposes. Therefore, we will rely on the existing result, leaving the correctness of a more general development for future work.

According to Figure 7.3, the rule pop in Figure 7.2 should actually be written as follows:

$$
\begin{aligned}
&\mathsf{pop} : \forall x{:}\mathsf{tok}.\, \forall l{:}\mathsf{dest}.\, \forall r{:}\mathsf{dest}. \\
&\qquad (\exists m_1{:}\mathsf{dest}.\, \mathsf{stack}\, x\, l\, m_1 \bullet (\exists m_2{:}\mathsf{dest}.\, \mathsf{hd}\, m_1\, m_2 \bullet \mathsf{right}\, x\, m_2\, r)) \\
&\qquad\quad \rightarrowtail \{\mathsf{hd}\, l\, r\}
\end{aligned}
$$

The destination-adding transformation as implemented produces rules that are equivalent to the specification in Figure 7.3 but that avoid unnecessary equalities and push existential quantifiers as far out as possible (which includes turning existential quantifiers $(\exists x.\, A^+) \rightarrowtail B^-$ into universal quantifiers $\forall x.\, A^+ \rightarrowtail B^-$). The result is a specification, equivalent at the level of synthetic transitions, that looks like the one in Figure 7.2. We write the result of the destination-adding transformation on the signature Σ as $Dest(\Sigma)$.

We can consider a further simplification: is it necessary to generate a new destination m by existential quantification in the head $\exists m.\, \mathsf{stack}\, x\, l\, m \bullet \mathsf{hd}\, m\, r$ of push in Figure 7.2? There is

[1] The monad $\{S_2\}$ did not actually appear in [SP11], and the presentation took polarity into account but was not explicitly polarized. We are justified in reading the lax modality back in by the erasure arguments discussed in Section 3.7.

already a destination m mentioned in the head that will be unused in the conclusion. It would, in fact, be possible to avoid generating new destinations in the transformation of rules $\forall \overline{x}.\, S_1 \rightarrowtail \{S_2\}$ where the head S_2 contains no more ordered atomic propositions than the premise S_1.

We don't perform this simplification for a number of reasons. First and foremost, the transformation described in Figure 7.3 more closely follows the previous work by Morrill, Moot, Piazza, and van Benthem discussed in Section 5.2, and using the transformation as given simplifies the correctness proof (Theorem 7.1). Pragmatically, the additional existential quantifiers also give us more structure to work with when considering program abstraction in Chapter 8. Finally, if we apply both the transformation in Figure 7.3 and a transformation that reuses destinations to an ordered abstract machine SSOS specification, the former transformation produces results that are more in line with existing destination-passing SSOS specifications.

To prove the correctness of destination-adding, we must describe a translation $[\![\Psi; \Delta]\!]$ from process states with ordered, linear, and persistent atomic propositions to ones with only linear and persistent atomic propositions:

$$
\begin{aligned}
[\![\Psi; \cdot]\!] &= (\Psi, d_L{:}\mathsf{dest}; \cdot) \\
[\![\Psi; \Delta, x{:}\langle \mathsf{a}\, t_1 \ldots t_n \rangle\ ord]\!] &= (\Psi', d_L{:}\mathsf{dest}, d_R{:}\mathsf{dest}; \Delta', x{:}\langle \mathsf{a}\, t_1 \ldots t_n\, d_L\, d_R \rangle) \\
&\qquad \text{(where a is ordered and } [\![\Psi; \Delta]\!] = (\Psi', d_L{:}\mathsf{dest}; \Delta')) \\
[\![\Psi; \Delta, x{:}S\ ord]\!] &= (\Psi', d_L{:}\mathsf{dest}, d_R{:}\mathsf{dest}; \Delta', x{:}[\![S]\!]^{d_L}_{d_R}\ ord) \\
&\qquad \text{(where a is ordered and } [\![\Psi; \Delta]\!] = (\Psi', d_L{:}\mathsf{dest}; \Delta')) \\
[\![\Psi; \Delta, x{:}\langle p^+_{eph} \rangle\ eph]\!] &= (\Psi'; \Delta', x{:}\langle p^+_{eph} \rangle) \\
&\qquad \text{(where } [\![\Psi; \Delta]\!] = (\Psi'; \Delta')) \\
[\![\Psi; \Delta, x{:}\langle p^+_{pers} \rangle\ pers]\!] &= (\Psi'; \Delta', x{:}\langle p^+_{pers} \rangle) \\
&\qquad \text{(where } [\![\Psi; \Delta]\!] = (\Psi'; \Delta'))
\end{aligned}
$$

Theorem 7.1 (Correctness of destination-adding)**.**
$[\![\Psi; \Delta]\!] \rightsquigarrow_{Dest(\Sigma)} (\Psi_l; \Delta_l)$ *if and only if* $(\Psi; \Delta) \rightsquigarrow_{\Sigma} (\Psi_o; \Delta_o)$ *and* $(\Psi_l; \Delta_l) = [\![\Psi_o, \Psi''; \Delta_o]\!]$ *for some variable context* Ψ'' *containing destinations free in the translation of* Δ *but not in the translation of* Δ_o.

Proof. This proof is given in detail in [SP11, Appendix A]. It involves a great deal of tedious tracking of destinations, but the intuition behind that tedious development is reasonably straightforward.

First, we need to prove that a right-focused proof of $\Psi; \Delta \vdash_{\Sigma} [S]$ implies that there is an analogous proof of $[\![\Psi; \Delta]\!] \vdash_{Dest(\Sigma)} [[\![S]\!]^{d_L}_{d_R}]$. Conversely, if we can prove $\Psi; \Delta \vdash_{Dest(\Sigma)} [[\![S]\!]^{d_L}_{d_R}]$ in right focus under then linear translation, then it is possible to reconstruct an ordered context $\Psi'; \Delta'$ such that $[\![\Psi'; \Delta']\!] = \Psi; \Delta$ and $\Psi'; \Delta' \vdash_{\Sigma} [S]$ by threading together the destinations from d_L to d_R in Δ. Both directions are established by structural induction on the given derivation. The critical property is that it is possible to reconstruct the ordered context from the context of any right-focus sequent that arises during translation. Proving that property is where the flat structure of rules is particularly helpful; the use of positive atomic propositions comes in handy too [SP11, Lemma 1].

```
eval: exp -> dest -> dest -> prop lin.
retn: exp -> dest -> dest -> prop lin.
cont: frame -> dest -> dest -> prop lin.

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

ev/app:  eval (app E1 E2) D' D
          >-> {Exists d1. eval E1 D' d1 * cont (app1 E2) d1 D}.

ev/app1: retn (lam \x. E x) D' D1 * cont (app1 E2) D1 D
          >-> {Exists d2. eval E2 D' d2 * cont (app2 \x. E x) d2 D}.

ev/app2: retn V2 D' D2 * cont (app2 \x. E x) D2 D
          >-> {eval (E V2) D' D}.
```

Figure 7.4: Translation of Figure 6.6 with vestigial destinations

Second, we need to prove that patterns can be translated in both directions: that if $(\Psi; \Delta) \Longrightarrow (\Psi'; \Delta_o)$ under the original signature then $P :: [\![\Psi; \Delta]\!] \Longrightarrow [\![\Psi'; \Delta_o]\!]$ under the translated signature [SP11, Lemma 4], and that if $P :: [\![\Psi; \Delta]\!] \Longrightarrow (\Psi'; \Delta_l)$ then there exists Δ_o such that $(\Psi'; \Delta_o) = [\![\Psi'; \Delta'']\!]$ [SP11, Lemma 5]. Both directions are again by induction over the structure of the given pattern.

The theorem then follows directly from these two lemmas. There is a trivial induction on spines to handle the sequence of quantifiers, but the core of a flat rule is a proposition $S_1 \rightarrowtail \{S_2\}$ – we reconstruct the ordered context from the value used to prove S_1, and then begin inverting with the positive proposition S_2 in the context. □

If we leave off explicitly mentioning the variable context Ψ, then the trace that represents successfully processing the string [( )] with the transformed push-down automaton specification in Figure 7.2 is as follows (we again underline hd for emphasis):

$$
\begin{aligned}
& y_0{:}\langle \underline{\mathsf{hd}}\ d_0\ d_1\rangle, x_1{:}\langle \mathsf{left\ sq}\ d_1\ d_2\rangle, x_2{:}\langle \mathsf{left\ pa}\ d_2\ d_3\rangle, x_3{:}\langle \mathsf{right\ pa}\ d_3\ d_4\rangle, x_4{:}\langle \mathsf{right\ sq}\ d_4\ d_5\rangle \\
\rightsquigarrow\ & z_1{:}\langle \mathsf{stack\ sq}\ d_0\ d_6\rangle, y_1{:}\langle \underline{\mathsf{hd}}\ d_6\ d_2\rangle, x_2{:}\langle \mathsf{left\ pa}\ d_2\ d_3\rangle, x_3{:}\langle \mathsf{right\ pa}\ d_3\ d_4\rangle, x_4{:}\langle \mathsf{right\ sq}\ d_4\ d_5\rangle \\
\rightsquigarrow\ & z_1{:}\langle \mathsf{stack\ sq}\ d_0\ d_6\rangle, z_2{:}\langle \mathsf{stack\ pa}\ d_6\ d_7\rangle, y_2{:}\langle \underline{\mathsf{hd}}\ d_7\ d_3\rangle, x_3{:}\langle \mathsf{right\ pa}\ d_3\ d_4\rangle, x_4{:}\langle \mathsf{right\ sq}\ d_4\ d_5\rangle \\
\rightsquigarrow\ & z_1{:}\langle \mathsf{stack\ sq}\ d_0\ d_6\rangle, y_3{:}\langle \underline{\mathsf{hd}}\ d_6\ d_4\rangle, x_4{:}\langle \mathsf{right\ sq}\ d_4\ d_5\rangle \\
\rightsquigarrow\ & y_4{:}\langle \underline{\mathsf{hd}}\ d_0\ d_5\rangle
\end{aligned}
$$

One reason for leaving off the variable context Ψ in this example is that by the end it contains the LF variables d_1, d_2, d_3, d_4, d_5, d_6, and d_7, none of which are actually present in the substructural context $y_4{:}\langle \underline{\mathsf{hd}}\ d_0\ d_5\rangle$. We can informally think of these destinations as having been "garbage collected," but this notion is not supported by the formal system we described in Chapter 4.

```
eval: exp -> dest -> prop lin.
retn: exp -> dest -> prop lin.
cont: frame -> dest -> dest -> prop lin.

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

ev/app:  eval (app E1 E2) D
          >-> {Exists d1. eval E1 d1 * cont (app1 E2) d1 D}.

ev/app1: retn (lam \x. E x) D1 * cont (app1 E2) D1 D
          >-> {Exists d2. eval E2 d2 * cont (app2 \x. E x) d2 D}.

ev/app2: retn V2 D2 * cont (app2 \x. E x) D2 D
          >-> {eval (E V2) D}.
```

Figure 7.5: Translation of Figure 6.6 without vestigial destinations

7.1.1 Vestigial destinations

When we apply the translation of expressions to the call-by-value lambda calculus specification from Figure 6.6, we get the specification in Figure 7.4. Because eval and retn are always unique and always appear at the leftmost end of this substructural context, this specification has a quirk: the second argument to eval and retn is always d', and the destination never changes; it is essentially a vestige of the destination-adding transformation. As long as we are transforming a sequential ordered abstract machine, we can eliminate this vestigial destination, giving us the specification in Figure 7.5. This extra destination is *not* vestigial when we translate a parallel specification, but as we discuss in Section 7.2.1, we don't necessarily want to apply destination-adding to parallel ordered abstract machines anyway.

7.1.2 Persistent destination passing

When we translate our PDA specification, it is actually not necessary to translate hd, left, right and stack as linear atomic propositions. If we translate hd as a linear predicate but translate the other predicates as persistent predicates, it will still be the case that there is always exactly one linear atomic proposition $\mathsf{hd}\, d_L\, d_R$ in the context, at most one $\mathsf{stack}\, x\, d\, d_L$ proposition with the same destination d_L, and at most one $\mathsf{right}\, x\, d_R\, d$ or $\mathsf{left}\, x\, d_R\, d$ with the same destination d_R. This means it is still the case that the PDA accepts the string if and only if there is the following series of transitions:

$$(x{:}\langle \mathsf{hd}\, d_0\, d_1\rangle, y_1{:}\langle \mathsf{left}\, x_1\, d_1\, d_2\rangle, \ldots, y_n{:}\langle \mathsf{right}\, x_n\, d_n\, d_{n+1}\rangle) \;\leadsto^*\; (\Gamma, z{:}\langle \mathsf{hd}\, d_0\, d_{n+1}\rangle)$$

Unlike the entirely-linear PDA specification, the final state may include some additional persistent propositions, represented by Γ. Specifically, the final state contains all the original $\mathsf{left}\, x\, d_i\, d_{i+1}$ and $\mathsf{right}\, x\, d_i\, d_{i+1}$ propositions along with all the $\mathsf{stack}\, x\, d\, d'$ propositions that were created during the course of evaluation.

I originally conjectured that a version of Theorem 7.1 would hold in any specification that turned some ordered atomic propositions linear and others persistent just as long as at least one atomic proposition in the premise of every rule remained linear after transformation. This would have given a generic justification for turning left, right and stack persistent in Figure 7.2 and to turning cont persistent in Figure 7.5. However, that condition is not strong enough. To see why, consider a signature with one rule, $\mathsf{a} \bullet \mathsf{b} \bullet \mathsf{a} \rightarrowtail \{\mathsf{b}\}$, where a and b are ordered atomic propositions. We can construct the following trace:

$$(x_1{:}\langle\mathsf{a}\rangle, x_2{:}\langle\mathsf{b}\rangle, x_3{:}\langle\mathsf{a}\rangle, x_4{:}\langle\mathsf{b}\rangle, x_5{:}\langle\mathsf{a}\rangle) \rightsquigarrow (x{:}\langle\mathsf{b}\rangle, x_4{:}\langle\mathsf{b}\rangle, x_5{:}\langle\mathsf{a}\rangle) \not\rightsquigarrow$$

From the same starting point, exactly one other trace is possible:

$$(x_1{:}\langle\mathsf{a}\rangle, x_2{:}\langle\mathsf{b}\rangle, x_3{:}\langle\mathsf{a}\rangle, x_4{:}\langle\mathsf{b}\rangle, x_5{:}\langle\mathsf{a}\rangle) \rightsquigarrow (x_1{:}\langle\mathsf{a}\rangle, x_2{:}\langle\mathsf{b}\rangle, x{:}\langle\mathsf{b}\rangle) \not\rightsquigarrow$$

However, if we perform the destination-passing transformation, letting $\mathsf{a}\, d\, d'$ be a persistent atomic proposition and letting $\mathsf{b}\, d\, d'$ be a linear atomic proposition, then we have a series of transitions in the transformed specification that can reuse the atomic proposition $\mathsf{a}\, d_2\, d_3$ in a way that doesn't correspond to any series of transitions in ordered logic:

$$\begin{aligned}
& x_1{:}\langle\mathsf{a}\, d_0\, d_1\rangle\ \mathit{pers}, x_2{:}\langle\mathsf{b}\, d_1\, d_2\rangle\ \mathit{eph}, x_3{:}\langle\mathsf{a}\, d_2\, d_3\rangle\ \mathit{pers}, x_4{:}\langle\mathsf{b}\, d_3\, d_4\rangle\ \mathit{eph}, x_5{:}\langle\mathsf{a}\, d_4\, d_5\rangle\ \mathit{pers} \\
\rightsquigarrow\ & x_1{:}\langle\mathsf{a}\, d_0\, d_1\rangle\ \mathit{pers}, \underline{x{:}\langle\mathsf{b}\, d_0\, d_3\rangle\ \mathit{eph}}, x_3{:}\langle\mathsf{a}\, d_2\, d_3\rangle\ \mathit{pers}, x_4{:}\langle\mathsf{b}\, d_3\, d_4\rangle\ \mathit{eph}, x_5{:}\langle\mathsf{a}\, d_4\, d_5\rangle\ \mathit{pers} \\
\rightsquigarrow\ & x_1{:}\langle\mathsf{a}\, d_0\, d_1\rangle\ \mathit{pers}, x{:}\langle\mathsf{b}\, d_0\, d_3\rangle\ \mathit{eph}, x_3{:}\langle\mathsf{a}\, d_2\, d_3\rangle\ \mathit{pers}, \underline{x'{:}\langle\mathsf{b}\, d_2\, d_5\rangle\ \mathit{eph}}, x_5{:}\langle\mathsf{a}\, d_4\, d_5\rangle\ \mathit{pers}
\end{aligned}$$

In the first process state, there is a path $d_0, d_1, d_2, d_3, d_4, d_5$ through the context that reconstructs the ordering in the original ordered context. In the second process state, there is still a path d_0, d_3, d_4, d_5 that allows us to reconstruct the ordered context $(x{:}\langle\mathsf{b}\rangle, x_4{:}\langle\mathsf{b}\rangle, x_5{:}\langle\mathsf{a}\rangle)$ by ignoring the persistent propositions associated with x_1 and x_3. However, in the third process state above, no path exists, so the final state cannot be reconstructed as any ordered context.

It would be good to identify a condition that allowed us to selectively turn some ordered propositions persistent when destination-adding without violating (a version of) Theorem 7.1. In the absence of such a generic condition, it is still straightforward to see that performing destination-passing and then turning some propositions persistent is an *abstraction*: if the original system can make a series of transitions, the transformed system can simulate those transitions, but the reverse may not be true. In any case, we can observe that, for many of systems we are interested in, a partially-persistent destination-passing specification can only make transitions that were possible in the ordered specification. The push-down automata with persistent stack, left, and right is one example of this, and we can similarly make the cont predicate persistent in SSOS specifications without introducing any new transitions. Turing the cont predicate persistent will be necessary for the discussion of first-class continuations in Section 7.2.4.

7.2 Exploring the richer fragment

In [SP11], we were interested in exact logical correspondence between ordered abstract machine SSOS specifications and destination-passing SSOS specifications. (Destination-adding is useful in that context because it exposes information about the control structure of computations;

```
cont2: frame -> dest -> dest -> dest -> prop lin.

ev/pair:  eval (pair E1 E2) D
           >-> {Exists d1. Exists d2.
                eval E1 d1 * eval E2 d2 * cont2 pair1 d1 d2 D}.

ev/pair1: retn V1 D1 * retn V2 D2 * cont2 pair1 D1 D2 D
           >-> {retn (pair V1 V2) D}.
```

Figure 7.6: Destination-passing semantics for parallel evaluation of pairs

this control structure can be harnessed by the program abstraction methodology described in Chapter 8 to derive program analyses.) In keeping with our broader use of the logical correspondence, this section will cover programming language features that are not easily expressible with ordered abstract machine SSOS specifications but that can be easily expressed with destination-passing SSOS specifications. Consequently, these are features that can be modularly added to (sequential) ordered abstract machine specifications that have undergone the destination-adding transformation.

The semantics of parallelism and failure presented in Section 7.2.1 are new. The semantics of futures (Section 7.2.3) and synchronization (Section 7.2.2) are based on the specifications first presented in the CLF tech report [CPWW02]. The semantics of first-class continuations (Section 7.2.4) were presented previously in [Pfe04, PS09]. In destination-passing semantics, when we are dealing with fine-grained issues of control flow, the interaction of programming language features becomes more delicate. Parallel evaluation, recoverable failure, and synchronization are compatible features, as are synchronization and futures. Failure and first-class continuations are also compatible. We will not handle other interactions, though it would be interesting to explore the adaptation of Moreau and Ribbens' abstract machine for Scheme with parallel evaluation and callcc as a substructural operational semantics [MR96].

7.2.1 Alternative semantics for parallelism and failure

In Section 6.5.4, we discussed how parallel evaluation and recoverable failure can be combined in an ordered abstract machine SSOS specification. Due to the fact that the two parts of a parallel ordered abstract machine are separated by an arbitrary amount of ordered context, some potentially desirable ways of integrating parallelism and failure were difficult or impossible to express, however.

Once we transition to destination-passing SSOS specifications, it is possible to give a more direct semantics to parallel evaluation that better facilitates talking about failure. Instead of having the stack frame associated with parallel pairs be cont pair1 (as in Figure 6.8) or cont2 pair1 (as discussed in Section 6.5.4), we create a continuation cont2 pair1 d_1 d_2 d with *three* destinations; d_1 and d_2 represent the return destinations for the two subcomputations, whereas d represents the destination to which the evaluated pair is to be returned. This strategy applied to the parallel evaluation of pairs is shown in Figure 7.6.

In ordered specifications, an ordered atomic proposition can be directly connected to at most

```
error: dest -> prop lin.
handle: exp -> dest -> dest -> prop lin.
terminate: dest -> prop lin.

ev/fail:   eval fail D >-> {error D}.
ev/error:  error D' * cont F D' D >-> {error D}.
ev/errorL: error D1 * cont2 F D1 D2 D >-> {error D * terminate D2}.
ev/errorR: error D2 * cont2 F D1 D2 D >-> {error D * terminate D1}.

term/retn: retn V D * terminate D >-> {one}. ; Returning in vain
term/err:  error D * terminate D >-> {one}.  ; Failing redundantly

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

Figure 7.7: Integration of parallelism and exceptions; signals failure as soon as possible

two other ordered propositions: the proposition immediately to the left in the ordered context, and the proposition immediately to the right in the ordered context. What Figure 7.6 demonstrates is that, with destinations, a linear proposition can be locally connected to *any fixed finite number* of other propositions. (If we encode lists of destinations, this need not even be fixed!) Whereas in ordered abstract machine specifications the parallel structure of a computation had to be reconstructed by parsing the context in postfix, a destination-passing specification uses destinations to thread together the treelike dependencies in the context. It would presumably be possible to consider a different version of parallel operationalization that targeted this desirable form of parallel destination-passing specification specifically, but we will not present such a transformation in this thesis.

Using destination-based parallel continuations, we give, in Figure 7.7, a semantics for recoverable failure that eagerly returns errors from either branch of a parallel computation. The rules ev/errorL and ev/errorR immediately pass on errors returned to a frame where the computation forked. Those two rules also leave behind a linear proposition terminate d that will abort the other branch of computation if it returns successfully (rule term/retn) or with an error (rule term/err). It would also be possible to add rules like $\forall d.\, \forall d'.\, \mathsf{cont}\ d'\ d \bullet \mathsf{terminate}\ d \rightarrowtail \{\mathsf{terminate}\ d'\}$ that actively abort the useless branch instead of passively waiting for it to finish. (In a language with state, this can make an observable difference in the results of computation.)

7.2.2 Synchronization

The CLF tech report gives a destination-passing presentation of nearly the full set of Concurrent ML primitives, omitting only negative acknowledgements [CPWW02]. We will present an SLS version of that Concurrent ML specification as a part of the hybrid specification in Appendix B. In Figure 7.8, rather than reprising that specification, we present an extremely simple form of synchronous communication.

```
ev/chan:  eval (chan \c. E c) D >-> {Exists c. eval (E c) D}.

ev/send:  eval (send C E) Dsend
           >-> {Exists d'. eval E d' * cont (send1 C) d' Dsend}.

ev/send1: retn V D' * cont (send1 C) D' Dsend * eval (recv C) Drecv
           >-> {retn unit Dsend * retn V Drecv}.
```

Figure 7.8: Semantics of simple synchronization

New channels are created by evaluating $\ulcorner \mathsf{chan}\, c.e \urcorner = \mathsf{chan}\, \lambda c. \ulcorner e \urcorner$, which introduces a new channel (an LF term of the type channel that has no constructors) and substitutes it for the bound variable c in e. Synchronization happens when there is both a send $\mathsf{send}\, c\, e$ being evaluated in one part of the process state and a receive $\mathsf{recv}\, c$ with the same channel being evaluated in a different part of the process state. The expression e will first evaluate to a value v (rule ev/send). Communication is driven by rule ev/send1, which allows computation to continue in both the sender and the receiver.

Synchronous communication introduces the possibility of deadlocks. Without synchronous communication, the presence of a suspended atomic proposition $\mathsf{eval}\, e\, d$ always indicates the possibility of some transition, and the combination of a proposition $\mathsf{retn}\, v\, d$ and a continuation $\mathsf{cont}\, f\, d\, d'$ can either immediately transition or else are permanently in a stuck state. In [PS09], this observation motivated a classification of atomic propositions as *active* propositions like $\mathsf{eval}\, e\, d$ that independently drive computation, *passive* propositions like $\mathsf{cont}\, f\, d'\, d$ that do not drive computation, and *latent* propositions like $\mathsf{retn}\, f\, d$ that may or may not drive computation based on the ambient environment of passive propositions.

The specification in Figure 7.8 does not respect this classification because a proposition of the form $\mathsf{eval}\, (\mathsf{recv}\, c)\, d$ cannot immediately transition. We could restore this classification by having a rule $\forall c.\, \forall d.\, \mathsf{eval}\, (\mathsf{recv}\, c)\, d \rightarrowtail \{\mathsf{await}\, c\, d\}$ for some new passive linear predicate await and then replacing the premise $\mathsf{eval}\, (\mathsf{recv}\, C)\, D$ in ev/send1 with $\mathsf{await}\, C\, D$.

Labeled transitions

Substructural operational semantics specifications retain much of the flavor of abstract machines, in that we are usually manipulating expressions along with their continuations. In ordered specifications, continuations are connected to evaluating expressions and returning values only by their relative positions in the ordered context; in destination-passing specifications, expressions and values are connected to continuations by the threading of destinations.

Abstract machines are not always the most natural way to express a semantics. This observation is part of what motivated our discussion of the operationalization transformation from natural semantics (motto: "natural" is our first name!) and our informal discussion of statefully-modular natural semantics in Section 6.5.5. In Chapter 6, we showed that the continuation-focused perspective of SSOS allowed us to expose computation to the ambient state. With the example of synchronization above, we see that destination-passing SSOS specifications also expose computations in the process state to *other computations*, which is what allows the synchronization in

```
bind: exp -> exp -> prop pers.    ; Future is complete
promise: dest -> exp -> prop lin. ; Future is waiting on a value

ev/bind:    eval X D * !bind X V >-> {retn V D}.
#| WAITING: eval X D * promise Dfuture X >-> ??? |#
ev/promise: retn V D * promise D X >-> {!bind X V}.

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

ev/fapp1:   retn (flam \x. E x) D1 * cont (app1 E2) D1 D
             >-> {Exists x. eval (E x) D *
                  Exists dfuture. eval E2 dfuture *
                  promise dfuture x}.
```

Figure 7.9: Semantics of call-by-future functions

rule ev/send1 to take place.

In small-step operational semantics, *labeled deduction* is used to describe specifications like the one above. At a high level, in a labeled transition system we inductively define a small step judgment $e \overset{lab}{\longmapsto} e'$ with the property that

- $e \overset{c!v}{\longmapsto} e'$ if e steps to e' by reducing some subterm $\mathsf{send}\, c\, v$, to $\langle\rangle$,
- $e \overset{c?v}{\longmapsto} e'$ if e steps to e' by reducing some subterm $\mathsf{recv}\, c$ to v, and
- e_1 in parallel with e_2 (and possibly also in parallel with some other e_3, e_4, etc.) can step to e_1' in parallel with e_2' (and in parallel with an unchanged e_3, e_4, etc.) if $e_1 \overset{c!v}{\longmapsto} e_1'$ and $e_2 \overset{c?v}{\longmapsto} e_2'$.

Labels essentially serve to pass messages up through the inductive structure of a proposition. In destination-passing SSOS semantics, on the other hand, the internal structure of e is spread out as a series of frames throughout the context, and so the innermost redexes of terms can be directly connected. It would be interesting (but probably quite nontrivial) to consider a translation from labeled deduction systems to destination-passing SSOS specifications along the lines of the operationalization transformation.

7.2.3 Futures

Futures can be seen as a parallel version of call-by-value, and the presentation in Figure 7.9 can be compared to the environment semantics for call-by-value in Figure 6.19. We introduce future-functions as a new kind of function $\mathsf{flam}\, \lambda x.e$ comparable to plain-vanilla call-by-value functions $\mathsf{lam}\, \lambda x.e$, lazy call-by-need functions $\mathsf{lazylam}\, \lambda x.e$, and environment-semantics functions $\mathsf{envlam}\, \lambda x.e$. As in the environment semantics specification, when a call-by-future function returns to a frame $\ulcorner \Box\, e_2 \urcorner = \mathsf{app1} \ulcorner e_2 \urcorner$, we create a new expression x by existential quantification. However, instead of suspending the function body on the stack as we did in Figure 6.19, in Figure 7.9 we create a new destination $dfuture$ and start evaluating the function argument

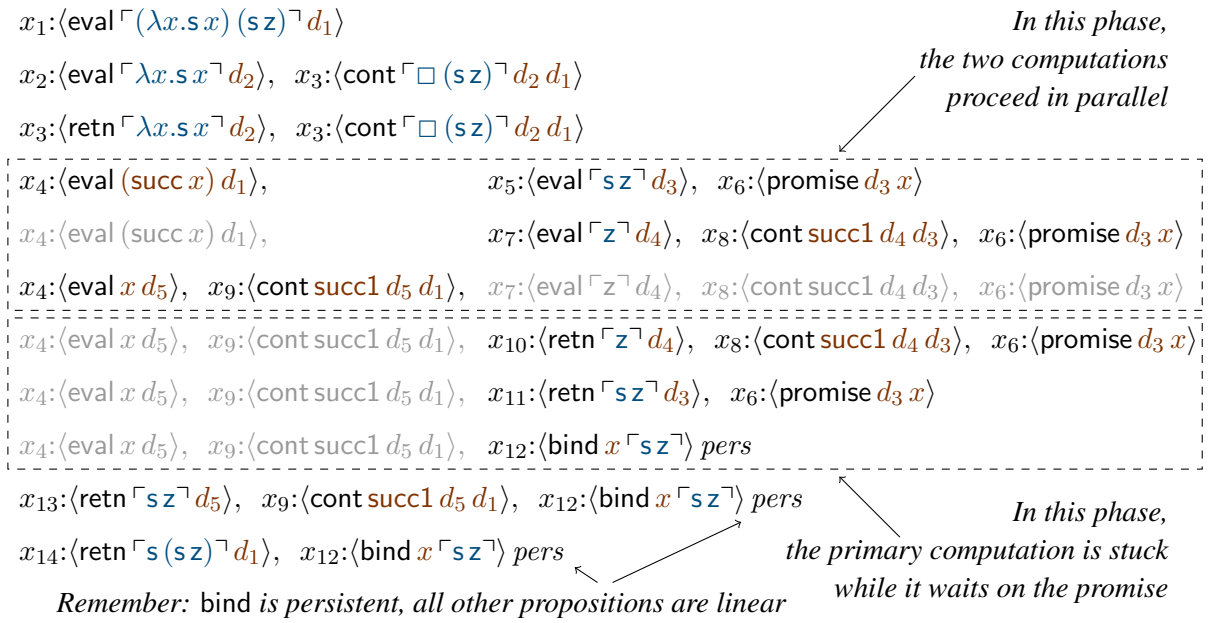

Figure 7.10: Series of process states in an example call-by-future evaluation

towards that destination (rule ev/fapp1). We also create a linear proposition – promise $dfuture\ x$ – that will take any value returned to $dfuture$ and permanently bind it to x (rule ev/promise). As a proposition that only exists during the course of evaluating the argument, promise is analogous to the black hole in our specification of lazy call-by-need.

Futures use destinations to create new and potentially disconnected threads of computation, which can be seen in the example evaluation of $(\lambda x.\mathsf{s}\,x)\,(\mathsf{s}\,\mathsf{z})$ – where $\lambda x.e$ is interpreted as a future function flam instead of lam as before – given in Figure 7.10. That figure illustrates how spawning a future splits the destination structure of the ordered context into two disconnected threads of computation. This was not possible in the ordered framework where every computation had to be somewhere specific in the ordered context relative to the current computation – either to the left, or to the right. These threads are connected not by destinations but by the variable x, which the primary computation needs the future to return before it can proceed.

Note the similarity between the commented-out rule fragment in Figure 7.9 and the commented out rule fragments in the specifications of call-by-need evaluation (Section 6.5.2). In the call-by-need specifications, needing an unavailable value was immediately fatal. With specifications, needing an unavailable value is not immediately fatal: the main thread of computation is stuck, but only until the future's promise is fulfilled. (This again violates the classification of eval as active; as before, this could again be fixed by adding a new latent proposition.)

The destination-passing semantics of futures interact seamlessly with the semantics of synchronization and parallelism, but not with the semantics of recoverable failure: we would have to make some choice about what to do when a future signals failure.

```
eval: exp -> dest -> prop lin.
retn: exp -> dest -> prop lin.
cont: frame -> dest -> dest -> prop pers.

ev/letcc:  eval (letcc \x. E x) D >-> {eval (E (contn D)) D}.
ev/throw2: retn (contn DK) D2 * !cont (throw2 V1) D2 D
            >-> {retn V1 DK}.
```

Figure 7.11: Semantics of first-class continuations (with letcc)

7.2.4 First-class continuations

First-class continuations are a sophisticated control feature. *Continuations* are another name for the stacks k in abstract machine semantics with states $k \rhd e$ and $k \lhd v$ (and also, potentially, $k\blacktriangleleft$ if we want to be able to return errors, as discussed in Section 6.5.4). First-class continuations introduce a new value, $\mathsf{contn}\, k$, to the language. Programmers cannot write continuations k directly, just as they cannot write locations l directly; rather, the expression $\ulcorner \mathsf{letcc}\, x.e \urcorner = \mathsf{letcc}\, \lambda x.\ulcorner e \urcorner$ captures the current expression as a continuation:

$$k \rhd \mathsf{letcc}\, x.e \;\mapsto\; k \rhd [\mathsf{contn}\, k/x]e$$

There is a third construct, $\ulcorner \mathsf{throw}\, e_1\, \mathsf{to}\, e_2 \urcorner = \mathsf{throw} \ulcorner e_1 \urcorner \ulcorner e_2 \urcorner$ that evaluates e_1 to a value v_1, evaluates e_2 to a continuation value $\mathsf{cont}\, k'$, and then throws away the current continuation in favor of returning v_1 to k':

$$\begin{aligned}
k \rhd \mathsf{throw}\, e_1\, \mathsf{to}\, e_2 &\;\mapsto\; (k; \mathsf{throw}\, \square\, \mathsf{to}\, e_2) \rhd e_1 \\
(k; \mathsf{throw}\, \square\, \mathsf{to}\, e_2) \lhd v_1 &\;\mapsto\; (k; \mathsf{throw}\, v_1\, \mathsf{to}\, \square) \rhd e_2 \\
(k; \mathsf{throw}\, v_1\, \mathsf{to}\, \square) \lhd \mathsf{contn}\, k' &\;\mapsto\; k' \lhd v_1
\end{aligned}$$

When handled in a typed setting, a programming language with first-class continuations can be seen as a Curry-Howard interpretation of classical logic.

In destination-passing SSOS specifications, we never represent continuations or control stacks k directly. However, we showed in Section 6.3 that a control stack k is encoded in the context as a series of cont frames. In a destination-passing specification, it is therefore reasonable to associate a k continuation with the destination d that points to the topmost frame $\mathsf{cont}\, f\, d\, d'$ in the stack k encoded in the process state. Destinations stand for continuations in much the same way that introduced variables x in the environment semantics stand for the values v they are bound to through persistent $\mathsf{bind}\, x\, v$ propositions. In Figure 7.11, the rule ev/letcc captures the current continuation d as an expression $\mathsf{cont}\, d$ that is substituted into the subexpression. In rule ev/throw2, the destination dk held by the value $\mathsf{contn}\, dk$ gets the value v_1 returned to it; the previous continuation, represented by the destination d, is abandoned.

Just as it is critical for the bind predicate in the environment semantics to be persistent, it is necessary, when dealing with first-class-continuations, to have the cont predicate be persistent. As discussed in Section 7.1.2, it does not change the behavior of any SSOS specifications we have discussed if linear cont predicates are turned into persistent cont predicates.

Turning cont into a persistent predicate does not, on its own, influence the transitions that are possible, so in a sense we have not changed our SSOS semantics very much in order to add first-class continuations. However, the implicit representation of stacks in the context does complicate adequacy arguments for the semantics in Figure 7.11 relative to the transition rules given above. We will return to this point in Section 9.6 when we discuss generative invariants that apply to specifications using first-class continuations.

Bibliography

[CPWW02] Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-2002-002, Department of Computer Science, Carnegie Mellon University, March 2002. Revised May 2003. 1.1, 2.1, 4.4, 5, 5.1, 7, 7.2, 7.2.2, B.5

[MR96] Luc Moreau and Daniel Ribbens. The semantics of pcall and fork in the presence of first-class continuations and side-effects. In *Parallel Symbolic Languages and Systems (PSLS'95)*, pages 53–77. Springer LNCS 1068, 1996. 7.2, 9.6

[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

[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

[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