Appendix A

A hybrid specification of Mini-ML

In this section, we present the specification that was illustrated in Figure 5.4 as a full SLS specification. This specification is a hybrid or chimera: it has individual features that are presented using big-step natural semantics, nested ordered abstract machine semantics, flat ordered abstract machine semantics, and destination-passing semantics.

Illustrating the logical correspondence methodology that we introduced in Chapter 5 and expounded upon in Chapters 6 and 7, all these specifications can transformed into a common flat destination-passing semantics. With the exception of Concurrent ML primitives, which were only alluded to in Section 7.2.2, all the pieces of this specification (or very similar variants) have been presented elsewhere in the dissertation. The specification in this section is careful to present the entire SLS specification, as opposed to other examples in which the relevant LF declarations were almost always omitted.

The lowest common denominator of destination-passing semantics can be represented in CLF, and the SLS implementation is able to output CLF code readable in Schack-Nielsen's Celf implementation [SNS08]. The implemented logic programming engine of Celf is therefore able to *execute* Mini-ML programs encoded as terms of type exp in our hybrid specification.

A.1 Pure Mini-ML

There are various toy languages calling themselves "Mini-ML" in the literature. All Mini-MLs reflect some of the flavor of functional programming while avoiding features such as complex pattern-matching and datatype declarations that make the core language of Standard ML [MTHM97] a bit more complicated. Of course, Mini-MLs universally avoid the sophisticated ML module language as well.

Like the PCF language [Plo77], Mini-ML variants usually have at least a fixed-point operator, unary natural numbers, and functions. We add Boolean and pair values to this mix, as well as the arbitrary choice operator $\ulcorner e_1 \circledcirc e_2 \urcorner = \mathsf{arb}\ulcorner e_1 \urcorner \ulcorner e_2 \urcorner$ from Section 6.4.1. The specification in Section B.1.2 is an encoding of the natural semantics judgment $\ulcorner e \Downarrow v \urcorner = \mathsf{ev}\ulcorner e \urcorner \ulcorner v \urcorner$ presented throughout Chapter 6. The language is pure – the only effect is nontermination – so we can fully specify the language as a big-step operational semantics.

A.1.1 Syntax

```
exp: type.
lam: (exp -> exp) -> exp.                  ; fn x => e
app: exp -> exp -> exp.                    ; e(e)
fix: (exp -> exp) -> exp.                  ; fix x.e
true: exp.                                 ; tt
false: exp.                                ; ff
ite: exp -> exp -> exp -> exp.             ; if e then et else ef
zero: exp.                                 ; z
succ: exp -> exp.                          ; s(e)
case: exp -> exp -> (exp -> exp) -> exp.   ; case e of z => ez | s x => es
unit: exp.                                 ; <>
pair: exp -> exp -> exp.                   ; <e1, e2>
fst: exp -> exp.                           ; e.1
snd: exp -> exp.                           ; e.2
arb: exp -> exp -> exp.                    ; e1 ?? e2
```

A.1.2 Natural semantics

```
#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.

ev/fix:   ev (fix \x. E x) V
           <- ev (E (fix \x. E x)) V.

ev/true:  ev true true.

ev/false: ev false false.

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

ev/ite:   ev (ite E Et Ef) V
           <- ev E V'
           <- caseb V' Et Ef V.

case/t:   caseb true Et Ef V
           <- ev Et V.

case/f:   caseb false Et Ef V
           <- ev Et V.

ev/zero:  ev zero zero.
```

```
ev/succ:  ev (succ E) (succ V)
           <- ev E V.

#mode casen + + + -.
casen: exp -> exp -> (exp -> exp) -> exp -> prop.

ev/case:  ev (case E Ez \x. Es x) V
           <- ev E V'
           <- casen V' Ez (\x. Es x) V.

case/z:   casen zero Ez (\x. Es x) V
           <- ev Ez V.

case/s:   casen (succ V) Ez (\x. Es x) V
           <- ev (Es V) V.

ev/unit:  ev unit unit.

ev/pair:  ev (pair E1 E2) (pair V1 V2)
           <- ev E1 V1
           <- ev E2 V2.

ev/fst:   ev (fst E) V1
           <- ev E (pair V1 V2).

ev/snd:   ev (snd E) V2
           <- ev E (pair V1 V2).

ev/arb1:  ev (arb E1 E2) V
           <- ev E1 V.

ev/arb2:  ev (arb E1 E2) V
           <- ev E2 V.
```

A.2 State

The strength of an ordered abstract machine semantics specification is its ability to handle modular addition of *stateful* features. While Section 6.5 discussed the modular extension of *flat* ordered abstract machines, nested ordered abstract machines are also perfectly capable of handling stateful features such as mutable storage (Section 6.5.1) and call-by-need recursive suspensions (Section 6.5.2).

A.2.1 Syntax

```
mutable_loc: type.
loc: mutable_loc -> exp.                  ; (no concrete syntax)
ref: exp -> exp.                          ; ref e
get: exp -> exp.                          ; !e
set: exp -> exp -> exp.                   ; e1 := e2
```

```
bind_loc: type.
issusp: bind_loc -> exp.              ; (no concrete syntax)
thunk: (exp -> exp) -> exp.           ; thunk x.e
force: exp -> exp.                    ; force e
```

A.2.2 Nested ordered abstract machine semantics

In Section 6.5.1, we discussed mutable storage as a flat ordered abstract machine (Figure 6.14), but it is equally straightforward to describe a nested ordered abstract machine for mutable storage.

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

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

ev/ref: eval (ref E1)
         >-> {eval E1 *
              (All V1. retn V1
                >-> {Exists l. retn (loc l) * $cell l V1})}.

ev/get: eval (get E1)
         >-> {eval E1 *
              (All L. retn (loc L) * $cell L V
                >-> {retn V * $cell L V})}.

ev/set: eval (set E1 E2)
         >-> {eval E1 *
              (All L. retn (loc L)
                >-> {eval E2 *
                     (All V2. All Vignored. retn V2 * $cell L Vignored
                       >-> {retn unit * $cell L V2})})}.
```

A.2.3 Flat ordered abstract machine semantics

In Section 6.5.2, we gave both a semantics for call-by-need recursive suspensions, both as a flat ordered abstract machine (Figure 6.16) and a nested ordered abstract machine (Figure 6.17). However, the nested ordered abstract machine from Figure 6.17 uses the with connective $A^- \mathbin{\&} B^-$, and our implementation of defunctionalization transformation doesn't handle this connective. Therefore, we repeat the flat ordered abstract machine from Figure 6.16. Note, however, that there is no technical reason why $A^- \mathbin{\&} B^-$ should be difficult to handle; any actual difficulty is mostly in terms of making sure uncurrying (Section 6.2.2) does something sensible.

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

force1: frame.
bind1: bind_loc -> frame.
```

```
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/suspended1: retn (issusp L) * cont force1 * $susp L (\x. E' x)
                >-> {eval (E' (issusp L)) * cont (bind1 L) * $blackhole L}.

ev/suspended2: retn V * cont (bind1 L) * $blackhole L
                >-> {retn V * !bind L V}.

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

A.3 Failure

The reason we introduced frames in Section 6.2.3 was to allow the semantics of recoverable failure to talk generically about all continuations. In Section B.3.2, we generalize the semantics from Section 6.5.4 by having exceptions carry a value.

A.3.1 Syntax

```
raise: exp -> exp.                      ; raise e
try: exp -> (exp -> exp) -> exp.        ; try e catch x.ef
```

A.3.2 Flat ordered abstract machine semantics

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

raise1: frame.

ev/raise:   eval (raise E) >-> {eval E * cont raise1}.

ev/raise1:  retn V * cont raise1 >-> {error V}.

ev/try:     eval (try E1 (\x. E2 x))
             >-> {eval E1 * handle (\x. E2 x)}.

error/cont: error V * cont F >-> {error V}.

error/hand: error V * handle (\x. E2 x) >-> {eval (E2 V)}.

retn/hand:  retn V * handle (\x. E2 x) >-> {retn V}.
```

A.4 Parallelism

While ordered abstract machines can represent parallel evaluation, and the operationalization transformation can expose it, parallel ordered abstract machines and the destination-adding transformation do not interact a helpful way. Therefore, for our hybrid specification, we will describe parallel evaluation at the destination-passing level, as in Section 7.2.1.

A.4.1 Destination-passing semantics

Instead of the parallel pairs shown in Figure 6.8 and Figure 7.6, we will use a parallel let construct $\ulcorner \mathsf{letpar}\,(x_1, x_2) = (e_1, e_2)\,\mathsf{in}\,e \urcorner = \mathsf{letpar}\,\ulcorner e_1 \urcorner \ulcorner e_2 \urcorner\, \lambda x_1.\, \lambda x_2.\, \ulcorner e \urcorner$.

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

letpar: exp -> exp -> (exp -> exp -> exp) -> exp.
letpar1: (exp -> exp -> exp) -> frame.

ev/letpar:  eval (letpar E1 E2 \x. \y. E x y) D
             >-> {Exists d1. eval E1 d1 *
                  Exists d2. eval E2 d2 *
                  cont2 (letpar1 \x. \y. E x y) d1 d2 D}.

ev/letpar1: retn V1 D1 * retn V2 D2 *
            cont2 (letpar1 \x. \y. E x y) D1 D2 D
             >-> {eval (E V1 V2) D}.
```

A.4.2 Integration of parallelism and exceptions

We have discussed two semantics for parallel evaluation. The first semantics, in Section 6.5.4, only raised an error if both parallel branches terminated and one raised an error. The second semantics, in Section 7.2.1, raised an error if *either* branch raised an error, and then allowed the other branch to return a value.

We will demonstrate a third option here, the sequential exception semantics used by Manticore [FRR08]. An error raised by the second scrutinee e_2 of letpar will only be passed up the stack if the first scrutinee e_1 returns a value. We also represent Manticore's *cancellation* – if the first branch of a parallel evaluation raises an exception, then rather than passively waiting for the second branch to terminate we proactively walk up its stack attempting to cancel the computation.

```
cancel: dest -> prop lin.

ev/errorL:    error V D1 * cont2 X D1 D2 D
               >-> {error V D * cancel D2}.

ev/errorR:    retn _ D1 * error V D2 * cont2 _ D1 D2 D
               >-> {error V D}.
```

```
cancel/eval:  eval _ D * cancel D >-> {one}.
cancel/retn:  retn _ D * cancel D >-> {one}.
cancel/error: error _ D * cancel D >-> {one}.
cancel/cont:  cont _ D' D * cancel D >-> {cancel D'}.
cancel/cont2: cont2 _ D1 D2 D * cancel D >-> {cancel D1 * cancel D2}.
```

A.5 Concurrency

Concurrent ML is an excellent example of the power of destination-passing specifications. The Concurrent ML primitives allow a computation to develop a rich interaction structure that does not mesh well with the use of ordered logic, but the destination-passing style allows for a clean specification that is fundamentally like the one used for simple synchronization in Section 7.2.2. This account directly follows Cervesato et al.'s account [CPWW02], similarly neglecting negative acknowledgements.

A.5.1 Syntax

```
channel: type.
spawn: exp -> exp.                         ; spawn e
exit: exp.                                 ; exit
newch: exp.                                ; channel
chan: channel -> exp.                      ; (no concrete syntax)
sync: exp -> exp.                          ; sync e
send: exp -> exp -> exp.                   ; send c e
recv: exp -> exp.                          ; recv c
always: exp -> exp.                        ; always e
choose: exp -> exp -> exp.                 ; e1 + e2
never: exp.                                ; 0
wrap: exp -> (exp -> exp) -> exp.          ; wrap e in x.e'
```

A.5.2 Natural semantics

Many of the pieces of Concurrent ML do not interact with concurrency directly; instead, they build channels and event values that drive synchronization. In our hybrid specification methodology, we can give these pure parts of the Concurrent ML specification a big-step natural semantics specification.

```
ev/chan:    ev (chan C) (chan C).

ev/always:  ev (always E1) (always V1)
             <- ev E1 V1.

ev/recv:    ev (recv E1) (recv V1)
             <- ev E1 V1.

ev/send:    ev (send E1 E2) (send V1 V2)
             <- ev E1 V1
             <- ev E2 V2.
```

```
ev/choose:  ev (choose E1 E2) (choose V1 V2)
             <- ev E1 V1
             <- ev E2 V2.

ev/never:   ev never never.

ev/wrap:    ev (wrap E1 \x. E2 x) (wrap V1 \x. E2 x)
             <- ev E1 V1.
```

A.5.3 Destination-passing semantics

The destination-passing semantics of Concurrent ML has three main parts. The first part, a spawn primitive, creates a new disconnected thread of computation – the same kind of disconnected thread that we used for the interaction of parallelism and failure in Section 7.2.1. The newch primitive creates a new channel for communication.

```
terminate: dest -> prop lin.

term/retn:  retn _ D * terminate D >-> {one}.
term/error: error _ D * terminate D >-> {one}.

ev/spawn:   eval (spawn E) D
             >-> {retn unit D *
                  Exists d'. eval E d' * terminate d'}.

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

The critical feature of Concurrent ML is synchronization, which is much more complex than the simple synchronization described in Section 7.2.2, and has something of the flavor of the labels described in that section. An action can include many alternatives, but if a send and a receive can simultaneously take place along a single channel, then the synch/communicate rule can enable both of the waiting synch $v\ d$ propositions to proceed evaluating as eval propositions.

Here as in Cervesato et al.'s specification, events are atomically paired up using the backward-chaining action rules, which are not transformed: the intent is for the action predicate to act like a backtracking, backward-chaining logic programs in the course of evaluation.

```
#mode action + - -.
action: exp -> exp -> (exp -> exp) -> prop.

act/t: action (always V) (always V) (\x. x).
act/s: action (send (chan C) V) (send (chan C) V) (\x. x).
act/v: action (recv (chan C)) (recv (chan C)) (\x. x).
act/l: action (choose Event1 Event2) Lab (\x. E x)
        <- action Event1 Lab (\x. E x).
act/r: action (choose Event1 Event2) Lab (\x. E x)
        <- action Event2 Lab (\x. E x).
act/w: action (wrap Event1 \x. E2 x) Lab (\x. app (lam (\x. E2 x)) (E x))
        <- action Event1 Lab (\x. E x).
```

```
synch: exp -> dest -> prop lin.

sync1: frame.

ev/sync:
  eval (sync E1) D >-> {Exists d1. eval E1 d1 * cont sync1 d1 D}.

ev/sync1:
  retn W D1 * cont sync1 D1 D >-> {synch W D}.

synch/always:
  synch Event D *
  !action Event (always V') (\x. E x)
   >-> {eval (E V') D}.

synch/communicate:
  synch Event1 D1 *
  !action Event1 (send (chan C) V) (\x. E1 x) *
  synch Event2 D2 *
  !action Event2 (recv (chan C)) (\x. E2 x)
   >-> {eval (E1 unit) D1 * eval (E2 V) D2}.
```

A.6 Composing the semantics

Within Standard ML, we can read the various specifications described in this appendix and use directives to successively operationalize, defunctionalize, add destinations, and output CLF that is readable by the Celf implementation.

```
CM.make "../../sls/sources.cm";
fun HEADING s = print ("\n\n== "^s^" ==\n\n");
Frontend.init ();

HEADING "NATURAL SEMANTICS";
Frontend.reset ();
Frontend.read "#operationalize \"ord-nested.auto.sls\" \
\                 (ev ~> eval retn)\
\                 (casen ~> casen retn)\
\                 (caseb ~> caseb retn).";
Frontend.load "compose/pure-exp.sls";
Frontend.load "compose/pure-natsem.sls";
Frontend.load "compose/concur-exp.sls";
Frontend.load "compose/concur-natsem.sls";
Frontend.read "#operationalize stop.";

HEADING "ORDERED ABSTRACT MACHINES (nested)";
Frontend.reset ();
Frontend.read "#defunctionalize \"ord-flat.auto.sls\" \
\                 (cont frame : ord).";
Frontend.load "ord-nested.auto.sls";
Frontend.load "compose/imp-exp.sls";
Frontend.load "compose/imp-ordmachine.sls";
Frontend.read "#defunctionalize stop.";

HEADING "ORDERED ABSTRACT MACHINES (flat)";
Frontend.reset ();
Frontend.read "#destadd \"dest.auto.sls\" \
\                  dest eval retn error casen caseb.";
Frontend.load "ord-flat.auto.sls";
Frontend.load "compose/control-exp.sls";
Frontend.load "compose/control-ordmachine.sls";
Frontend.load "compose/susp-ordmachine.sls";

HEADING "DESTINATION-PASSING";
Frontend.reset ();
Frontend.read "#clf \"miniml.clf\".";
Frontend.load "dest.auto.sls";
Frontend.load "compose/par-dest1.sls";
Frontend.load "compose/par-dest2.sls";
Frontend.load "compose/concur-dest1.sls";
Frontend.load "compose/concur-dest2.sls";
Frontend.read "#clf stop.";
```

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

[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

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML*. MIT Press, 1997. B.1

[Plo77] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, 1977. B.1

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