6.5 A Continuation Machine

The natural semantics for Mini-ML presented in Chapter 2 is called a *big-step semantics*, since its only judgment relates an expression to its final value—a "big step". There are a variety of properties of a programming language which are difficult or impossible to express as properties of a big-step semantics. One of the central ones is that "well-typed programs do not go wrong". Type preservation, as proved in Section 2.6, does not capture this property, since it presumes that we are already given a complete evaluation of an expression e to a final value v and then relates the types of e and v. This means that despite the type preservation theorem, it is possible that an attempt to find a value of an expression e leads to an intermediate expression such as **fst z** which is ill-typed and to which no evaluation rule applies. Furthermore, a big-step semantics does not easily permit an analysis of non-terminating computations.

An alternative style of language description is a *small-step semantics*. The main judgment in a small-step operational semantics relates the state of an abstract machine (which includes the expression to be evaluated) to an immediate successor state. These small steps are chained together until a value is reached. This level of description is usually more complicated than a natural semantics, since the current state must embody enough information to determine the next and all remaining computation steps up to the final answer. It is also committed to the order in which subexpressions are evaluated and thus somewhat less abstract than a natural, big-step semantics.

In this section we construct a machine directly from the original natural semantics of Mini-ML in Section 2.3 (and not from the environment-based semantics in Section 6.1). This illustrates the general technique of *continuations* to sequentialize computations. Another application of the technique at the level of expressions (rather than computations) is given in Section **??**.

In order to describe the continuation machine as simply as possible, we move to a presentation of the language in which expressions and values are explicitly separated. An analogous separation formed the basis for environment-based evaluation on de Bruijn expression in Section 6.2. We now also explicitly distinguish variables x ranging over values and variables u ranging over expressions. This makes it immediately apparent, for example, that the language has a call-by-value semantics for functions; a call-by-name version for functions would be written as $\mathbf{lam}\, u.\, e$.

Expressions	e	$::=$	$\mathbf{z} \mid \mathbf{s}\ e \mid (\mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\ x \Rightarrow e_3)$	*Natural numbers*
			$\mid \langle e_1, e_2 \rangle \mid \mathbf{fst}\ e \mid \mathbf{snd}\ e$	*Pairs*
			$\mid \mathbf{lam}\ x.\ e \mid e_1\ e_2$	*Functions*
			$\mid \mathbf{let\, val}\ x = e_1\ \mathbf{in}\ e_2$	*Definitions*
			$\mid \mathbf{let\, name}\ u = e_1\ \mathbf{in}\ e_2$	
			$\mid \mathbf{fix}\ u.\ e$	*Recursion*
			$\mid u$	*Variables*
			$\mid v$	*Values*
Values	v	$::=$	$\mathbf{z}^* \mid \mathbf{s}^*\ v$	*Natural numbers*
			$\mid \langle v_1, v_2 \rangle^*$	*Pairs*
			$\mid \mathbf{lam}^*\ x.\ e$	*Functions*
			$\mid x$	*Variables*

Note that expressions and values are mutually recursive syntactic categories, but that an arbitrary value can occur as an expression. The implementation in Elf is completely straightforward, keeping in mind that we have to provide an explicit coercion `vl` from values to expressions.

```
exp  : type.  %name exp E
val  : type.  %name val V

z     : exp.
s     : exp -> exp.
case  : exp -> exp -> (val -> exp) -> exp.
pair  : exp -> exp -> exp.
fst   : exp -> exp.
snd   : exp -> exp.
lam   : (val -> exp) -> exp.
app   : exp -> exp -> exp.
letv  : exp -> (val -> exp) -> exp.
letn  : exp -> (exp -> exp) -> exp.
fix   : (exp -> exp) -> exp.

vl    : val -> exp.

z*    : val.
s*    : val -> val.
pair* : val -> val -> val.
lam*  : (val -> exp) -> val.
```

The standard operational semantics for this representation of expressions and values is straightforward and can be found in the code supplementing these notes. The equivalence proof is left to Exercise 6.17.

Our goal now is define a small-step semantics. For this, we isolate an expression e to be evaluated, and a *continuation* K which contains enough information to carry out the rest of the evaluation necessary to compute the overall value. For example, to evaluate a pair $\langle e_1, e_2 \rangle$ we first compute the value of e_1, remembering that the next task will be the evaluation of e_2, after which the two values have to be paired. This also shows the need for intermediate *instructions*, such as "*evaluate the second element of a pair*" or "*combine two values into a pair*". One particular kind of instruction, written simply as e, triggers the first step in the computation based on the structure of e.

Because we always fully evaluate one expression before moving on to the next, the continuation has the form of a stack. Because the result of evaluating the current expression must be communicated to the continuation, each item on the stack is a function from values to instructions. Finally, when we have computed a value, we *return* it by applying the first item on the continuation stack. Thus the following structure emerges, to be supplement by further auxiliary instructions as necessary.

$$\begin{array}{rrcl} \text{Instructions} & i & ::= & e \mid \mathbf{return}\ v \\ \text{Continuations} & K & ::= & \mathbf{init} \mid K; \lambda x.\, i \\ \text{Machine States} & S & ::= & K \diamond I \mid \mathbf{answer}\ v \end{array}$$

Here, **init** is the initial continuation, indicating that nothing further remains to be done. The machine state **answer** v represents the final value of a computation sequence. Based on the general consideration, we have the following transitions of the abstract machine.

$$S \Longrightarrow S' \quad S \text{ goes to } S' \text{ in one computation step}$$

$$\frac{}{\mathbf{init} \diamond \mathbf{return}\ v \Longrightarrow \mathbf{answer}\ v}\,\mathsf{st\_init} \qquad \frac{}{K; \lambda x.\, i \diamond \mathbf{return}\ v \Longrightarrow K \diamond [v/x]i}\,\mathsf{st\_return}$$

$$\frac{}{K \diamond v \Longrightarrow K \diamond \mathbf{return}\ v}\,\mathsf{st\_vl}$$

Further rules arise from considering each expression constructor in turn, possibly adding new special-purpose intermediate instructions. We will write the rules in the form *label* :: $S \Longrightarrow S'$ as a more concise alternative to the format used above. The meaning, however, remains the same: each rules is an axiom defining the transition judgment.

$$
\begin{array}{llllllll}
\mathsf{st\_z} & :: & K & \diamond & \mathbf{z} & \Longrightarrow & K & \diamond\ \mathbf{return}\ \mathbf{z}^* \\
\mathsf{st\_s} & :: & K & \diamond & \mathbf{s}\ e & & & \\
 & & & & & \Longrightarrow & \multicolumn{2}{l}{K; \lambda x.\ \mathbf{return}\ (\mathbf{s}^*\ x) \diamond e} \\
\mathsf{st\_case} & :: & K & \diamond & \mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\ x \Rightarrow e_3 & & & \\
 & & & & \multicolumn{4}{r}{\Longrightarrow\ K; \lambda x_1.\ \mathbf{case}_1\ x_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\ x \Rightarrow e_3\ \diamond\ e_1} \\
\mathsf{st\_case1\_z} & :: & K & \diamond & \mathbf{case}_1\ \mathbf{z}^*\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\ x \Rightarrow e_3 & \Longrightarrow & K & \diamond\ e_2 \\
\mathsf{st\_case1\_s} & :: & K & \diamond & \mathbf{case}_1\ \mathbf{s}^*\ v_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\ x \Rightarrow e_3 & \Longrightarrow & K & \diamond\ [v_1'/x]e_3
\end{array}
$$

We can see that the **case** construct requires a new instruction of the form $\mathbf{case}_1\ v_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\ x \Rightarrow e_3$. This is distinct from $\mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\ x \Rightarrow e_3$ in that the case subject is known to be a value. Without an explicit new construct, computation could get into an infinite loop since every value is also an expression which evaluates to itself. It should now be clear how pairs and projections are computed; the new instructions are $\langle v_1, e_2\rangle_1$, $\mathbf{fst}_1$, and $\mathbf{snd}_1$.

$$
\begin{array}{lllllllll}
\mathsf{st\_pair} & :: & K & \diamond & \langle e_1, e_2\rangle & \Longrightarrow & K; \lambda x_1.\ \langle x_1, e_2\rangle_1 & \diamond & e_1 \\
\mathsf{st\_pair1} & :: & K & \diamond & \langle v_1, e_2\rangle_1 & \Longrightarrow & K; \lambda x_1.\ \mathbf{return}\ \langle v_1, x_2\rangle^* & \diamond & e_2 \\
\mathsf{st\_fst} & :: & K & \diamond & \mathbf{fst}\ e & \Longrightarrow & K; \lambda x.\ \mathbf{fst}_1\ x & \diamond & e \\
\mathsf{st\_fst1} & :: & K & \diamond & \mathbf{fst}\ \langle v_1, v_2\rangle^* & \Longrightarrow & K & \diamond & \mathbf{return}\ v_1 \\
\mathsf{st\_snd} & :: & K & \diamond & \mathbf{snd}\ e & \Longrightarrow & K; \lambda x.\ \mathbf{snd}_1\ x & \diamond & e \\
\mathsf{st\_snd1} & :: & K & \diamond & \mathbf{snd}\ \langle v_1, v_2\rangle^* & \Longrightarrow & K & \diamond & \mathbf{return}\ v_2
\end{array}
$$

Neither functions, nor definitions or recursion introduce any essentially new ideas. We add two new forms of instructions, $\mathbf{app}_1$ and $\mathbf{app}_2$, for the intermediate forms while evaluating applications.

$$
\begin{array}{lllllllll}
\mathsf{st\_lam} & :: & K & \diamond & \mathbf{lam}\ x.\ e & \Longrightarrow & K & \diamond & \mathbf{return}\ \mathbf{lam}^* x.\ e \\
\mathsf{st\_app} & :: & K & \diamond & e_1\ e_2 & \Longrightarrow & K; \lambda x_1.\ \mathbf{app}_1\ x_1\ e_2 & \diamond & e_1 \\
\mathsf{st\_app1} & :: & K & \diamond & \mathbf{app}_1\ v_1\ e_2 & \Longrightarrow & K; \lambda x_2.\ \mathbf{app}_1\ v_1\ x_2 & \diamond & e_2 \\
\mathsf{st\_app2} & :: & K & \diamond & \mathbf{app}_2\ (\mathbf{lam}\ x.\ e_1')\ v_2 & \Longrightarrow & K & \diamond & [v_2/x]e_1' \\
\\
\mathsf{st\_letv} & :: & K & \diamond & \mathbf{let\ val}\ x = e_1\ \mathbf{in}\ e_2 & \Longrightarrow & K; \lambda x_1.\ [x_1/x]e_2 & \diamond & e_1 \\
\mathsf{st\_letn} & :: & K & \diamond & \mathbf{let\ name}\ u = e_1\ \mathbf{in}\ e_2 & \Longrightarrow & K & \diamond & [e_1/x]e_2 \\
\\
\mathsf{st\_fix} & :: & K & \diamond & \mathbf{fix}\ u.\ e & \Longrightarrow & K & \diamond & [\mathbf{fix}\ u.\ e/u]e
\end{array}
$$

The complete set of instructions as extracted from the transitions above:

$$
\begin{array}{llll}
\text{Instructions} & i & ::= & e \mid \mathbf{return}\ v \\
 & & & \mid \mathbf{case}_1\ v_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2\ \mid \mathbf{s}\ x \Rightarrow e_3 \quad \text{Natural numbers} \\
 & & & \mid \langle v_1, e_2\rangle_1 \mid \mathbf{fst}_1\ v \mid \mathbf{snd}_1\ v \quad \text{Pairs} \\
 & & & \mid \mathbf{app}_1\ v_1\ e_2 \mid \mathbf{app}_2\ v_1\ v_2 \quad \text{Functions}
\end{array}
$$

The implementation of instructions, continuations, and machine states in Elf uses infix operations to make continuations and states more readable.

```
% Machine Instructions
inst : type.  %name inst I

ev : exp -> inst.
return : val -> inst.

case1 : val -> exp -> (val -> exp) -> inst.
pair1 : val -> exp -> inst.
fst1 : val -> inst.
snd1 : val -> inst.
app1 : val -> exp -> inst.
app2 : val -> val -> inst.

% Continuations
cont : type.  %name cont K

init : cont.
;    : cont -> (val -> inst) -> cont.
%infix left 8 ;

% Continuation Machine States
state : type.  %name state S

# : cont -> inst -> state.
answer : val -> state.
%infix none 7 #
```

The following declarations constitute a direct translation of the transition rules above.

```
=> : state -> state -> type.          %name => St
%infix none 6 =>

% Natural Numbers
st_z : K # (ev z) => K # (return z*).
st_s : K # (ev (s E)) => (K ; [x:val] return (s* x)) # (ev E).
st_case : K # (ev (case E1 E2 E3)) => (K ; [x1:val] case1 x1 E2 E3) # (ev E1).
st_case1_z : K # (case1 (z*) E2 E3) => K # (ev E2).
st_case1_s : K # (case1 (s* V1') E2 E3) => K # (ev (E3 V1')).

% Pairs
st_pair : K # (ev (pair E1 E2)) => (K ; [x1:val] pair1 x1 E2) # (ev E1).
st_pair1 : K # (pair1 V1 E2) => (K ; [x2:val] return (pair* V1 x2)) # (ev E2).
```

```
st_fst : K # (ev (fst E)) => (K ; [x:val] fst1 x) # (ev E).
st_fst1 : K # (fst1 (pair* V1 V2)) => K # (return V1).
st_snd : K # (ev (snd E)) => (K ; [x:val] snd1 x) # (ev E).
st_snd1 : K # (snd1 (pair* V1 V2)) => K # (return V2).

% Functions
st_lam : K # (ev (lam E)) => K # (return (lam* E)).
st_app : K # (ev (app E1 E2)) => (K ; [x1:val] app1 x1 E2) # (ev E1).
st_app1 : K # (app1 V1 E2) => (K ; [x2:val] app2 V1 x2) # (ev E2).
st_app2 : K # (app2 (lam* E1') V2) => K # (ev (E1' V2)).

% Definitions
st_letv : K # (ev (letv E1 E2)) => (K ; [x1:val] ev (E2 x1)) # (ev E1).
st_letn : K # (ev (letn E1 E2)) => K # (ev (E2 E1)).

% Recursion
st_fix : K # (ev (fix E)) => K # (ev (E (fix E))).

% Values
st_vl : K # (ev (vl V)) => K # (return V).

% Return Instructions
st_return : (K ; C) # (return V) => K # (C V).
st_init : (init) # (return V) => (answer V).
```

Multi-step computation sequences could be represented as lists of single step transitions. However, we would like to use dependent types to guarantee that, in a valid computation sequence, the result state of one transition matches the start state of the next transition. This is difficult to accomplish using a generic type of lists; instead we introduce specific instances of this type which are structurally just like lists, but have strong internal validity conditions.

$S \overset{*}{\Longrightarrow} S'$	S goes to S' in zero or more steps
$e \overset{c}{\hookrightarrow} v$	e evaluates to v using the continuation machine

$$\frac{}{S \overset{*}{\Longrightarrow} S}\,\mathsf{stop} \qquad \frac{S \Longrightarrow S' \qquad S' \overset{*}{\Longrightarrow} S''}{S \overset{*}{\Longrightarrow} S''}\,\mathsf{step}$$

$$\frac{\mathbf{init} \diamond e \overset{*}{\Longrightarrow} \mathbf{answer}\ v}{e \overset{c}{\hookrightarrow} v}\,\mathsf{cev}$$

We would like the implementation to be operational, that is, queries of the form `?- ceval` $\ulcorner e \urcorner$ `V.` should compute the value `V` of a given e. This means the

$S \Longrightarrow S'$ should be the first subgoal and hence the second argument of the step rule. In addition, we employ a visual trick to display computation sequences in a readable format by representing the step rule as a left associative infix operator.

```
=>* : state -> state -> type.          %name =>* C
%infix none 5 =>*

stop : S =>* S.
<< : S =>* S''
      <- S => S'
      <- S' =>* S''.
%infix left 5 <<

ceval : exp -> val -> type.            %name ceval CE

cev : ceval E V
       <- (init) # (ev E) =>* (answer V).
```

We then get a reasonable display of the sequence of computation steps which must be read from right to left.

```
?- C : (init) # (ev (app (lam [x] (vl x)) z)) =>* (answer V).
Solving...

V = z*,
C =
   stop << st_init << st_vl << st_app2 << st_return << st_z
      << st_app1 << st_return << st_lam << st_app.
```

The overall task now is to prove that $e \hookrightarrow v$ if and only if $e \overset{c}{\hookrightarrow} v$. In one direction we have to find a translation from tree-structured derivations $\mathcal{D} :: e \hookrightarrow v$ to sequential computations $\mathcal{C} :: \mathbf{init} \diamond e \overset{*}{\Longrightarrow} \mathbf{answer}\ v$. In the other direction we have to find a way to chop a sequential computation into pieces which can be reassembled into a tree-structured derivation.

We start with the easier of the two proofs. We assume that $e \hookrightarrow v$ and try to show that $e \overset{c}{\hookrightarrow} v$. This immediately reduces to showing that $\mathbf{init} \diamond e \overset{*}{\Longrightarrow} \mathbf{answer}\ v$. This does not follow directly by induction, since subcomputations will neither start from the initial computation nor return the final answer. If we generalize the claim to state that for all continuations K we have that $K \diamond e \overset{*}{\Longrightarrow} K \diamond \mathbf{return}\ v$, then it follows directly by induction, using some simple lemmas regarding the concatenation of computation sequences (see Exercise 6.18).

We can avoid explicit concatenation of computation sequences and obtain a more direct proof (and more efficient program) if we introduce an *accumulator argument*.

This argument contains the remainder of the computation, starting from the state $K \diamond \mathbf{return}\ v$. To the front of this given computation we add the computation from $K \diamond e \stackrel{*}{\Longrightarrow} K \diamond \mathbf{return}\ v$, passing the resulting computation as the next value of the accumulator argument. Translating this intuition to a logical statement requires explicitly universally quantifying over the accumulator argument.

Lemma 6.1 *For any closed expression e, value v and derivation $\mathcal{D} :: e \hookrightarrow v$, if $\mathcal{C}' :: K \diamond \mathbf{return}\ v \stackrel{*}{\Longrightarrow} \mathbf{answer}\ w$ for any K and w, then $\mathcal{C} :: K \diamond e \stackrel{*}{\Longrightarrow} \mathbf{answer}\ w$.*

Proof: The proof proceeds by induction on the structure of $\mathcal{D}$. Since the accumulator argument must already hold the remainder of the overall computation upon appeal to the induction hypothesis, we apply the induction hypothesis on the immediate subderivations of $\mathcal{D}$ in right-to-left order.

The proof is implemented by a type family

```
ccp : eval E V
       -> K # (return V) =>* (answer W)
       -> K # (ev E) =>* (answer W)
       -> type.
```

Operationally, the first argument is the induction argument, the second argument the accumlator, and the last the output argument.

We only show a couple of cases in the proof; the others follow in a similar manner.

Case:

$$\mathcal{D} = \frac{}{\mathbf{lam}\ x.\ e_1 \hookrightarrow \mathbf{lam}^*\ x.\ e_1}\ \textsf{ev\_lam}$$

$\mathcal{C}' :: K \diamond \mathbf{return}\ \mathbf{lam}^*\ x.\ e_1 \stackrel{*}{\Longrightarrow} \mathbf{answer}\ w$ — Assumption

$\mathcal{C} :: K \diamond \mathbf{lam}\ x.\ e_1 \Longrightarrow \mathbf{answer}\ w$ — By $\textsf{st\_lam}$ followed by $\mathcal{C}'$

In this case we have added a step $\textsf{st\_lam}$ to a computation; in the implementation, this will be an application of the $\textsf{step}$ rule for the $S \stackrel{*}{\Longrightarrow} S'$ judgment, which is written as << in infix notation. Recall that the reversal of the evaluation order means that computations (visually) proceed from right to left.

```
ccp_lam : ccp (ev_lam) C' (C' << st_lam).
```

Case:

$$\mathcal{D} = \frac{\begin{array}{c}\mathcal{D}_1\\ e_1 \hookrightarrow \mathbf{lam}\ x.\ e_1'\end{array} \qquad \begin{array}{c}\mathcal{D}_2\\ e_2 \hookrightarrow v_2\end{array} \qquad \begin{array}{c}\mathcal{D}_3\\ [v_2/x]e_1' \hookrightarrow v\end{array}}{e_1\ e_2 \hookrightarrow v}\ \textsf{ev\_app}$$

$\mathcal{C}' :: K \diamond \mathbf{return}\ v \stackrel{*}{\Longrightarrow} \mathbf{answer}\ w$ Assumption
$\mathcal{C}_3 :: K \diamond [v_2/x]e'_1 \stackrel{*}{\Longrightarrow} \mathbf{answer}\ w$ By ind. hyp. on $\mathcal{D}_3$ and $\mathcal{C}'$
$\mathcal{C}'_2 :: K; \lambda x_2.\ \mathbf{app}_2\ (\mathbf{lam}^*\ x.\ e'_1)\ x_2 \diamond \mathbf{return}\ v_2 \stackrel{*}{\Longrightarrow} \mathbf{answer}\ w$
By st_return and st_app2 followed by $\mathcal{C}_3$
$\mathcal{C}_2 :: K; \lambda x_2.\ \mathbf{app}_2\ (\mathbf{lam}^*\ x.\ e'_1)\ x_2 \diamond e_2 \stackrel{*}{\Longrightarrow} \mathbf{answer}\ w$
By ind. hyp. on $\mathcal{D}_2$ and $\mathcal{C}'_2$
$\mathcal{C}'_1 :: K; \lambda x_1.\ \mathbf{app}_1\ x_1\ e_2 \diamond \mathbf{return}\ \mathbf{lam}^*\ x.\ e'_1 \stackrel{*}{\Longrightarrow} \mathbf{answer}\ w$
By st_return and st_app1 followed by $\mathcal{C}_2$.
$\mathcal{C}_1 :: K; \lambda x_1.\ \mathbf{app}_1\ x_1\ e_2 \diamond e_1 \stackrel{*}{\Longrightarrow} \mathbf{answer}\ w$ By ind. hyp. on $\mathcal{D}_1$ and $\mathcal{C}'_1$
$\mathcal{C} :: K \diamond e_1\ e_2 \stackrel{*}{\Longrightarrow} \mathbf{answer}\ w$ By st_app followed by $\mathcal{C}_1$.

The implementation threads the accumulator argument, adding steps concerned with application as in the proof above.

```
ccp_app : ccp (ev_app D3 D2 D1) C' (C1 << st_app)
           <- ccp D3 C' C3
           <- ccp D2 (C3 << st_app2 << st_return) C2
           <- ccp D1 (C2 << st_app1 << st_return) C1.
```

□

From this, the completeness of the abstract machine follows directly.

Theorem 6.2 (Completeness of the Continuation Machine) *For any closed expression e and value v, if $e \hookrightarrow v$ then $e \stackrel{c}{\hookrightarrow} v$.*

Proof: We use Lemma 6.1 with $K = \mathbf{init}$, $w = v$, and $\mathcal{C}'$ the computation with st_init as the only step, to conclude that there is a computation $\mathcal{C} :: \mathbf{init} \diamond e \stackrel{*}{\Longrightarrow} \mathbf{answer}\ v$. Therefore, by rule cev, $e \stackrel{c}{\hookrightarrow} v$.

The implementation is straightforward, using ccp, the implementation of the main lemma above.

```
ceval_complete : eval E V -> ceval E V -> type.

cevcp : ceval_complete D (cev C)
         <- ccp D (stop << st_init) C.
```

□

Now we turn our attention to the soundness of the continuation machine: whenever it produces a value v then the natural semantics can also produce the value v from the same expression. This is more difficult to prove than completeness. The reason is that in the completeness proof, every subderivation of $\mathcal{D} :: e \hookrightarrow v$ can inductively be translated to a sequence of computation steps, but not every sequence of computation steps corresponds to an evaluation. For example, the partial computation

$$K \diamond e_1\ e_2 \overset{*}{\Longrightarrow} K; \lambda x_1.\ \mathbf{app}_1\ x_1\ e_2 \diamond e_1$$

represents only a fragment of an evaluation. In order to translate a computation sequence we must ensure that it is sufficiently long. A simple way to accomplish this is to require that the given computation goes all the way to a final answer. Thus, we have a state $K \diamond e$ at the beginning of a computation sequence $\mathcal{C}$ to a final answer w, there must be some initial segment of $\mathcal{C}'$ which corresponds to an evaluation of e to a value v, while the remaining computation goes from $K \diamond \mathbf{return}\ v$ to the final answer w. This can then be proved by induction.

Lemma 6.3 *For any continuation K, closed expression e and value w, if $\mathcal{C} :: K \diamond e \overset{*}{\Longrightarrow}$ **answer** w then there is a value v a derivation $\mathcal{D} :: e \hookrightarrow v$, and a subcomputation $\mathcal{C}'$ of $\mathcal{C}$ of the form $K \diamond$ **return** $v \overset{*}{\Longrightarrow}$ **answer** w.*

Proof: By complete induction on the structure of $\mathcal{C}$. Here *complete induction*, as opposed to a simple structural induction, means that we can apply the induction hypothesis to any subderivation of $\mathcal{C}$, not just to the immediate subderivations. It should be intuitively clear that this is a valid induction principle (see also Section 6.4).

In the implementation we have chosen not to represent the evidence for the assertion that $\mathcal{C}'$ is a subderivation of $\mathcal{C}$. This can be added, either directly to the implementation or as a higher-level judgment (see Exercise **??**). This information is not required to execute the proof on specific computation sequences, although it is critical for seeing that it always terminates.

```
csd : K # (ev E) =>* (answer W)
       -> eval E V
       -> K # (return V) =>* (answer W)
       -> type.
```

We only show a few typical cases; the others follow similarly.

Case: The first step of $\mathcal{C}$ is st_lam followed by $\mathcal{C}_1 :: K \diamond \mathbf{return}\ \mathbf{lam}^*\ x.\ e \overset{*}{\Longrightarrow}$ **answer** w.

In this case we let $\mathcal{D} =$ ev_lam and $\mathcal{C}' = \mathcal{C}_1$. The implementation (where step is written as << in infix notation):

```
csd_lam : csd (C' << st_lam) (ev_lam) C'.
```

Case: The first step of $\mathcal{C}$ is st_app followed by $\mathcal{C}_1 :: K; \lambda x_1.\ \mathbf{app}_1\ x_1\ e_2 \diamond e_1 \overset{*}{\Longrightarrow}$ **answer** w, where $e = e_1\ e_2$.

$\mathcal{D}_1 :: e_1 \hookrightarrow v_1$ for some v_1 and
$\mathcal{C}_1' :: K; \lambda x_1.\, \mathbf{app}_1\ x_1\ e_2 \diamond \mathbf{return}\ v_1 \overset{*}{\Longrightarrow} \mathbf{answer}\ w$ By ind. hyp. on $\mathcal{C}_1$
$\mathcal{C}_1'' :: K \diamond \mathbf{app}_1\ v_1\ e_2 \overset{*}{\Longrightarrow} \mathbf{answer}\ w$ By inversion on $\mathcal{C}_1'$
$\mathcal{C}_2 :: K; \lambda x_2.\, \mathbf{app}_2\ v_1\ x_2 \diamond e_2 \overset{*}{\Longrightarrow} \mathbf{answer}\ w$ By inversion on $\mathcal{C}_1''$
$\mathcal{D}_2 :: e_2 \hookrightarrow v_2$ form some v_2 and
$\mathcal{C}_2' :: K; \lambda x_2.\, \mathbf{app}_2\ v_1\ x_2 \diamond \mathbf{return}\ v_2 \overset{*}{\Longrightarrow} \mathbf{answer}\ w$ By ind. hyp. on $\mathcal{C}_2$
$\mathcal{C}_2'' :: K \diamond \mathbf{app}_2\ v_1\ v_2 \overset{*}{\Longrightarrow} \mathbf{answer}\ w$ By inversion on $\mathcal{C}_2'$
$v_1 = \mathbf{lam}\, x.\, e_1'$ and
$\mathcal{C}_3 :: K \diamond [v_2/x]e_1' \overset{*}{\Longrightarrow} \mathbf{answer}\ w$ By inversion on $\mathcal{C}_2''$
$\mathcal{D}_3 :: [v_2/x]e_1' \hookrightarrow v$ for some v and
$\mathcal{C}' :: K \diamond \mathbf{return}\ v \overset{*}{\Longrightarrow} \mathbf{answer}\ w$ By ind. hyp on $\mathcal{C}_3$
$\mathcal{D} :: e_1\ e_2 \hookrightarrow v$ By rule `ev_app` from $\mathcal{D}_1$, $\mathcal{D}_2$, and $\mathcal{D}_3$.

The evaluation $\mathcal{D}$ and computation sequence $\mathcal{C}'$ now satisfy the requirements of the lemma. The appeals to the induction hypothesis are all legal, since $\mathcal{C} > \mathcal{C}_1 > \mathcal{C}_1'' > \mathcal{C}_2 > \mathcal{C}_2' > \mathcal{C}_2'' > \mathcal{C}_3 > \mathcal{C}'$, where $>$ is the subcomputation judgment. Each of the subcomputation judgments in this chain follows either immediately, or by induction hypothesis.

The implementation:

```
csd_app : csd (C1 << st_app) (ev_app D3 D2 D1) C'
           <- csd C1 D1 (C2 << st_app1 << st_return)
           <- csd C2 D2 (C3 << st_app2 << st_return)
           <- csd C3 D3 C'.
```

□

Once again, the main theorem follows directly from the lemma.

Theorem 6.4 (Soundness of the Continuation Machine) *For any closed expression e and value v, if $e \overset{c}{\hookrightarrow} v$ then $e \hookrightarrow v$.*

Proof: By inversion, $\mathcal{C} :: \mathbf{init} \diamond e \overset{*}{\Longrightarrow} \mathbf{answer}\ v$. By Lemma 6.3 there is a derivation $\mathcal{D} :: e \hookrightarrow v'$ and $\mathcal{C}' :: \mathbf{init} \diamond \mathbf{return}\ v' \overset{*}{\Longrightarrow} \mathbf{answer}\ v$ for some v'. By inversion on $\mathcal{C}'$ we see that $v = v'$ and therefore $\mathcal{D}$ satisfies the requirements of the theorem.

```
ceval_sound : ceval E V -> eval E V -> type.

cevsd : ceval_sound (cev C) D
         <- csd C D (stop << st_init).
```

□

6.6 Relating Relations between Derivations

Upon inspection we see that the higher-level judgments implementing our soundness and completeness proofs are similar. Consider:

```
csd : K # (ev E) =>* (answer W)
       -> eval E V
       -> K # (return V) =>* (answer W)
       -> type.

ccp : eval E V
       -> K # (return V) =>* (answer W)
       -> K # (ev E) =>* (answer W)
       -> type.
```

The families `csd` and `ccp` relate derivations of exactly the same judgments, the only difference being their order. But the analogy is even stronger. By inspecting the (higher-level) rules for application in each of these families,

```
csd_app : csd (C1 << st_app) (ev_app D3 D2 D1) C'
           <- csd C1 D1 (C2 << st_app1 << st_return)
           <- csd C2 D2 (C3 << st_app2 << st_return)
           <- csd C3 D3 C'.

ccp_app : ccp (ev_app D3 D2 D1) C' (C1 << st_app)
           <- ccp D3 C' C3
           <- ccp D2 (C3 << st_app2 << st_return) C2
           <- ccp D1 (C2 << st_app1 << st_return) C1.
```

we conjecture the the premisses are also identical, varying only in their order. This turns out to be true. Thus the translations between evaluations and computation sequences form a bijection: translating in either of the two directions and then back yields the original derivation.

We find it difficult to make such an argument precise in informal mathematical language, since we have not reified the translations between evaluation trees and computation sequences, except in LF. However, this can be done as in Section 3.7 or directly via an ordinary mathematical function. Either of these is tedious and serves only to transcribe the Elf implementation of the soundness and completeness proofs into another language, so we will skip this step and simply state the theorem directly.

Theorem 6.5 *Evaluations* $\mathcal{D} :: e \hookrightarrow v$ *and computations* $\mathcal{CE} :: e \overset{c}{\hookrightarrow} v$ *are in bijective correspondence.*

Proof: We examine the translations implicit in the constructive proofs of completeness (Lemma 6.1 and Theorem 6.2) and soundness (Lemma 6.3 and Theorem 6.4) to show that they are inverses of each other. More formally, the argument would proceed by induction on the definition of the translation in each direction.

The implementation makes this readily apparent. We only show the declaration of this level 3[1] type family and the cases for functions.

```
peq : csd C D C'
       -> ccp D C' C
       -> type.

% Functions
peq_lam : peq (csd_lam) (ccp_lam).
peq_app : peq (csd_app SD3 SD2 SD1) (ccp_app CP1 CP2 CP3)
           <- peq SD1 CP1
           <- peq SD2 CP2
           <- peq SD3 CP3.
```

Note that the judgment `peq` must be read in both directions in order to obtain the bijective property. At the top level, the relation directly reduces to the inductively defined judgment above.

```
proof_equiv : ceval_sound CE D -> ceval_complete D CE -> type.

pequiv : proof_equiv (cevsd SD) (cevcp CP)
          <- peq SD CP.
```

□

6.7 Contextual Semantics

One might ask if the continuations we have introduced in Section 6.5 are actually necessary to describe a small-step semantics. That is, can we describe a semantics where the initial expression e is successively transformed until we arrive at a value v? The answer is yes, although each small step in this style of semantic description is complex: First we have to isolate a subexpression r of e to be reduced next, then we actually perform the reduction from r to r', and finally we reconstitute an expression e' in an occurrence of r has been replaced by r'.

This style of semantic description is called a *contextual semantics*, since its central idea is the decomposition of an expression e into an *evaluation context* C and a *redex* r. The evaluation context C contains a *hole* which, when filled with

[1] [*check terminology*]

r yields e, and when filled with the reduct r' yields e', the next expression in the computation sequence. By restricting evaluation contexts appropriately we can guarantee that the semantics remains deterministic.

Unlike the other forms of semantic description we have encountered, the contextual semantics makes the relation between the reduction rules of a λ-calculus (as we will see in Section 7.4) and the operational semantics of a functional language explicit. On the other hand, each "small" step in this semantics requires complex auxiliary operations and it is therefore not as direct as either the natural semantics or the continuation machine.

We first identify the reductions on the representation of Mini-ML which separates values and expressions (see Section 6.5). These reductions are of two kinds: *essential reductions* which come directly from an underlying λ-calculus, and *auxiliary reductions* which are used to manage the passage from expression to their corresponding values. The latter ones are usually omitted in informal presentations, but we do not have this luxury in Elf due to the absence of any form of subtyping in LF.

$$e \Longrightarrow e' \quad e \text{ reduces to } e' \text{ in one step}$$

First, the essential reductions. Note that every value is regarded as an expression, but the corresponding coercion is not shown in the concrete syntax.

$$\begin{array}{lcll}
\mathsf{red_case_z} & :: & \mathbf{case}\ \mathbf{z}^*\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\ x \Rightarrow e_3 & \Longrightarrow\ e_2 \\
\mathsf{red_case_s} & :: & \mathbf{case}\ \mathbf{s}^*\ v_1'\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\ x \Rightarrow e_2 & \Longrightarrow\ [v_1'/x]e_3 \\
\mathsf{red_fst} & :: & \mathbf{fst}\ \langle v_1, v_2\rangle^* & \Longrightarrow\ v_1 \\
\mathsf{red_snd} & :: & \mathbf{snd}\ \langle v_1, v_2\rangle^* & \Longrightarrow\ v_2 \\
\mathsf{red_app} & :: & (\mathbf{lam}\ x.\ e_1')\ v_2 & \Longrightarrow\ [v_2/x]e_1' \\
\mathsf{red_letv} & :: & \mathbf{let\ val}\ x = v_1\ \mathbf{in}\ e_2 & \Longrightarrow\ [v_1/x]e_2 \\
\mathsf{red_letn} & :: & \mathbf{let\ name}\ u = e_1\ \mathbf{in}\ e_2 & \Longrightarrow\ [e_1/u]e_2 \\
\mathsf{red_fix} & :: & \mathbf{fix}\ u.\ e & \Longrightarrow\ [\mathbf{fix}\ u.\ e/u]e
\end{array}$$

In addition, we have the following auxiliary reductions.

$$\begin{array}{lclcl}
\mathsf{red_z} & :: & \mathbf{z} & \Longrightarrow & \mathbf{z}^* \\
\mathsf{red_s} & :: & \mathbf{s}\ v & \Longrightarrow & \mathbf{s}^*\ v \\
\mathsf{red_pair} & :: & \langle v_1, v_2\rangle & \Longrightarrow & \langle v_1, v_2\rangle^* \\
\mathsf{red_lam} & :: & \mathbf{lam}\ x.\ e & \Longrightarrow & \mathbf{lam}^*\ x.\ e
\end{array}$$

Expressions which are already values cannot be reduced any further. We say that e is a redex if there is an e' such that $e \Longrightarrow e'$.

The implementation of the one-step reduction and redex judgments are straightforward.

```
==> : exp -> exp -> type.
%infix none 8 ==>
```

```
red_z      : z                           ==> (vl z*).
red_s      : s (vl V)                    ==> vl (s* V).
red_case_z : case (vl z*) E2 E3          ==> E2.
red_case_s : case (vl (s* V1')) E2 E3    ==> E3 V1'.

red_pair   : pair (vl V1) (vl V2)        ==> vl (pair* V1 V2).
red_fst    : fst (vl (pair* V1 V2))      ==> (vl V1).
red_snd    : snd (vl (pair* V2 V2))      ==> (vl V2).

red_lam    : lam E                       ==> vl (lam* E).
red_app    : app (vl (lam* E1')) (vl V2) ==> E1' V2.

red_letv   : letv (vl V1) E2             ==> E2 V1.
red_letn   : letn E1 E2                  ==> E2 E1.

red_fix    : fix E                       ==> E (fix E).

% no red_vl rule

%%% Redices
redex : exp -> type.

rdx : redex E
       <- E ==> E'.
```

The specification of the reduction judgment already incorporates some of the basic decision regarding the semantics of the language. For example, we can see that pairs are eager, since **fst** e can be reduced only when e is a value (and thus consists of a pair of values). Similarly, the language is call-by-value (rule red_app) and **let name** is call-by-name (rule red_letn). However, we have not yet specified in which order subexpressions must be evaluated. This is fixed by specifying precisely in which context a redex must appear before it can be reduced. In other words, we have to define *evaluation contexts*. We write $[\,]$ for the hole in the evaluation context, filling the hole in an evaluation context C with an expression r is written as $C[r]$.

Evaluation Contexts	C	$::=$	$[\,]$	*Hole*
			$\mid \mathbf{s}\ C \mid \mathbf{case}\ C\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2 \mid \mathbf{s}\ x \Rightarrow e_3$	*Natural numbers*
			$\mid \langle C, e_2\rangle \mid \langle v_1, C\rangle \mid \mathbf{fst}\ C \mid \mathbf{snd}\ C$	*Pairs*
			$\mid C\ e_2 \mid v_1\ C$	*Functions*
			$\mid \mathbf{let\ val}\ x = C\ \mathbf{in}\ e_2$	*Definitions*

A hole $[\,]$ is the only base case in this definition. It is instructive to consider which kinds of occurrences of a hole in a context are ruled out by this definition. For example, we cannot reduce any redex in the second component of a pair until the first component has been reduced to a value (clause $\langle v_1, C\rangle$). Furthermore, we cannot reduce in a branch of a **case** expression, in the body of a **let val** expression, or anywhere in the scope of a **let name** or **fix** construct.

Single step reduction can now be extended to one-step computation.

$$e \Longrightarrow_1 e' \quad e \text{ goes to } e' \text{ in one computation step}$$

$$\frac{r \Longrightarrow r'}{C[r] \Longrightarrow_1 C[r']}\,\mathsf{ostp}$$

The straightforward implementation requires the check that a function from expressions to expression is a valid evaluation context from Exercise **??**, written here as `evctx`.

```
evctx : (exp -> exp) -> type.
% Exercise...

one_step : exp -> exp -> type.

ostp : one_step (C R) (C R')
        <- evctx C
        <- redex R
        <- R ==> R'.
```

An operationally more direct implementation uses a splitting judgment which relates an expression e to an evaluation context C and a redex r such that $e = C[r]$. We only show this judgment in its implementation. It illustrates how index functions can be manipulated in Elf programs. The splitting judgment above uses an auxiliary judgment which uses an accumulator argument for the evaluation context C which is computed as we descend into the expression e. This auxiliary judgment relates C and e (initially: $C = [\,]$ and e is the given expression) to a C' and e' such that e' is a redex and $C[e] = C'[e']$.

```
%%% Splitting an Expression
split : (exp -> exp) -> exp -> (exp -> exp) -> exp -> type.

%{ split C E C' E'
   Evaluation context C and expression E are given,
   C' and E' are constructed.
   Invariant: (C E) == (C' E')
}%
```

```
% Redices
sp_redex : split C E C E
            <- redex E.

% Natural Numbers
% no sp_z
sp_s : split C (s E1) C' E'
        <- split ([h:exp] C (s h)) E1 C' E'.

sp_case : split C (case E1 E2 E3) C' E'
           <- split ([h:exp] C (case h E2 E3)) E1 C' E'.

% Pairs
sp_pair2 : split C (pair (vl V1) E2) C' E'
            <- split ([h:exp] C (pair (vl V1) h)) E2 C' E'.
sp_pair1 : split C (pair E1 E2) C' E'
            <- split ([h:exp] C (pair h E2)) E1 C' E'.
sp_fst   : split C (fst E) C' E'
            <- split ([h:exp] C (fst h)) E C' E'.
sp_snd   : split C (snd E) C' E'
            <- split ([h:exp] C (snd h)) E C' E'.

% Functions
% no sp_lam
sp_app2 : split C (app (vl V1) E2) C' E'
           <- split ([h:exp] C (app (vl V1) h)) E2 C' E'.
sp_app1 : split C (app E1 E2) C' E'
           <- split ([h:exp] C (app h E2)) E1 C' E'.

% Definitions
sp_letv : split C (letv E1 E2) C' E'
           <- split ([h:exp] C (letv h E2)) E1 C' E'.
% no sp_letn

% Recursion
% no sp_fix

% Values
% no sp_vl

%%% Top-Level Splitting
```

```
split_exp : exp -> (exp -> exp) -> exp -> type.

spe : split_exp E C E'
       <- split ([h:exp] h) E C E'.
```

The splitting judgment is then used in the definition of contextual evaluation in lieu of an explicit check.

```
one_step : exp -> exp -> type.

ostp : one_step E (C R')
        <- split_exp E C R
        <- R ==> R'.

%%% Full Contextual Evaluation

xeval : exp -> val -> type.

xev_vl : xeval (vl V) V.
xev_step : xeval E V
            <- one_step E E'
            <- xeval E' V.
```

Evaluation contexts bear a close resemblance to the continuations in Section 6.5. In fact, there is a bijective correspondence between evaluation contexts (following the grammar above) and the kind of continuation which arises from computation starting with the initial continuation **init**. We do not investigate this relationship here.[2]

6.8 Additional Exercises

Exercise 6.17 Show that the purely expression-based natural semantics of Section 2.3 is equivalent to the one based on a separation between expressions and values in Section 6.5. Implement your proof, including all necessary lemmas, in Elf.

Exercise 6.18 Carry out the alternative proof of completeness of the continuation machine sketched on page 191. Implement the proof and all necessary lemmas in Elf.

[2][*an earlier version of these notes contained a buggy translation here. please search and destroy!*]

Exercise 6.19 Do the equivalence proof in Lemma 6.1 and the alternative in Exercise 6.18 define the same relation between derivations? If so, exhibit the bijection in the form of a higher-level judgment relating the Elf implementations as in Section 6.6. Be careful to write out necessary lemmas regarding concatenation. You may restrict yourself to functional abstraction, application, and the necessary computation rules.

Exercise 6.20 [*an exercise about the mechanical translation from small-step to big-step semantics*]