Chapter 6

Compilation

The model of evaluation introduced in Section 2.3 and formalized in Section 3.6 builds only on the expressions of the Mini-ML language itself. This leads very naturally to an *interpreter* in Elf which is given in Section 4.3. Our specification of the operational semantics is in the style of *natural semantics* which very often lends itself to direct, though inefficient, execution. The inefficiency of the interpreter in 4.3 is more than just a practical issue, since it is clearly the wrong model if we would like to reason about the complexity of functions defined in Mini-ML. One can refine the evaluation model in two ways: one is to consider more efficient interpreters (see Exercises 2.11 and 4.2), another is to consider compilation. In this chapter we pursue the latter possibility and describe and prove the correctness of a compiler for Mini-ML.

In order to define a compiler we need a *target language* for compilation, that is, the language into which programs in the source language are translated. This target language has its own operational semantics, and we must show the correctness of compilation with respect to these two languages and their semantics. The ultimate target language for compilation is determined by the architecture and instruction set of the machine the programs are to be run on. In order to insulate compilers from the details of particular machine architectures it is advisable to design an intermediate language and execution model which is influenced by a set of target architectures and by constructs of the source language. We refer to this intermediate level as an *abstract machine*. Abstract machine code can then itself either be interpreted or compiled further to actual machine code. In this chapter we take a stepwise approach to compilation, using two intermediate forms between Mini-ML and a variant of the SECD machine [Lan64] which is also related to the Categorical Abstract Machine (CAM) [CCM87]. This decomposition simplifies the correctness proofs and localizes ideas which are necessary to understand the compiler in its totality.

The material presented in this chapter follows work by Hannan [HM90, Han91], both in general approach and in many details. An extended abstract that also addresses correctness issues and methods of formalization can be found in [HP92]. A different approach to compilation using *continuations* may be found in Section 9.2.

6.1 An Environment Model for Evaluation

The evaluation judgment $e \hookrightarrow v$ requires that all information about the state of the computation is contained in the Mini-ML expression e. The application of a function formed by λ-abstraction, $\mathbf{lam}\ x.\ e$, to an argument v thus requires the substitution of v for x in e and evaluation of the result. In order to avoid this substitution it may seem reasonable to formulate evaluation as a hypothetical judgment (e is evaluated under the hypothesis that x evaluates to v) but this attempt fails (see Exercise 6.1). Instead, we allow free variables in expressions which are given values in an *environment*, which is explicitly represented as part of a revised evaluation judgment. Variables are evaluated by looking up their value in the environment; previously we always eliminated them by substitution, so no separate rule was required. However, this leads to a problem with the scope of variables. Consider the expression $\mathbf{lam}\ y.\ x$ in an environment that binds x to $\mathbf{z}$. According to our natural semantics the value of this expression should be $\mathbf{lam}\ y.\ \mathbf{z}$, but this requires the substitution of $\mathbf{z}$ for x. Simply returning $\mathbf{lam}\ y.\ x$ is incorrect if this value may later be interpreted in an environment in which x is not bound, or bound to a different value. The practical solution is to return a *closure* consisting of an environment K and an expression $\mathbf{lam}\ y.\ e$. K must contain at least all the variables free in $\mathbf{lam}\ y.\ e$. We ignore certain questions of efficiency in our presentation and simply pair up the complete current environment with the expression to form the closure.

This approach leads to the question how to represent environments and closures. A simple solution is to represent an environment as a list of values and a variable as a pointer into this list. It was de Bruijn's idea [dB72] to implement such pointers as natural numbers where n refers to the n^{th} element of the environment list. This works smoothly if we also represent bound variables in this fashion: an occurence of a bound variable points backwards to the place where it is bound. This pointer takes the form of a positive integer, where 1 refers to the innermost binder and 1 is added for every binding encountered when going upward through the expression. For example

$$\mathbf{lam}\ x.\ \mathbf{lam}\ y.\ x\ (\mathbf{lam}\ z.\ y\ z)$$

would be written as

$$\Lambda\ (\Lambda\ (2\ (\Lambda\ (2\ 1))))$$

where Λ binds an (unnamed) variable. In this form expressions that differ only in the names of their bound variables are syntactically identical. If we restrict attention to pure λ-terms for the moment, this leads to the definition

$$\begin{array}{rrcl} \text{de Bruijn Expressions} & D & ::= & n \mid \Lambda D \mid D_1\ D_2 \\ \text{de Bruijn Indices} & n & ::= & 1 \mid 2 \mid \ldots \end{array}$$

Instead of using integers and general arithmetic operations on them, we use only the integer 1 to refer to the innermost element of the environment and the operator $\uparrow$ (read: shift, written in post-fix notation) to increment variable references. That is, the integer $n + 1$ is represented as

$$1 \underbrace{\uparrow \cdots \uparrow}_{n \text{ times}}.$$

But $\uparrow$ can also be applied to other expressions, in effect raising each integer in the expression by 1. For example, the expression

$$\textbf{lam}\ x.\ \textbf{lam}\ y.\ x\ x$$

can be represented by

$$\Lambda\ (\Lambda\ ((1\uparrow)\ (1\uparrow)))$$

or

$$\Lambda\ (\Lambda\ ((1\ 1)\uparrow)).$$

This is a very simple form of a λ-calculus with *explicit substitutions* where $\uparrow$ is the only available substitution (see [ACCL91]).

$$\text{Modified de Bruijn Expressions} \quad F \quad ::= \quad 1 \mid F\uparrow \mid \Lambda F \mid F_1\ F_2$$

We use the convention that the postfix operator $\uparrow$ binds stronger than application which in turn binds stronger that the prefix operator Λ. Thus the two examples above can be written as $\Lambda\ \Lambda\ 1\uparrow\ 1\uparrow$ and $\Lambda\ \Lambda\ (1\ 1)\uparrow$, respectively.

The next step is to introduce environments. These depend on *values* and *vice versa*, since a closure is a pair of an environment and an expression, and an environment is a list of values. This can be carried to the extreme: in the Categorical Abstract Machine (CAM), for example, environments are built as iterated pairs and are thus values. Our representation will not make this identification. Since we have simplified our language to a pure λ-calculus, the only kind of value which can arise is a closure.

$$\begin{array}{rrcl} \text{Environments} & K & ::= & \cdot \mid K; W \\ \text{Values} & W & ::= & \{K, F\} \end{array}$$

We write w for parameters ranging over values. During the course of evaluation, only closures over Λ-expressions will arise, that is, all closures have the form $\{K, \Lambda F'\}$ (see Exercise 6.2).

The specification of modified de Bruijn expressions, values, and environments is straightforward. The abstract syntax is now first-order, since the language does not contain any name binding constructs.

```
exp'    : type.  %name exp' F

1       : exp'.
^       : exp' -> exp'.  %postfix 20 ^
lam'    : exp' -> exp'.
app'    : exp' -> exp' -> exp'.

env     : type.  %name env K
val     : type.  %name val W

empty   : env.
;       : env -> val -> env.   %infix left 10 ;

clo     : env -> exp' -> val.
```

There are two main judgments that achieve compilation: one relates a de Bruijn expression F in an environment K to an ordinary expression e, another relates a value W to an expression v. We also need an evaluation judgment relating de Bruijn expressions and values in a given environment.

$K \vdash F \leftrightarrow e$	F translates to e in environment K
$W \Leftrightarrow v$	W translates to v
$K \vdash F \hookrightarrow W$	F evaluates to W in environment K

When we evaluate a given expression e using these judgments, we translate it to a de Bruijn expression F in the empty environment, evaluate F in the empty environment to obtain a value W, and then translate W to an expression v in the original language. This is depicted in the following diagram.

$$\begin{tikzcd}
e \arrow[r, "\mathcal{D} :: e \hookrightarrow v"] \arrow[d, swap, "\mathcal{C} :: \cdot \vdash F \leftrightarrow e"] & v \\
F \arrow[r, swap, "\mathcal{D}' :: \cdot \vdash F \hookrightarrow W"] & W \arrow[u, swap, "\mathcal{U} :: W \Leftrightarrow v"]
\end{tikzcd}$$

The correctness of this phase of compilation can then be decomposed into two statements. For *completeness*, we assume that $\mathcal{D}$ and therefore e and v are given, and we would like to show that there exist $\mathcal{C}$, $\mathcal{D}'$, and $\mathcal{U}$ completing the diagram. This means that for every evaluation of e to a value v, this value could also have been produced by evaluating the compiled expression and translating the resulting value back to the original language. The dual of this is *soundness*: we assume that $\mathcal{C}$, $\mathcal{D}'$ and $\mathcal{U}$ are given and we have to show that an evaluation $\mathcal{D}$ exists. That is, every value which can be produced by compilation and evaluation of compiled expressions can also be produced by direct evaluation.

We will continue to restrict ourselves to expressions built up only from abstraction and application. When we generalize this later only the case of fixpoint expressions will introduce an essential complication. First we define evaluation of de Bruijn expressions in an environment K, written as $K \vdash F \hookrightarrow W$. The variable 1 refers to the first value in the environment (counting from right to left); its evaluation just returns that value.

$$\frac{}{K; W \vdash 1 \hookrightarrow W}\,\textsf{fev\_1}$$

The meaning of an expression $F{\uparrow}$ in an environment $K; W$ is the same as the meaning of F in the environment K. Intuitively, the environment references from F into K are shifted by one. The typical case is one where a reference to the n^{th} value in K is represented by the expression $1{\uparrow}\cdots{\uparrow}$, where the shift operator is applied $n-1$ times.

$$\frac{K \vdash F \hookrightarrow W}{K; W' \vdash F{\uparrow} \hookrightarrow W}\,\textsf{fev\_}{\uparrow}$$

A functional abstraction usually immediately evaluates to itself. Here this is insufficient, since an expression ΛF may contain references to the environment K. Thus we need to combine the environment K with ΛF to produce a closed (and self-contained) value.

$$\frac{}{K \vdash \Lambda F \hookrightarrow \{K, \Lambda F\}}\,\textsf{fev\_lam}$$

In order to evaluate $F_1\ F_2$ in an environment K we evaluate both F_1 and F_2 in that environment, yielding the closure $\{K', \Lambda F_1'\}$ and value W_2, respectively. We then add W_2 to the environment K', in effect binding the variable previously bound by Λ in $\Lambda F_1'$ to W_2 and then evaluate F_1' in the extended environment to obtain the overall value W.

$$\frac{K \vdash F_1 \hookrightarrow \{K', \Lambda F_1'\} \qquad K \vdash F_2 \hookrightarrow W_2 \qquad K'; W_2 \vdash F_1' \hookrightarrow W}{K \vdash F_1\ F_2 \hookrightarrow W}\,\textsf{fev\_app}$$

Here is the implementation of this judgment as the type family `feval` in Elf.

```
feval : env -> exp' -> val -> type.  %name feval D

% Variables
fev_1 : feval (K ; W) 1 W.
fev_^ : feval (K ; W') (F ^) W
          <- feval K F W.
```

```
% Functions
fev_lam : feval K (lam' F) (clo K (lam' F)).
fev_app : feval K (app' F1 F2) W
           <- feval K F1 (clo K' (lam' F1'))
           <- feval K F2 W2
           <- feval (K' ; W2) F1' W.
```

We have written this signature in a way that emphasizes its operational reading, because it serves as an implementation of an interpreter. As an example, consider the evaluation of the expression $(\Lambda\ (\Lambda\ (1\uparrow)))\ (\Lambda\ 1)$, which is a representation of $(\mathbf{lam}\ x.\ \mathbf{lam}\ y.\ x)\ (\mathbf{lam}\ v.\ v)$.

```
?- D : feval empty (app' (lam' (lam' (1 ^))) (lam' 1)) W.

W = clo (empty ; clo empty (lam' 1)) (lam' (1 ^)),
D = fev_app fev_lam fev_lam fev_lam.
```

The resulting closure, $\{(\cdot; \{\cdot, \Lambda 1\}), \Lambda(1\uparrow)\}$, represents the de Bruijn expressions $\Lambda(\Lambda 1)$, since $(1\uparrow)$ refers to the first value in the environment.

The translation between ordinary and de Bruijn expressions is specified by the following rules which employ a parametric and hypothetical judgment.

$$\frac{K \vdash F_1 \leftrightarrow e_1 \qquad K \vdash F_2 \leftrightarrow e_2}{K \vdash F_1\ F_2 \leftrightarrow e_1\ e_2}\,\mathsf{tr\_app} \qquad \frac{\begin{array}{c}\overline{w \Leftrightarrow x}\,u \\ \vdots \\ K; w \vdash F \leftrightarrow e\end{array}}{K \vdash \Lambda F \leftrightarrow \mathbf{lam}\ x.\ e}\,\mathsf{tr\_lam}^{w,x,u}$$

$$\frac{W \Leftrightarrow e}{K; W \vdash 1 \leftrightarrow e}\,\mathsf{tr\_1} \qquad \frac{K \vdash F \leftrightarrow e}{K; W \vdash F\uparrow \leftrightarrow e}\,\mathsf{tr\_}\!\uparrow$$

where the rule tr_lam is restricted to the case where w and x are new parameters not free in any other hypothesis, and u is a new label. The translation of values is defined by a single rule in this language fragment.

$$\frac{K \vdash \Lambda F \leftrightarrow \mathbf{lam}\ x.\ e}{\{K, \Lambda F\} \Leftrightarrow \mathbf{lam}\ x.\ e}\,\mathsf{vtr\_lam}$$

As remarked earlier this translation can be non-deterministic if K and e are given and F is to be generated. This is the direction in which this judgment would be used for compilation. Here is an example of a translation.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{w \Leftrightarrow x}\,u}{\cdot; w \vdash 1 \leftrightarrow x}\,\mathsf{tr\_1}}{\cdot; w; w' \vdash 1\uparrow \leftrightarrow x}\,\mathsf{tr\_}\!\uparrow}{\cdot; w \vdash \Lambda 1\uparrow \leftrightarrow \mathbf{lam}\ y.\ x}\,\mathsf{tr\_lam}^{w,',y,u'}}{\cdot \vdash \Lambda\Lambda 1\uparrow \leftrightarrow \mathbf{lam}\ x.\ \mathbf{lam}\ y.\ x}\,\mathsf{tr\_lam}^{w,x,u} \qquad \dfrac{\dfrac{\dfrac{}{w'' \Leftrightarrow v}\,u''}{\cdot; w'' \vdash 1 \leftrightarrow v}\,\mathsf{tr\_1}}{\cdot \vdash \Lambda 1 \leftrightarrow \mathbf{lam}\ v.\ v}\,\mathsf{tr\_lam}^{w'',v,u''}}{\cdot \vdash (\Lambda\Lambda 1\uparrow)\ (\Lambda 1) \leftrightarrow (\mathbf{lam}\ x.\ \mathbf{lam}\ y.\ x)\ (\mathbf{lam}\ v.\ v)}\,\mathsf{tr\_app}$$

The representation of the translation judgment relies on the standard technique for representing deductions of hypothetical judgments as functions.

```
trans  : env -> exp' -> exp -> type.  %name trans C
vtrans : val -> exp -> type.          %name vtrans U

tr_lam : trans K (lam' F) (lam E)
          <- {w:val} {x:exp}
                vtrans w x -> trans (K ; w) F (E x).
tr_app : trans K (app' F1 F2) (app E1 E2)
          <- trans K F1 E1
          <- trans K F2 E2.

tr_1   : trans (K ; W) 1 E <- vtrans W E.
tr_^   : trans (K ; W) (F ^) E <- trans K F E.

vtr_lam : vtrans (clo K (lam' F)) (lam E)
           <- trans K (lam' F) (lam E).
```

The judgment

$$\begin{array}{c} \dfrac{}{w \Leftrightarrow x}\,u \\ \vdots \\ K; w \vdash F \leftrightarrow e \end{array}$$

in the premiss of the tr_lam is parametric in the variables w and x and hypothetical in u. It is represented by a function which, when given a value W', an expression e', and a deduction $\mathcal{U}' :: W' \Leftrightarrow e'$ returns a deduction $\mathcal{D}' :: K; W' \vdash F \leftrightarrow [e'/x]e$. This property is crucial in the proof of compiler correctness.

The signature above can be executed as a non-deterministic program for translation between de Bruijn and ordinary expressions in both directions. For the compilation of expressions it is important to keep the clauses `tr_1` and `tr_^` in the given order so as to avoid unnecessary backtracking. This non-determinism arises, since the expression `E` in the rules `tr_1` and `tr_^` does not change in the recursive calls. For other possible implementations see Exercise 6.3. Here is an execution which yields the example deduction above.

```
?- C : trans empty F (app (lam [x] lam [y] x) (lam [v] v)).

F = app' (lam' (lam' (1 ^))) (lam' 1),
C =
   tr_app (tr_lam ([w'':val] [v:exp] [u'':vtrans w v] tr_1 u''))
      (tr_lam
          ([w:val] [x:exp] [u:vtrans w x]
              tr_lam ([w':val] [y:exp] [u':vtrans w' y]
                          tr_^ (tr_1 u)))).
;
no more solutions
```

It is not immediately obvious that every source expression e can in fact be compiled using this judgment. This is the subject of the following theorem.

Theorem 6.1 *For every closed expression e there exists a de Bruijn expression F such that $\cdot \vdash F \leftrightarrow e$.*

Proof: A direct attempt at an induction argument fails—a typical situation when proving properties of judgments which involve hypothetical reasoning. However, the theorem follows immediately from Lemma 6.2 below. □

Lemma 6.2 *Let $w_1, \ldots, w_n$ be parameters ranging over values and let K be the environment $\cdot; w_n; \ldots; w_1$. Furthermore, let $x_1, \ldots, x_n$ range over expression variables. For any expression e with free variables among $x_1, \ldots, x_n$ there exists a de Bruijn expression F and a deduction $\mathcal{C}$ of $K \vdash F \leftrightarrow e$ from hypotheses $u_1 :: w_1 \Leftrightarrow x_1, \ldots, u_n :: w_n \Leftrightarrow x_n$.*

Proof: By induction on the structure of e.

Case: $e = e_1\ e_2$. By induction hypothesis on e_1 and e_2, there exist F_1 and F_2 and deductions $\mathcal{C}_1 :: K \vdash F_1 \leftrightarrow e_1$ and $\mathcal{C}_2 :: K \vdash F_2 \leftrightarrow e_2$. Applying the rule tr_app to $\mathcal{C}_1$ and $\mathcal{C}_2$ yields the desired deduction $\mathcal{C} :: K \vdash F_1\ F_2 \leftrightarrow e_1\ e_2$.

Case: $e = \textbf{lam}\ x.\ e_1$. Here we apply the induction hypothesis to the expression e_1, environment $K; w$ for a new parameter w, and hypotheses $u_1 :: w_1 \Leftrightarrow x_1, \ldots, u_n :: w_n \Leftrightarrow x_n, u :: w \Leftrightarrow x$ to obtain an F_1 and a deduction

$$\begin{array}{c} \overline{w \Leftrightarrow x}\ u \\ \mathcal{C}_1 \\ K; w \vdash F_1 \leftrightarrow e_1 \end{array}$$

possibly also using hypotheses labelled $u_1, \ldots, u_n$. Note that e_1 is an expression with free variables among $x_1, \ldots, x_n, x$. Applying the rule $\mathsf{tr\_lam}$ discharges the hypothesis u and we obtain the desired deduction

$$\mathcal{C} = \begin{array}{c} \dfrac{}{w \Leftrightarrow x}\,u \\ \mathcal{C}_1 \\ \dfrac{K; w \vdash F_1 \leftrightarrow e_1}{K \vdash \Lambda F_1 \leftrightarrow \mathbf{lam}\; x.\, e_1}\,\mathsf{tr\_lam}^u \end{array}$$

Case: $e = x$. Then $x = x_i$ for some i between 1 and n and we let $F = 1 \underbrace{\uparrow \cdots \uparrow}_{i-1 \text{ times}}$ and

$$\mathcal{C} = \begin{array}{c} \dfrac{\dfrac{\dfrac{\dfrac{}{w_i \Leftrightarrow x_i}\,u_i}{\cdot; w_n; \ldots; w_i \vdash 1 \leftrightarrow x_i}\,\mathsf{tr\_1}}{\cdots}\,\mathsf{tr\_}{\uparrow}}{\cdot; w_n; \ldots; w_1 \vdash 1{\uparrow}\cdots{\uparrow} \leftrightarrow x_i}\,\mathsf{tr\_}{\uparrow} \end{array}$$

□

At present we do not know how to represent this proof in Elf because we cannot employ the usual technique for representing hypothetical judgments as functions. The difficulty is that the order of the hypotheses is important for returning the correct variable $1{\uparrow}\cdots{\uparrow}$, but hypothetical judgments are generally invariant under reordering of hypotheses. Hannan [Han91] has suggested a different, deterministic translation for which termination is relatively easy to show, but which complicates the proofs of the remaining properties of compiler correctness. Thus our formalization does not capture the desirable property that compilation always terminates. All the remaining parts, however, are implemented. The first property states that translation followed by evaluation leads to the same result as evaluation followed by translation. We generalize this for arbitrary environments K in order to allow a proof by induction. This property is depicted in the following diagram.

$$\begin{tikzcd}
e \arrow[r, "\mathcal{D} :: e \hookrightarrow v"] \arrow[d, "\mathcal{C} :: K \vdash F \leftrightarrow e"'] & v \\
K, F \arrow[r, dotted, "\mathcal{D}' :: K \vdash F \hookrightarrow W"'] & W \arrow[u, dotted, "\mathcal{U} :: W \Leftrightarrow v"']
\end{tikzcd}$$

The solid lines indicate deductions that are assumed, dotted lines represent the deductions whose existence we assert and prove below.

Lemma 6.3 *For any closed expressions e and v, environment K, de Bruijn expression F, deductions $\mathcal{D} :: e \hookrightarrow v$ and $\mathcal{C} :: K \vdash F \leftrightarrow e$, there exist a value W and deductions $\mathcal{D}' :: K \vdash F \hookrightarrow W$ and $\mathcal{U} :: W \Leftrightarrow v$.*

Proof: By induction on the structures of $\mathcal{D} :: e \hookrightarrow v$ and $\mathcal{C} :: K \vdash F \leftrightarrow e$. In this induction we assume the induction hypothesis on the premisses of $\mathcal{D}$ and for arbitrary $\mathcal{C}$ and on the premisses of $\mathcal{C}$, but for the same $\mathcal{D}$. This is sometimes called *lexicographic* induction on the pair consisting of $\mathcal{D}$ and $\mathcal{C}$. It should be intuitively clear that this form of induction is valid. We represent this proof as a judgment relating the four deductions involved in the diagram.

```
map_eval : eval E V -> trans K F E
              -> feval K F W -> vtrans W V -> type.
```

Case: $\mathcal{C}$ ends in an application of the tr_1 rule.

$$\mathcal{C} = \dfrac{\begin{array}{c}\mathcal{U}_1\\ W_1 \Leftrightarrow e\end{array}}{K_1; W_1 \vdash 1 \leftrightarrow e}\,\textsf{tr\_1}$$

$\mathcal{D} :: e \hookrightarrow v$ Assumption
$\mathcal{C}_1 :: K_1' \vdash \Lambda F_1' \leftrightarrow e$ and $W_1 = \{K_1', \Lambda F_1'\}$ By inversion on $\mathcal{U}_1$
$e = \textbf{lam}\ x.\ e_1$ By inversion on $\mathcal{C}_1$
$v = \textbf{lam}\ x.\ e_1 = e$ By inversion on $\mathcal{D}$

Then $W = W_1$, $\mathcal{U} = \mathcal{U}_1 :: W_1 \Leftrightarrow e$ and $\mathcal{D}' = \textsf{fev\_1} :: K_1; W_1 \vdash 1 \hookrightarrow W_1$ satisfy the requirements of the theorem. This case is captured in the clause

```
mp_1 : map_eval (ev_lam) (tr_1 (vtr_lam (tr_lam C2)))
                (fev_1) (vtr_lam (tr_lam C2)).
```

Case: $\mathcal{C}$ ends in an application of the tr_↑ rule.

$$\mathcal{C} = \dfrac{\begin{array}{c}\mathcal{C}_1\\ K_1 \vdash F_1 \leftrightarrow e\end{array}}{K_1; W_1' \vdash F_1\!\uparrow\, \leftrightarrow e}\,\textsf{tr\_}\!\uparrow$$

$\mathcal{D} :: e \hookrightarrow v$ Assumption
$\mathcal{D}_1' :: K_1 \vdash F_1 \hookrightarrow W_1$
and $\mathcal{U}_1 :: W_1 \Leftrightarrow v$ By ind. hyp. on $\mathcal{D}$ and $\mathcal{C}_1$

Now we let $W = W_1$, $\mathcal{U} = \mathcal{U}_1$, and obtain $\mathcal{D}' :: K_1; W_1' \vdash F_1{\uparrow} \hookrightarrow W_1$ by fev_↑ from $\mathcal{D}_1'$.

```
mp_^ : map_eval D (tr_^ C1) (fev_^ D1') U1
          <- map_eval D C1 D1' U1.
```

For the remaining cases we assume that the previous two cases do not apply. We refer to this assumption as *exclusion.*

Case: $\mathcal{D}$ ends in an application of the ev_lam rule.

$$\mathcal{D} = \frac{}{\textbf{lam}\ x.\ e_1 \hookrightarrow \textbf{lam}\ x.\ e_1}\,\textsf{ev\_lam}$$

$\mathcal{C} :: K \vdash F \leftrightarrow \textbf{lam}\ x.\ e_1$	By assumption
$F = \Lambda F_1$	By inversion and exclusion

Then we let $W = \{K, \Lambda F_1\}$, $\mathcal{D}' = \textsf{fev\_lam} :: K \vdash \Lambda F_1 \hookrightarrow \{K, \Lambda F_1\}$, and obtain $\mathcal{U} :: \{K, \Lambda F_1\} \Leftrightarrow \textbf{lam}\ x.\ e_1$ by vtr_lam from $\mathcal{C}$.

```
mp_lam : map_eval (ev_lam) (tr_lam C1)
                  (fev_lam) (vtr_lam (tr_lam C1)).
```

Case: $\mathcal{D}$ ends in an application of the ev_app rule.

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

This is the most interesting case, since it contains the essence of the argument how substitution can be replaced by binding variables to values in an environment.

$\mathcal{C} :: K \vdash F \leftrightarrow e_1\ e_2$	By assumption
$F = F_1\ F_2$,	
$\mathcal{C}_1 :: K \vdash F_1 \leftrightarrow e_1$, and	
$\mathcal{C}_2 :: K \vdash F_2 \leftrightarrow e_2$	By inversion and exclusion
$\mathcal{D}_2' :: K \vdash F_2 \hookrightarrow W_2$ and	
$\mathcal{U}_2 :: W_2 \Leftrightarrow v_2$	By ind. hyp. on $\mathcal{D}_2$ and $\mathcal{C}_2$
$\mathcal{D}_1' :: K \vdash F_1 \hookrightarrow W_1$ and	
$\mathcal{U}_1 :: W_1 \Leftrightarrow \textbf{lam}\ x.\ e_1'$	By ind. hyp. on $\mathcal{D}_1$ and $\mathcal{C}_1$
$W_1 = \{K_1, \Lambda F_1'\}$ and	
$\mathcal{C}_1' :: K_1 \vdash \Lambda F_1' \leftrightarrow \textbf{lam}\ x.\ e_1'$	By inversion on $\mathcal{U}_1$

Applying inversion again to $\mathcal{C}_1'$ shows that the premiss must be the deduction of a hypothetical judgment. That is,

$$\mathcal{C}_1' = \begin{array}{c} \overline{w \Leftrightarrow x}\ u \\ \mathcal{C}_3 \\ K_1; w \vdash F_1' \leftrightarrow e_1' \end{array}$$

where w is a new parameter ranging over values. This judgment is parametric in w and x and hypothetical in u. We can thus substitute W_2 for w, v_2 for x, and $\mathcal{U}_2$ for u to obtain a deduction

$$\mathcal{C}_3' :: K_1; W_2 \vdash F_1' \leftrightarrow [v_2/x]e_1'.$$

Now we apply the induction hypothesis to $\mathcal{D}_3$ and $\mathcal{C}_3'$ to obtain a W_3 and

$\mathcal{D}_3' :: K_1; W_2 \vdash F_1' \hookrightarrow W_3$ and
$\mathcal{U}_3 :: W_3 \Leftrightarrow v$.

We let $W = W_3$, $\mathcal{U} = \mathcal{U}_3$, and obtain $\mathcal{D}' :: K \vdash F_1\ F_2 \hookrightarrow W$ by fev_app from $\mathcal{D}_1'$, $\mathcal{D}_2'$, and $\mathcal{D}_3'$.

The implementation of this relatively complex reasoning employs again the magic of hypothetical judgments: the substitution we need to carry out to obtain $\mathcal{C}_3'$ from $\mathcal{C}_3$ is implemented as a function application.

```
mp_app : map_eval (ev_app D3 D2 D1) (tr_app C2 C1)
                  (fev_app D3' D2' D1') U3
             <- map_eval D1 C1 D1' (vtr_lam (tr_lam C3))
             <- map_eval D2 C2 D2' U2
             <- map_eval D3 (C3 W2 V2 U2) D3' U3.
```

This completes the proof once we have convinced ourselves that all possible cases have been considered. Note that whenever $\mathcal{C}$ ends in an application of the tr_1 or tr_↑ rules, then the first two cases apply. Otherwise one of the other two cases must apply, depending on the shape of $\mathcal{D}$. □

Theorem 6.1 and Lemma 6.3 together guarantee completeness of the translation.

Theorem 6.4 (Completeness) *For any closed expressions e and v and evaluation $\mathcal{D} :: e \hookrightarrow v$, there exist a de Bruijn expression F, a value W and deductions $\mathcal{C} :: \cdot \vdash F \leftrightarrow e$, $\mathcal{D}' :: \cdot \vdash F \hookrightarrow W$, and $\mathcal{U} :: W \Leftrightarrow v$.*

Proof: Lemma 6.3 shows that an evaluation $\mathcal{D}' :: K \vdash F \hookrightarrow W$ and a translation $W \Leftrightarrow v$ exist for any translation $\mathcal{C} :: K \vdash F \leftrightarrow e$. Theorem 6.1 shows that a particular F and translation $\mathcal{C} :: \cdot \vdash F \leftrightarrow e$ exist, thus proving the theorem. □

Completeness is insufficient to guarantee compiler correctness. For example, the translation of values $W \Leftrightarrow v$ could relate *any* expression v to any value W, which would make the statement of the previous theorem almost trivially true. We need to check a further property, namely that any value which could be produced by evaluating the compiled code, could also be produced by direct evaluation as specified by the natural semantics. This is shown in the diagram below.

$$\begin{tikzcd}
e \arrow[r, dotted, "\mathcal{D} :: e \hookrightarrow v"] \arrow[d, "\mathcal{C} :: K \vdash F \leftrightarrow e"'] & v \\
K, F \arrow[r, "\mathcal{D}' :: K \vdash F \hookrightarrow W"'] & W \arrow[u, "\mathcal{U} :: W \Leftrightarrow v"']
\end{tikzcd}$$

We call this property *soundness* of the compiler, since it prohibits the compiled code from producing incorrect values. We prove this from a lemma which asserts the existence of an expression v, evaluation $\mathcal{D}$ and translation $\mathcal{U}$, given the translation $\mathcal{C}$ and evaluation $\mathcal{D}'$. This yields the theorem by showing that the translation $\mathcal{U} :: W \Leftrightarrow v$, is uniquely determined from W.

Lemma 6.5 *For any closed expression e, de Bruijn expression F, environment K, value W, deductions $\mathcal{D}' :: K \vdash F \hookrightarrow W$ and $\mathcal{C} :: K \vdash F \leftrightarrow e$, there exist an expression v and deductions $\mathcal{D} :: e \hookrightarrow v$ and $\mathcal{U} :: W \Leftrightarrow v$.*

Proof: The proof proceeds by a straightforward induction over the structure of $\mathcal{D}' :: K \vdash F \hookrightarrow W$. It heavily employs inversion (as the proof of completeness, Lemma 6.3). Interestingly, this proof can be implemented by literally the same judgment. We leave it as exercise 6.6 to write out the informal proof—its representation from the proof of completeness is summarized below. Using is as a program in this instance means that we assume that second and third arguments are given and the first and last argument are logic variables whose instantiation terms are to be constructed.

```
map_eval : eval E V -> trans K F E
              -> feval K F W -> vtrans W V -> type.

mp_1 : map_eval (ev_lam) (tr_1 (vtr_lam (tr_lam C2)))
                (fev_1) (vtr_lam (tr_lam C2)).

mp_^ : map_eval D (tr_^ C1) (fev_^ D1') U1
          <- map_eval D C1 D1' U1.

mp_lam : map_eval (ev_lam) (tr_lam C1)
```

```
                  (fev_lam) (vtr_lam (tr_lam C1)).

mp_app : map_eval (ev_app D3 D2 D1) (tr_app C2 C1)
                  (fev_app D3' D2' D1') U3
               <- map_eval D1 C1 D1' (vtr_lam (tr_lam C3))
               <- map_eval D2 C2 D2' U2
               <- map_eval D3 (C3 W2 V2 U2) D3' U3.
```

□

Theorem 6.6 (Uniqueness of Translations) *For any value W if there exist a v and a translation $\mathcal{U} :: W \Leftrightarrow v$, then v and $\mathcal{U}$ are unique. Furthermore, for any environment K and de Bruijn expression F, if there exist an e and a translation $\mathcal{C} :: K \vdash F \leftrightarrow e$, then e and $\mathcal{C}$ are unique.*

Proof: By simultaneous induction on the structures of $\mathcal{U}$ and $\mathcal{C}$. In each case, either W or F uniquely determine the last inference. Since the translated expressions in the premisses are unique by induction hypothesis, so is the translated value in the conclusion. □

The proof requires no separate implementation in Elf in the same way that appeals to inversion remain implicit in the formulation of higher-level judgments. It is obtained by direct inspection of properties of the inference rules.

Theorem 6.7 (Soundness) *For any closed expressions e and v, de Bruijn expression F, environment K, value W, deductions $\mathcal{D}' :: K \vdash F \hookrightarrow W$, $\mathcal{C} :: K \vdash F \leftrightarrow e$, and $\mathcal{U} :: W \Leftrightarrow v$, there exists a deduction $\mathcal{D} :: e \hookrightarrow v$.*

Proof: From Lemma 6.5 we infer the existence of a v, $\mathcal{U}$, and $\mathcal{D}$, given $\mathcal{C}$ and $\mathcal{D}'$. Theorem 6.6 shows that v and $\mathcal{U}$ are unique, and thus the property must hold for all v and $\mathcal{U} :: W \Leftrightarrow v$, which is what we needed to show. □

6.2 Adding Data Values and Recursion

In the previous section we treated only a very restricted core language of Mini-ML. In this section we will extend the compiler to the full Mini-ML language as presented in Chapter 2. The main additions to the core language which affect the compiler are data values (such as natural numbers and pairs) and recursion. The language of de Bruijn expressions is extended by allowing constructors that parallel ordinary expressions. We maintain a similar syntax, but mark de Bruijn expression constructors with a prime (′).

$$
\begin{array}{llll}
\text{Expressions} & F & ::= & \mid \mathbf{z}' \mid \mathbf{s}'\ F \mid \mathbf{case}'\ F_1\ F_2\ F_3 & \textit{Natural Numbers} \\
 & & & \mid \langle F_1, F_2\rangle' \mid \mathbf{fst}'\ F \mid \mathbf{snd}'\ F & \textit{Pairs} \\
 & & & \mid \Lambda F \mid F_1\ F_2 & \textit{Functions} \\
 & & & \mid \mathbf{let}'\,\mathbf{val}\ F_1\ \mathbf{in}\ F_2 & \textit{Definitions} \\
 & & & \mid \mathbf{let}'\,\mathbf{name}\ F_1\ \mathbf{in}\ F_2 & \\
 & & & \mid \mathbf{fix}'\ F & \textit{Recursion} \\
 & & & \mid 1 \mid F{\uparrow} & \textit{Variables}
\end{array}
$$

Expressions of the form $F{\uparrow}$ are not necessarily variables (where F is a sequence of shifts applied to 1), but it may be intuitively helpful to think of them that way. In the representation we need only first-order constants, since this language has no constructs binding variables by name.

```
exp'   : type.  %name exp' F

1      : exp'.
^      : exp' -> exp'.  %postfix 20 ^
z'     : exp'.
s'     : exp' -> exp'.
case'  : exp' -> exp' -> exp' -> exp'.
pair'  : exp' -> exp' -> exp'.
fst'   : exp' -> exp'.
snd'   : exp' -> exp'.
lam'   : exp' -> exp'.
app'   : exp' -> exp' -> exp'.
letv'  : exp' -> exp' -> exp'.
letn'  : exp' -> exp' -> exp'.
fix'   : exp' -> exp'.
```

Next we need to extend the language of values. While data values can be added in a straightforward fashion, **let name** and recursion present some difficulties. Consider the evaluation rule for fixpoints.

$$\frac{[\mathbf{fix}\, x.\ e/x]e \hookrightarrow v}{\mathbf{fix}\, x.\ e \hookrightarrow v}\ \mathsf{ev\_fix}$$

We introduced the environment model of evaluation in order to eliminate the need for explicit substitution, where an environment is a list of values. In the case of the fixpoint construction we would need to bind the variable x to the expression $\mathbf{fix}\ x.\ e$ in the environment in order to avoid substutition, but $\mathbf{fix}\ x.\ e$ is not a value. The evaluation rules for de Bruijn expressions take advantage of the invariant that an environment contains only values. In particular, the rule

$$\frac{}{K; W \vdash 1 \hookrightarrow W}\,\mathsf{fev\_1}$$

requires that an environment contain only values. We will thus need to add a new environment constructor $K + F$ in order to allow unevaluated expressions in the environment. These considerations yield the following mutually recursive definitions of environments and values. We mark data values with a star (*) to distinguish them from expressions and de Bruijn expressions with the same name.

$$\begin{array}{rrcll}
\text{Environments} & K & ::= & \cdot \mid K; W \mid K + F & \\
\text{Values} & W & ::= & \mid \mathbf{z}^* \mid \mathbf{s}^*\, W & \textit{Natural Numbers} \\
 & & & \mid \langle W_1, W_2 \rangle^* & \textit{Pairs} \\
 & & & \mid \{K, F\} & \textit{Closures}
\end{array}$$

The Elf representation is direct.

```
env    : type.  %name env K
val    : type.  %name val W

empty  : env.
;      : env -> val -> env.   %infix left 10 ;
+      : env -> exp' -> env.  %infix left 10 +

z*     : val.
s*     : val -> val.

pair*  : val -> val -> val.

clo    : env -> exp' -> val.
```

In the extension of the evaluation rule to this completed language, we must exercise care in the treatment of the new environment constructor for unevaluated expression: when such an expression is looked up in the environment, it must be evaluated.

$$\frac{K \vdash F \hookrightarrow W}{K + F \vdash 1 \hookrightarrow W}\,\mathsf{fev\_1+} \qquad \frac{K \vdash F \hookrightarrow W}{K + F' \vdash F\!\uparrow \hookrightarrow W}\,\mathsf{fev\_\!\uparrow\!+}$$

The rules involving data values generally follow the patterns established in the natural semantics for ordinary expressions. The main departure from the earlier formulation is the separation of values from expressions. We show only four of the relevant rules.

$$\frac{}{K \vdash \mathbf{z}' \hookrightarrow \mathbf{z}^*}\,\mathsf{fev\_z} \qquad \frac{K \vdash F \hookrightarrow W}{K \vdash \mathbf{s}'\ F \hookrightarrow \mathbf{s}^*\ W}\,\mathsf{fev\_s}$$

$$\frac{K \vdash F_1 \hookrightarrow \mathbf{z}^* \qquad K \vdash F_2 \hookrightarrow W}{K \vdash \mathbf{case}'\ F_1\ F_2\ F_3 \hookrightarrow W}\,\mathsf{fev\_case\_z}$$

$$\frac{K \vdash F_1 \hookrightarrow \mathbf{s}^*\ W_1' \qquad K; W_1' \vdash F_3 \hookrightarrow W}{K \vdash \mathbf{case}'\ F_1\ F_2\ F_3 \hookrightarrow W}\,\mathsf{fev\_case\_s}$$

Evaluating a **let val**-expression also binds a variable to value by extending the environment.

$$\frac{K \vdash F_1 \hookrightarrow W_1 \qquad K; W_1 \vdash F_2 \hookrightarrow W}{K \vdash \mathbf{let\,val}'\ F_1\ \mathbf{in}\ F_2 \hookrightarrow W}\,\mathsf{fev\_letv}$$

Evaluating a **let name**-expression binds a variable to an expression and thus requires the new environment constructor.

$$\frac{K + F_1 \vdash F_2 \hookrightarrow W}{K \vdash \mathbf{let\,name}'\ F_1\ \mathbf{in}\ F_2 \hookrightarrow W}\,\mathsf{fev\_letn}$$

Fixpoint expressions are similar, except that the variable is bound to the **fix** expression itself.

$$\frac{K + \mathbf{fix}'\ F \vdash F \hookrightarrow W}{K \vdash \mathbf{fix}'\ F \hookrightarrow W}\,\mathsf{fev\_fix}$$

For example, **fix** $x.\ x$ (considered on page 16) is represented by $\mathbf{fix}'\ 1$. Intuitively, evaluation of this expression should not terminate. An attempt to construct an evaluation leads to the sequence

$$\cfrac{\cfrac{\cfrac{\vdots}{\cdot \vdash \mathbf{fix}'\ 1 \hookrightarrow W}\,\mathsf{fev\_fix}}{\cdot + \mathbf{fix}'\ 1 \vdash 1 \hookrightarrow W}\,\mathsf{fev\_1+}}{\cdot \vdash \mathbf{fix}'\ 1 \hookrightarrow W}\,\mathsf{fev\_fix}.$$

The implementation of these rules in Elf poses no particular difficulties. We show only the rules from above.

```
feval : env -> exp' -> val -> type.  %name feval D
```

```
% Variables
fev_1+ : feval (K + F) 1 W
            <- feval K F W.
fev_^+ : feval (K + F') (F ^) W
            <- feval K F W.

% Natural Numbers
fev_z : feval K z' z*.
fev_s : feval K (s' F) (s* W)
         <- feval K F W.
fev_case_z : feval K (case' F1 F2 F3) W
              <- feval K F1 z*
              <- feval K F2 W.
fev_case_s : feval K (case' F1 F2 F3) W
              <- feval K F1 (s* W1)
              <- feval (K ; W1) F3 W.

% Definitions
fev_letv : feval K (letv' F1 F2) W
            <- feval K F1 W1
            <- feval (K ; W1) F2 W.

fev_letn : feval K (letn' F1 F2) W
            <- feval (K + F1) F2 W.

% Recursion
fev_fix  : feval K (fix' F) W
             <- feval (K + (fix' F)) F W.
```

Next we need to extend the translation between expressions and de Bruijn expressions and values. We show a few interesting cases in the extended judgments $K \vdash F \leftrightarrow e$ and $W \Leftrightarrow v$. The case for **let val** is handled just like the case for **lam**, since we will always substitute a *value* for the variable bound by the **let** during execution.

$$\dfrac{}{K \vdash \mathbf{z}' \leftrightarrow \mathbf{z}}\,\textsf{tr\_z} \qquad\qquad \dfrac{K \vdash F \leftrightarrow e}{K \vdash \mathbf{s}'\ F \leftrightarrow \mathbf{s}\ e}\,\textsf{tr\_s}$$

$$\dfrac{K \vdash F_1 \leftrightarrow e_1 \qquad \begin{array}{c}\overline{w \Leftrightarrow x}^{\,u}\\ \vdots \\ K; w \vdash F_2 \leftrightarrow e_2\end{array}}{K \vdash \mathbf{let\,val}'\ F_1\ \mathbf{in}\ F_2 \leftrightarrow \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2}\,\textsf{tr\_letv}^{w,x,u}$$

where the right premiss of $\textsf{tr\_let}$ is parametric in w and x and hypothetical in u. In order to preserve the basic structure of the proofs of lemmas 6.3 and 6.5, we must treat the **let name** and **fix** constructs somewhat differently: we extend the environment with an *expression* parameter (not a value parameter) using the new environment constructor $+$.

$$\dfrac{K \vdash F_1 \leftrightarrow e_1 \qquad \begin{array}{c}\overline{K \vdash f \Leftrightarrow x}^{\,u}\\ \vdots \\ K + f \vdash F_2 \leftrightarrow e_2\end{array}}{K \vdash \mathbf{let\,name}'\ F_1\ \mathbf{in}\ F_2 \leftrightarrow \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2}\,\textsf{tr\_letn}^{f,x,u}$$

$$\dfrac{\begin{array}{c}\overline{K \vdash f \leftrightarrow x}^{\,u}\\ \vdots \\ K + f \vdash F \leftrightarrow e\end{array}}{K \vdash \mathbf{fix}'\ F \leftrightarrow \mathbf{fix}\ x.\ e}\,\textsf{tr\_fix}^{f,x,u}$$

$$\dfrac{K \vdash F \leftrightarrow e}{K + F \vdash 1 \leftrightarrow e}\,\textsf{tr\_1+} \qquad\qquad \dfrac{K \vdash F \leftrightarrow e}{K + F' \vdash F{\uparrow} \leftrightarrow e}\,\textsf{tr\_}{\uparrow}\textsf{+}$$

Finally, the value translation does not have to deal with fixpoint-expressions (they are not values). We only show the three new cases.

$$\dfrac{}{\mathbf{z}^* \Leftrightarrow \mathbf{z}}\,\textsf{vtr\_z} \qquad\qquad \dfrac{W \Leftrightarrow v}{\mathbf{s}^*\ W \Leftrightarrow \mathbf{s}\ v}\,\textsf{vtr\_s}$$

$$\dfrac{W_1 \Leftrightarrow v_1 \qquad W_2 \Leftrightarrow v_2}{\langle W_1, W_2\rangle^* \Leftrightarrow \langle v_1, v_2\rangle}\,\textsf{vtr\_pair}$$

Deductions of parametric and hypothetical judgments are represented by functions, as usual.

```
trans  : env -> exp' -> exp -> type.  %name trans C
vtrans : val -> exp -> type.          %name vtrans U

% Natural numbers
tr_z     : trans K z' z.
tr_s     : trans K (s' F) (s E)
            <- trans K F E.

% Definitions
tr_letv: trans K (letv' F1 F2) (letv E1 E2)
            <- trans K F1 E1
            <- ({w:val} {x:exp}
                  vtrans w x -> trans (K ; w) F2 (E2 x)).

tr_letn: trans K (letn' F1 F2) (letn E1 E2)
            <- trans K F1 E1
            <- ({f:exp'} {x:exp}
                  trans K f x -> trans (K + f) F2 (E2 x)).

% Recursion
tr_fix : trans K (fix' F) (fix E)
            <- ({f:exp'} {x:exp}
                   trans K f x -> trans (K + f) F (E x)).

% Variables
tr_1+ : trans (K + F) 1 E <- trans K F E.
tr_^+ : trans (K + F') (F ^) E <- trans K F E.

% Natural number values
vtr_z : vtrans z* z.
vtr_s : vtrans (s* W) (s V)
         <- vtrans W V.

% Pair values
vtr_pair : vtrans (pair* W1 W2) (pair V1 V2)
            <- vtrans W1 V1
            <- vtrans W2 V2.
```

In order to extend the proof of compiler correctness in Section 6.1 we need to extend various lemmas.

Theorem 6.8 *For every closed expression e there exists a de Bruijn expression F such that $\cdot \vdash F \leftrightarrow e$.*

Proof: We generalize analogously to Lemma 6.2 and prove the modified lemma by induction on the structure of e (see Exercise 6.7). □

Lemma 6.9 *If $W \Leftrightarrow e$ is derivable, then e Value is derivable.*

Proof: By a straightforward induction on the structure of $\mathcal{U} :: W \Leftrightarrow e$. □

Lemma 6.10 *If e Value and $e \hookrightarrow v$ are derivable then $e = v$.*

Proof: By a straightforward induction on the structure of $\mathcal{P} :: e$ *Value.* □

The Elf implementations of the proofs of Lemmas 6.9 and 6.10 is straightforward and can be found in the on-line material that accompanies these notes. The type families are

```
vtrans_val : vtrans W E -> value E -> type.
val_eval   : value E -> eval E E -> type.
```

The next lemma is the main lemma is the proof of completeness, that is, every value which can obtained by direct evaluation can also be obtained by compilation, evaluation of the compiled code, and translation of the returned value to the original language.

Lemma 6.11 *For any closed expressions e and v, environment K, de Bruijn expression F, deduction $\mathcal{D} :: e \hookrightarrow v$ and $\mathcal{C} :: K \vdash F \leftrightarrow e$, there exist a value W and deductions $\mathcal{D}' :: K \vdash F \hookrightarrow W$ and $\mathcal{U} :: W \Leftrightarrow v$.*

Proof: By induction on the structure of $\mathcal{D} :: e \hookrightarrow v$ and $\mathcal{C} :: K \vdash F \leftrightarrow e$. In this induction, as in the proof of Lemma 6.3, we assume the induction hypothesis on the premisses of $\mathcal{D}$ and for arbitrary $\mathcal{C}$, and on the premisses of $\mathcal{C}$ if $\mathcal{D}$ remains fixed. The implementation is an extension of the previous higher-level judgment,

```
map_eval : eval E V -> trans K F E
              -> feval K F W -> vtrans W V -> type.
```

We show only some of the typical cases—the others are straightforward and left to the reader or remain unchanged from the proof of Lemma 6.3

Case: $\mathcal{C}$ ends in an application of the tr_1 rule.

$$\mathcal{C} = \dfrac{\begin{array}{c}\mathcal{U}_1\\ W_1 \Leftrightarrow e\end{array}}{K_1; W_1 \vdash 1 \leftrightarrow e}\,\mathsf{tr\_1}$$

This case changes from the previous proof, since there we applied simple inversion (there was only one possible kind of value) to conclude that $e = v$. Here we need two lemmas from above.

$\mathcal{D} :: e \hookrightarrow v$	Assumption
$\mathcal{P} :: e \ \mathit{Value}$	By Lemma 6.9 from $\mathcal{U}_1$
$e = v$	By Lemma 6.10 from $\mathcal{P}$

Hence we can let W be W_1, $\mathcal{U}$ be $\mathcal{U}_1$, and $\mathcal{D}'$ be $\mathsf{fev\_1} :: K_1; W_1 \vdash 1 \hookrightarrow W_1$. The implementation explicitly appeals to the implementations of the lemmas.

```
mp_1 : map_eval D (tr_1 U1) (fev_1) U1
          <- vtrans_val U1 P
          <- val_eval P D.
```

Case: $\mathcal{C}$ ends in an application of the $\mathsf{tr\_}\!\uparrow$ rule. This case proceeds as before.

```
mp_^ : map_eval D (tr_^ C1) (fev_^ D1') U1
          <- map_eval D C1 D1' U1.
```

Case: $\mathcal{C}$ ends in an application of the $\mathsf{tr\_1+}$ rule.

$$\mathcal{C} = \frac{\begin{array}{c}\mathcal{C}_1 \\ K_1 \vdash F_1 \leftrightarrow e\end{array}}{K_1 + F_1 \vdash 1 \leftrightarrow e}\,\mathsf{tr\_1+}$$

$\mathcal{D} :: e \hookrightarrow v$	Assumption
$\mathcal{D}_1' :: K_1 \vdash F_1 \hookrightarrow W_1$ and	
$\mathcal{U}_1 :: W_1 \Leftrightarrow v$	By ind. hyp. on $\mathcal{D}$ and $\mathcal{C}_1$
$\mathcal{D}' :: K_1 + F_1 \vdash 1 \hookrightarrow W$	By $\mathsf{fev\_1+}$ from $\mathcal{D}_1'$

and we can let $W = W_1$ and $\mathcal{U} = \mathcal{U}_1$.

```
mp_1+ : map_eval D (tr_1+ C1) (fev_1+ D1') U1
           <- map_eval D C1 D1' U1.
```

Case: $\mathcal{C}$ ends in an application of the $\mathsf{tr\_}\!\uparrow\!+$ rule. This case is just like the $\mathsf{tr\_}\!\uparrow$ case.

```
mp_^+ : map_eval D (tr_^+ C1) (fev_^+ D1') U1
           <- map_eval D C1 D1' U1.
```

For the remaining cases we may assume that none of the four cases above apply. We only show the case for fixpoints.

Case: $\mathcal{D}$ ends in an application of the $\mathsf{ev\_fix}$ rule.

$$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ [\mathbf{fix}\, x.\, e_1/x]e_1 \hookrightarrow v\end{array}}{\mathbf{fix}\, x.\, e_1 \hookrightarrow v}\,\mathsf{ev\_fix}$$

$\mathcal{C} :: K \vdash F \leftrightarrow \mathbf{fix}\, x.\, e_1$ By assumption

By inversion and exclusion (of the previous cases), $\mathcal{C}$ must end in an application of the $\mathsf{tr\_fix}$ rule and thus $F = \mathbf{fix}'\ F_1$ for some F_1 and there is a deduction $\mathcal{C}_1$, parametric in f and x and hypothetical in u, of the form

$$\begin{array}{c}\overline{K \vdash f \leftrightarrow x}\,u\\ \mathcal{C}_1\\ K + f \vdash F_1 \leftrightarrow e_1\end{array}$$

In this deduction we can substitute $\mathbf{fix}'\ F_1$ for f and $\mathbf{fix}\ x.\ e_1$ for x, and replace the resulting hypothesis $u :: K \vdash \mathbf{fix}'\ F_1 \leftrightarrow \mathbf{fix}\ x.\ e_1$ by $\mathcal{C}$! This way we obtain a deduction

$$\mathcal{C}_1' :: K + \mathbf{fix}'\ F_1 \vdash F_1 \leftrightarrow [\mathbf{fix}\, x.\, e_1/x]e_1.$$

Now we can apply the induction hypothesis to $\mathcal{D}_1$ and $\mathcal{C}_1'$ which yields a W_1 and deductions

$\mathcal{D}_1' :: K + \mathbf{fix}'\ F_1 \vdash F_1 \hookrightarrow W_1$ and
$\mathcal{U}_1 :: W_1 \Leftrightarrow v$ By ind. hyp. on $\mathcal{D}_1$ and $\mathcal{C}_1'$

Applying $\mathsf{fev\_fix}$ to $\mathcal{D}_1'$ results in a deduction

$$\mathcal{D}' :: K \vdash \mathbf{fix}'\ F_1 \hookrightarrow W_1$$

and we let W be W_1 and $\mathcal{U}$ be $\mathcal{U}_1$. In Elf, the substitutions into the hypothetical deduction are implemented by applications of the representing function `C1`.

```
mp_fix : map_eval (ev_fix D1) (tr_fix C1)
                  (fev_fix D1') U1
              <- map_eval D1 (C1 (fix' F1) (fix E1) (tr_fix C1))
                      D1' U1.
```

□

This lemma and the totality of the translation relation in its expression argument (Theorem 6.8) together guarantee completeness of the translation.

Theorem 6.12 (Completeness) *For any closed expressions e and v and evaluation $\mathcal{D} :: e \hookrightarrow v$, there exist a de Bruijn expression F, a value W and deductions $\mathcal{C} :: \cdot \vdash F \leftrightarrow e$, $\mathcal{D}' :: \cdot \vdash F \hookrightarrow W$, and $\mathcal{U} :: W \Leftrightarrow v$.*

Proof: As in the proof of Theorem 6.4, but using Lemma 6.11 and Theorem 6.8 instead of Lemma 6.3 and Theorem 6.1. □

Lemma 6.13 *For any closed expression e, de Bruijn expression F, environment K, value W, deduction $\mathcal{D}' :: K \vdash F \hookrightarrow W$ and $\mathcal{C} :: K \vdash F \leftrightarrow e$, there exist an expression v and deductions $\mathcal{D} :: e \hookrightarrow v$ and $\mathcal{U} :: W \Leftrightarrow v$.*

Proof: By induction on the structure of $\mathcal{D}' :: K \vdash F \hookrightarrow W$. The family `map_eval` which implements the main lemma in the soundness proof, also implements the proof of this lemma without any change. □

Theorem 6.14 (Uniqueness of Translations) *For any value W if there exist a v and a translation $\mathcal{U} :: W \Leftrightarrow v$, then v and $\mathcal{U}$ are unique. Furthermore, for any environment K and de Bruijn expression F, if there exist an e and a translation $\mathcal{C} :: K \vdash F \leftrightarrow e$, then e and $\mathcal{C}$ are unique.*

Proof: As before, by a simultaneous induction on the structures of $\mathcal{U}$ and $\mathcal{C}$. □

Theorem 6.15 (Soundness) *For any closed expressions e and v, de Bruijn expression F, environment K, value W, deductions $\mathcal{D}' :: K \vdash F \hookrightarrow W$, $\mathcal{C} :: K \vdash F \leftrightarrow e$, and $\mathcal{U} :: W \Leftrightarrow v$, there exists a deduction $\mathcal{D} :: e \hookrightarrow v$.*

Proof: From Lemma 6.13 we infer the existence of a v, $\mathcal{U}$, and $\mathcal{D}$, given $\mathcal{C}$ and $\mathcal{D}'$. Theorem 6.14 shows that v and $\mathcal{U}$ are unique, and thus the property must hold for all v and $\mathcal{U} :: W \Leftrightarrow v$, which is what we needed to show. □

6.3 Computations as Transition Sequences

So far, we have modelled evaluation as the construction of a deduction of the evaluation judgment. This is true for evaluation based on substitution in Section 2.3 and for evaluation based on environments in Section 6.1. In an abstract machine (and, of course, in an actual machine) a more natural model for computation is a sequence of states. In this section we will develop the CLS machine, an abstract machine similar in scope to the SECD machine [Lan64]. The CLS machine still interprets expressions, so the step from environment based evaluation to this abstract machine does not involve any compilation. Instead, we flatten evaluation trees to sequences of states that describe the computation. This flattening involves some rather arbitrary decisions about which subcomputations should be performed first. We linearize the evaluation deductions beginning with the deduction of the leftmost premiss.

Throughout the remainder of this chapter, we will drop the prime ($'$) from the expression constructors. This should not lead to any confusion, since we no longer need to refer to the original expressions. Now consider the rule for evaluating pairs as a simple example where an evaluation tree has two branches.

$$\frac{K \vdash F_1 \hookrightarrow W_1 \qquad K \vdash F_2 \hookrightarrow W_2}{K \vdash \langle F_1, F_2 \rangle \hookrightarrow \langle W_1, W_2 \rangle^*}\,\textsf{fev\_pair}$$

An abstract machine would presumably start in a state where it is given the environment K and the expression $\langle F_1, F_2\rangle$. The final state of the machine should somehow indicate the final value $\langle W_1, W_2\rangle^*$. The computation naturally decomposes into three phases: the first phase computes the value of F_1 in environment K, the second phase computes the value of F_2 in environment K, and the third phase combines the two values to form a pair. These phases mean that we have to preserve the environment K and also the expression F_2 while we are computing the value of F_1. Similarly, we have to save the value W_1 while computing the value of F_2. A natural data structure for saving components of a state is a stack. The considerations above suggest three stacks: a stack KS of environments, a stack of expressions to be evaluated, and a stack S of values. However, we also need to remember that, after the evaluation of F_2 we need to combine W_1 and W_2 into a pair. Thus, instead of a stack of expression to be evaluated, we maintain a *program* which consists of expressions and special instructions (such as: *make a pair* written as *mkpair*).

We will need more instructions later, but so far we have:

$$\begin{array}{rrcl}
\text{Instructions} & I & ::= & F \mid \mathit{mkpair} \mid \ldots \\
\text{Programs} & P & ::= & \mathit{done} \mid I \,\&\, P \\
\text{Environment Stacks} & \mathit{KS} & ::= & \cdot \mid \mathit{KS}; K \\
\text{Value Stacks} & S & ::= & \cdot \mid S; W \\
\text{State} & \mathit{St} & ::= & \langle \mathit{KS}, P, S \rangle
\end{array}$$

Note that value stacks are simply environments, so we will not formally distinguish them from environments. The instructions of a program a sequenced with &; the program *done* indicates that there are no further instructions, that is, computation should stop.

A state consists of an environment stack KS, a program P and a value stack S, written as $\langle KS, P, S\rangle$. We have single-step and multi-step transition judgments:

$$\begin{array}{ll}
\mathit{St} \Longrightarrow \mathit{St}' & \mathit{St} \text{ goes to } \mathit{St}' \text{ in one computation step} \\
\mathit{St} \overset{*}{\Longrightarrow} \mathit{St}' & \mathit{St} \text{ goes to } \mathit{St}' \text{ in zero or more steps}
\end{array}$$

We define the transition judgment so that

$$\langle(\cdot; K), F \mathbin{\&} \mathit{done}, \cdot\rangle \overset{*}{\Longrightarrow} \langle\cdot, \mathit{done}, (\cdot; W)\rangle$$

corresponds to the evaluation of F in environment K to value W. The free variables of F are therefore bound in the innermost environment, and the value resulting from evaluation is deposited on the top of the value stack, which starts out empty. Global evaluation is expressed in the judgment

$$K \vdash F \overset{*}{\Longrightarrow\!\!\!\!\!\Rightarrow} W \quad F \text{ computes to } W \text{ in environment } K$$

which is defined by the single inference rule

$$\frac{\langle(\cdot; K), F \mathbin{\&} \mathit{done}, \cdot\rangle \overset{*}{\Longrightarrow} \langle\cdot, \mathit{done}, (\cdot; W)\rangle}{K \vdash F \overset{*}{\Longrightarrow\!\!\!\!\!\Rightarrow} W}\,\textsf{run}.$$

We prove in Theorem 6.19 that $K \vdash F \overset{*}{\Longrightarrow\!\!\!\!\!\Rightarrow} W$ iff $K \vdash F \hookrightarrow W$. We cannot prove this statement directly by induction (in either direction), since during a computation situations arise where the environment stack consists of more than a single environment, the remaining program is not *done*, *etc.* In one direction we generalize it to

$$\langle(KS; K), F \mathbin{\&} P, S\rangle \overset{*}{\Longrightarrow} \langle KS, P, (S; W)\rangle$$

if $K \vdash F \hookrightarrow W$. This is the subject of Lemma 6.16. A slightly modified form of the converse is given in Lemma 6.18.

The transition rules and the remaining instructions can be developed systematically from the intuition provided above. First, we reconsider the evaluation of pairing. The first rule decomposes the pair expression and saves the environment K on the environment stack.

$$\textsf{c\_pair} :: \langle(KS; K), \langle F_1, F_2\rangle \mathbin{\&} P, S\rangle \Longrightarrow \langle(KS; K; K), F_1 \mathbin{\&} F_2 \mathbin{\&} \mathit{mkpair} \mathbin{\&} P, S\rangle$$

Here $\textsf{c\_pair}$ labels the rule and can be thought of as the deduction of the given transition judgment. The evaluation of F_1, if it terminates, leads to a state

$$\langle(KS; K), F_2 \mathbin{\&} \mathit{mkpair} \mathbin{\&} P, (S; W_1)\rangle,$$

and the further evaluation of F_2 then leads to a state

$$\langle KS, \mathit{mkpair} \mathbin{\&} P, (S; W_1; W_2)\rangle.$$

Thus, the *mkpair* instruction should cause the machine to create a pair from the first two elements on the value stack and deposit the result again on the value stack. That is, we need as another rule:

$$\textsf{c\_mkpair} :: \langle KS, \mathit{mkpair} \mathbin{\&} P, (S; W_1; W_2)\rangle \Longrightarrow \langle KS, P, (S; \langle W_1, W_2\rangle^*)\rangle.$$

We consider one other construct in detail: application. To evaluate an application $F_1\ F_2$ we first evaluate F_1 and then we evaluate F_2. If the value of F_1 is a closure, we have to bind its variable to the value of F_2 and continue evaluation in an extended environment. The instruction that unwraps the closure and extends the environment is called *apply*.

$$\begin{array}{rcl} \mathsf{c\_app} & :: & \langle (KS;K), F_1\ F_2 \,\&\, P, S\rangle \Longrightarrow \langle (KS;K;K), F_1 \,\&\, F_2 \,\&\, \mathit{apply} \,\&\, P, S\rangle \\ \mathsf{c\_apply} & :: & \langle KS, \mathit{apply} \,\&\, P, (S;\{K', \Lambda F_1'\};W_2)\rangle \Longrightarrow \langle (KS;(K';W_2)), F_1' \,\&\, P, S\rangle \end{array}$$

The rules for applying zero and successor are straightforward, but they necessitate a new operator *add1* to increment the first value on the stack.

$$\begin{array}{rcl} \mathsf{c\_z} & :: & \langle (KS;K), \mathbf{z} \,\&\, P, S\rangle \Longrightarrow \langle KS, P, (S;\mathbf{z}^*)\rangle \\ \mathsf{c\_s} & :: & \langle (KS;K), \mathbf{s}\ F \,\&\, P, S\rangle \Longrightarrow \langle (KS;K), F \,\&\, \mathit{add1} \,\&\, P, S\rangle \\ \mathsf{c\_add1} & :: & \langle KS, \mathit{add1} \,\&\, P, (S;W)\rangle \Longrightarrow \langle KS, P, (S;\mathbf{s}^*\ W)\rangle \end{array}$$

For expressions of the form **case** $F_1\ F_2\ F_3$, we need to evaluate F_1 and then evaluate either F_2 or F_3, depending on the value of F_1. This requires a new instruction, *branch*, which either goes to the next instructions or skips the next instruction. In the latter case it also needs to bind a new variable in the environment to the predecessor of the value of F_1.

$$\begin{array}{rl} \mathsf{c\_case} :: & \langle (KS;K), \mathbf{case}\ F_1\ F_2\ F_3 \,\&\, P, S\rangle \\ & \Longrightarrow \langle (KS;K;K), F_1 \,\&\, \mathit{branch} \,\&\, F_2 \,\&\, F_3 \,\&\, P, S\rangle \\ \mathsf{c\_branch\_z} :: & \langle (KS;K), \mathit{branch} \,\&\, F_2 \,\&\, F_3 \,\&\, P, (S;\mathbf{z}^*)\rangle \Longrightarrow \langle (KS;K), F_2 \,\&\, P, S\rangle \\ \mathsf{c\_branch\_s} :: & \langle (KS;K), \mathit{branch} \,\&\, F_2 \,\&\, F_3 \,\&\, P, (S;\mathbf{s}^*\ W)\rangle \\ & \Longrightarrow \langle (KS;(K;W)), F_3 \,\&\, P, S\rangle \end{array}$$

Rules for **fst** and **snd** require new instructions to extract the first or second element of the value on the top of the stack.

$$\begin{array}{rcl} \mathsf{c\_fst} & :: & \langle (KS;K), \mathbf{fst}\ F \,\&\, P, S\rangle \Longrightarrow \langle (KS;K), F \,\&\, \mathit{getfst} \,\&\, P, S\rangle \\ \mathsf{c\_getfst} & :: & \langle KS, \mathit{getfst} \,\&\, P, (S;\langle W_1, W_2\rangle^*)\rangle \Longrightarrow \langle KS, P, (S;W_1)\rangle \\ \mathsf{c\_snd} & :: & \langle (KS;K), \mathbf{snd}\ F \,\&\, P, S\rangle \Longrightarrow \langle (KS;K), F \,\&\, \mathit{getsnd} \,\&\, P, S\rangle \\ \mathsf{c\_getsnd} & :: & \langle KS, \mathit{getsnd} \,\&\, P, (S;\langle W_1, W_2\rangle^*)\rangle \Longrightarrow \langle KS, P, (S;W_2)\rangle \end{array}$$

In order to handle **let val** we introduce another new instruction *bind*, even though it is not strictly necessary and could be simulated with other instructions (see Exercise 6.10).

$$\begin{array}{rl} \mathsf{c\_let} :: & \langle (KS;K), \mathbf{let}\ F_1\ \mathbf{in}\ F_2 \,\&\, P, S\rangle \Longrightarrow \langle (KS;K;K), F_1 \,\&\, \mathit{bind} \,\&\, F_2 \,\&\, P, S\rangle \\ \mathsf{c\_bind} :: & \langle (KS;K), \mathit{bind} \,\&\, F_2 \,\&\, P, (S;W_1)\rangle \Longrightarrow \langle (KS;(K;W_1)), F_2 \,\&\, P, S\rangle \end{array}$$

We leave the rules for recursion to Exercise 6.11. The rules for variables and abstractions thus complete the specification of the single-step transition relation.

$$\begin{array}{rcl} \mathsf{c\_1} & :: & \langle (KS;(K;W)), 1 \,\&\, P, S\rangle \Longrightarrow \langle KS, P, (S;W)\rangle \\ \mathsf{c\_\uparrow} & :: & \langle (KS;(K;W')), F\!\uparrow \,\&\, P, S\rangle \Longrightarrow \langle (KS;K), F \,\&\, P, S\rangle \\ \mathsf{c\_lam} & :: & \langle (KS;K), \Lambda F \,\&\, P, S\rangle \Longrightarrow \langle KS, P, (S;\{K, \Lambda F\})\rangle \end{array}$$

The set of instructions extracted from these rules is

$$\text{Instructions} \quad I \quad ::= \quad F \mid \mathit{add1} \mid \mathit{branch} \mid \mathit{mkpair} \mid \mathit{getfst} \mid \mathit{getsnd} \mid \mathit{apply} \mid \mathit{bind}.$$

We view each of the transition rules for the single-step transition judgment as an axiom. Note that there are no other inference rules for this judgment. A partial computation is defined as a multi-step transition. This is easily defined via the following two inference rules.

$$\frac{}{St \overset{*}{\Longrightarrow} St}\,\mathsf{id} \qquad \frac{St \Longrightarrow St' \qquad St' \overset{*}{\Longrightarrow} St''}{St \overset{*}{\Longrightarrow} St''}\,\mathsf{step}$$

This definition guarantees that the end state of one transition matches the beginning state of the remaining transition sequence. Without the aid of dependent types we would have to define a computation as a list states and ensure externally that the end state of each transition matches the beginning state of the next. This use of dependent types to express complex constraints is one of the reasons why simple lists do not arise very frequently in Elf programming.

Deductions of the judgment $St \overset{*}{\Longrightarrow} St'$ have a very simple form: They all consist of a sequence of single steps terminated by an application of the id rule. We will follow standard practice and use a linear notation for sequences of steps:

$$St_1 \Longrightarrow St_2 \Longrightarrow \cdots \Longrightarrow St_n$$

Similarly, we will mix multi-step and single-step transitions in sequences, with the obvious meaning. We write $\mathcal{C}_1 \circ \mathcal{C}_2$ for the result of appending computations $\mathcal{C}_1$ and $\mathcal{C}_2$. This only makes sense if the final state of $\mathcal{C}_1$ is the same as the start state of $\mathcal{C}_2$. The $\circ$ operator is associative (see Exercise 6.12).

Recall that a complete computation was defined as a sequence of transitions from an initial state to a final state. The latter is characterized by the program *done*, and empty environment stack, and a value stack containing exactly one value, namely the result of the computation.

$$\frac{\langle (\cdot; K), F \,\&\, \mathit{done}, \cdot \rangle \overset{*}{\Longrightarrow} \langle \cdot, \mathit{done}, (\cdot; W) \rangle}{K \vdash F \overset{*}{\Longrightarrow\!\!\!\!>} W}\,\mathsf{run}$$

The representation of the abstract machine and the computation judgments present no particular difficulties. We begin with the syntax.

```
instruction : type.  %name instruction I
program     : type.  %name program P
envstack    : type.  %name envstack Ks
state       : type.  %name state St
```

```
ev     : exp' -> instruction.
add1   : instruction.
branch : instruction.
mkpair : instruction.
getfst : instruction.
getsnd : instruction.
apply  : instruction.
bind   : instruction.

done : program.
&    : instruction -> program -> program.
%infix right 10 &

emptys : envstack.
;;     : envstack -> env -> envstack.
%infix left 10 ;;

st : envstack -> program -> env -> state.
```

The computation rules are also a straightforward transcription of the rules above. The judgment $St \overset{*}{\Longrightarrow} St'$ is represented by a type `St => St'` where `=>` is a type family indexed by two states and written in infix notation. We show only three example rules.

```
=> : state -> state -> type.  %infix none 10 =>
                              %name => R

c_z     : st (Ks ;; K) (ev z' & P) S => st Ks P (S ; z*).

c_app   : st (Ks ;; K) (ev (app' F1 F2) & P) S
             => st (Ks ;; K ;; K) (ev F1 & ev F2 & apply & P) S.

c_apply : st Ks (apply & P) (S ; clo K' (lam' F1') ; W2)
             => st (Ks ;; (K' ; W2)) (ev F1' & P) S.
```

The multi-step transition is defined by the transcription of its two inference rules. We write `~` in infix notation rather than step since it leads to a concise and readable notation for sequences of computation steps.

```
=>*  : state -> state -> type.  %infix none 10 =>*
                                %name =>* C

id   : St =>* St.
```

```
~    : St => St'
    -> St' =>* St''
    -> St =>* St''.

%infix right 10 ~
```

Complete computations appeal directly to the multi-step computation judgment. We write `ceval K F W` for $K \vdash F \overset{*}{\Longrightarrow} W$.

```
ceval : env -> exp' -> val -> type.

run : st (emptys ;; K) (ev F & done) (empty)
      =>* st (emptys) (done) (empty ; W)
       -> ceval K F W.
```

While this representation is declaratively adequate it has a serious operational defect when used for evaluation, that is, when K and F are given and W is to be determined. The declaration for step (written as ~) solves the innermost subgoal first, that is we reduce the goal of finding a computation $\mathcal{C}'' :: St \overset{*}{\Longrightarrow} St''$ to finding a state St' and computation of $\mathcal{C}' :: St' \overset{*}{\Longrightarrow} St''$ and only then a single transition $R :: St \Longrightarrow St'$. This leads to non-termination, since the interpreter is trying to work its way backwards through the space of possible computation sequences. Instead, we can get linear, backtracking-free behavior if we first find the single step $R :: St \Longrightarrow St'$ and then the remainder of the computation $\mathcal{C}' :: St' \overset{*}{\Longrightarrow} St''$. Since there is exactly one rule for any instruction I and id will apply only when the program P is *done*, finding a computation now becomes a deterministic process. Executable versions of the last two judgments are given below. They differ from the one above only in the order of the recursive calls and it is a simple matter to relate the two versions formally.

```
>=>*  : state -> state -> type.  %infix none 10 >=>*

id<   : St >=>* St.
<=<   : St >=>* St''
         <- St => St'
         <- St' >=>* St''.
%infix left 10 <=<

>ceval : env -> exp' -> val -> type.

>run   : >ceval K F W
          <- st (emptys ;; K) (ev F & done) (empty)
             >=>* st (emptys) (done) (empty ; W).
```

This example clearly illustrates that Elf should be thought of a uniform language in which one can express specifications (such as the computations above) and implementations (the operational versions below), but that many specifications will not be executable. This is generally the situation in logic programming languages.

In the informal development it is clear (and not usually separately formulated as a lemma) that computation sequences can be concatenated if the final state of the first computation matches the initial state of the second computation. In the formalization of the proofs below, we will need to explicitly implement a type family that appends computation sequences. It cannot be formulated as a function, since such a function would have to be recursive and is thus not definable in LF.

```
append : st Ks P S =>* st Ks' P' S'
          -> st Ks' P' S' =>* st Ks'' P'' S''
          -> st Ks P S =>* st Ks'' P'' S''
          -> type.
```

The defining clauses are left as Exercise 6.12.

We now return to the task of proving the correctness of the abstract machine. The first lemma states the fundamental motivating property for this model of computation.

Lemma 6.16 *Let K be an environment, F an expression, and W a value such that $K \vdash F \hookrightarrow W$. Then, for any environment stack KS, program P and stack S,*

$$\langle (KS; K), F \,\&\, P, S\rangle \overset{*}{\Longrightarrow} \langle KS, P, (S; W)\rangle$$

Proof: By induction on the structure of $\mathcal{D} :: K \vdash F \hookrightarrow W$. We will construct a deduction of $\mathcal{C} :: \langle (KS; K), F \,\&\, P, S\rangle \overset{*}{\Longrightarrow} \langle KS, P, (S; W)\rangle$. The proof is straightforward and we show only two typical cases. The implementation in Elf takes the form of a higher-level judgment `subcomp` that relates evaluations to computation sequences.

```
subcomp : feval K F W
           -> st (Ks ;; K) (ev F & P) S =>* st Ks P (S ; W)
           -> type.
```

Case: $\mathcal{D}$ ends in an application of the rule fev_z.

$$\mathcal{D} = \quad \frac{}{K \vdash \mathbf{z} \hookrightarrow \mathbf{z}^*}\,\textsf{fev\_z}.$$

Then the single-step transition

$$\langle (KS; K), \mathbf{z} \,\&\, P, S\rangle \overset{*}{\Longrightarrow} \langle KS, P, (S; \mathbf{z}^*)\rangle$$

satisfies the requirements of the lemma. The clause corresponding to this case:

```
sc_z : subcomp (fev_z) (c_z ~ id).
```

Case: $\mathcal{D}$ ends in an application of the fev_app rule.

$$\mathcal{D} = \dfrac{\begin{array}{ccc}\mathcal{D}_1 & \mathcal{D}_2 & \mathcal{D}_3 \\ K \vdash F_1 \hookrightarrow \{K', \Lambda F_1'\} & K \vdash F_2 \hookrightarrow W_2 & K'; W_2 \vdash F_1' \hookrightarrow W\end{array}}{K \vdash F_1\ F_2 \hookrightarrow W}\,\textsf{fev\_app}$$

Then

$$\begin{array}{lr}
\langle (KS; K), F_1\ F_2 \,\&\, P, S\rangle & \\
\Longrightarrow \langle (KS; K; K), F_1 \,\&\, F_2 \,\&\, \mathit{apply} \,\&\, P, S\rangle & \text{By rule c\_app} \\
\overset{*}{\Longrightarrow} \langle (KS; K), F_2 \,\&\, \mathit{apply} \,\&\, P, (S; \{K', \Lambda F_1'\})\rangle & \text{By ind. hyp. on } \mathcal{D}_1 \\
\overset{*}{\Longrightarrow} \langle KS, \mathit{apply} \,\&\, P, (S; \{K', \Lambda F_1'\}; W_2)\rangle & \text{By ind. hyp. on } \mathcal{D}_2 \\
\Longrightarrow \langle (KS; (K'; W_2)), F_1' \,\&\, P, S\rangle & \text{By rule c\_apply} \\
\overset{*}{\Longrightarrow} \langle KS, P, (S; W)\rangle & \text{By ind. hyp. on } \mathcal{D}_3.
\end{array}$$

The implementation of this case requires the `append` family defined above. Note how an appeal to the induction hypothesis is represented as a recursive call.

```
sc_app : subcomp (fev_app D3 D2 D1) C
          <- subcomp D1 C1
          <- subcomp D2 C2
          <- subcomp D3 C3
          <- append (c_app ~ C1) C2 C'
          <- append C' (c_apply ~ C3) C.
```

□

The first direction of Theorem 6.19 is a special case of this lemma. The other direction is more intricate. The basic problem is to extract a tree-structured evaluation from a linear computation. We must then show that this extraction will always succeed for complete computations. Note that it is obviously not possible to extract evaluations from arbitrary incomplete sequences of transitions of the abstract machine.

In order to write computation sequences more concisely, we introduce some notation. Let $R :: St \Longrightarrow St'$ and $\mathcal{C} :: St' \overset{*}{\Longrightarrow} St''$. Then we write

$$R \sim \mathcal{C} :: St \Longrightarrow St''$$

for the computation which begins with R and then proceeds with $\mathcal{C}$. Such a computation exists by the step inference rule. This corresponds directly to the notation in the Elf implementation.

For the proof of the central lemma of this section, we will need a new form of induction often referred to as *complete induction*. During a proof by complete induction we assume the induction hypothesis not only for the immediate premisses of the last inference rule, but for all proper subderivations. Intuitively, this is justified, since all proper subderivations are "smaller" than the given derivation. For a more formal discussion of the complete induction principle for derivations see Section 6.4. The judgment $\mathcal{C} < \mathcal{C}'$ ($\mathcal{C}$ is a *proper subcomputation of* $\mathcal{C}'$) is defined by the following inference rules.

$$\frac{}{\mathcal{C} < R \sim \mathcal{C}}\,\textsf{sub\_imm} \qquad\qquad \frac{\mathcal{C} < \mathcal{C}'}{\mathcal{C} < R \sim \mathcal{C}'}\,\textsf{sub\_med}$$

It is easy to see that the proper subcomputation relation is transitive.

Lemma 6.17 *If* $\mathcal{C}_1 < \mathcal{C}_2$ *and* $\mathcal{C}_2 < \mathcal{C}_3$ *then* $\mathcal{C}_1 < \mathcal{C}_3$.

Proof: By a simple induction (see Exercise 6.12). □

The implementation of this ordering and the proof of transitivity are immediate.

```
 < : (st KS1 P1 S1) =>* (st KS P S)
      -> (st KS2 P2 S2) =>* (st KS P S)
      -> type.
%infix none 8 <

sub_imm : C < R ~ C.

sub_med : C < C'
           -> C < R ~ C'.

trans : C1 < C2 -> C2 < C3 -> C1 < C3 -> type.
```

The representation of the proof of transitivity is left to Exercise 6.12.

We are now prepared for the lemma that a complete computation with an appropriate initial state can be translated into an evaluation followed by another complete computation.

Lemma 6.18 *If*

$$\mathcal{C} :: \langle (KS; K), F \,\&\, P, S\rangle \overset{*}{\Longrightarrow} \langle \cdot, \mathit{done}, (\cdot; W')\rangle$$

there exists a value W*, an evaluation*

$$\mathcal{D} :: K \vdash F \hookrightarrow W,$$

and a computation

$$\mathcal{C}' :: \langle KS, P, (S; W)\rangle \overset{*}{\Longrightarrow} \langle \cdot, \mathit{done}, (\cdot; W')\rangle$$

such that $\mathcal{C}' < \mathcal{C}$.

Proof: By complete induction on the $\mathcal{C}$. We only show a few cases; the others similar. We use the abbreviation $\mathit{final} = \langle \cdot, \mathit{done}, (\cdot; W')\rangle$. The representing type family `spl` is indexed by four deductions: $\mathcal{C}$, $\mathcal{C}'$, $\mathcal{D}$, and the derivation which shows that $\mathcal{C}' < \mathcal{C}$. In the declaration we need to use the dependent kind constructor in order to name $\mathcal{C}$ and $\mathcal{C}'$ so they can be related explicitly.

```
spl : {C : (st (KS ;; K) (ev F & P) S)
           =>* (st (emptys) (done) (empty ; W'))}
      feval K F W ->
      {C' : (st KS P (S ; W))
            =>* (st (emptys) (done) (empty ; W'))}
      C' < C -> type.
```

Case: $\mathcal{C}$ begins with $\mathsf{c\_z}$, that is, $\mathcal{C} = \mathsf{c\_z} \sim \mathcal{C}_1$. Then $W = \mathbf{z}^*$,

$$\mathcal{D} = \frac{}{K \vdash \mathbf{z} \hookrightarrow \mathbf{z}^*}\,\mathsf{fev\_z},$$

and $\mathcal{C}' = \mathcal{C}_1$. Furthermore, $\mathcal{C}' = \mathcal{C}_1 < \mathsf{c\_z} \sim \mathcal{C}_1$ by rule $\mathsf{sub\_imm}$. The representation in Elf:

```
spl_z : spl (c_z ~ C1) (fev_z) C1 (sub_imm).
```

Case: $\mathcal{C}$ begins with $\mathsf{c\_app}$. Then $\mathcal{C} = \mathsf{c\_app} \sim \mathcal{C}_1$ where

$$\mathcal{C}_1 :: \langle (KS; K; K), F_1 \,\&\, F_2 \,\&\, \mathit{apply} \,\&\, P, S\rangle \overset{*}{\Longrightarrow} \mathit{final}.$$

By induction hypothesis on $\mathcal{C}_1$ there exists a W_1, an evaluation

$$\mathcal{D}_1 :: K \vdash F_1 \hookrightarrow W_1$$

and a computation

$$\mathcal{C}_2 :: \langle (KS; K), F_2 \,\&\, \mathit{apply} \,\&\, P, (S; W_1)\rangle \overset{*}{\Longrightarrow} \mathit{final}$$

such that $\mathcal{C}_2 < \mathcal{C}_1$. We can thus apply the induction hypothesis to $\mathcal{C}_2$ to obtain a W_2, an evaluation

$$\mathcal{D}_2 :: K \vdash F_2 \hookrightarrow W_2$$

and a computation

$$\mathcal{C}_3 :: \langle KS, \mathit{apply} \,\&\, P, (S; W_1; W_2)\rangle \stackrel{*}{\Longrightarrow} \mathit{final}$$

such that $\mathcal{C}_3 < \mathcal{C}_2$. By inversion, $\mathcal{C}_3 = \mathsf{c\_apply} \sim \mathcal{C}_3'$ and $W_1 = \{K', \Lambda F_1'\}$ where

$$\mathcal{C}_3' :: \langle (KS; (K'; W_2)), F_1' \,\&\, P, S\rangle \stackrel{*}{\Longrightarrow} \mathit{final}.$$

Then $\mathcal{C}_3' < \mathcal{C}_3$ and by induction hypothesis on $\mathcal{C}_3'$ there is a value W_3, an evaluation

$$\mathcal{D}_3 :: K'; W_2 \vdash F_1' \hookrightarrow W_3$$

and a compuation

$$\mathcal{C}_4 :: \langle KS, P, (S; W_3)\rangle \stackrel{*}{\Longrightarrow} \mathit{final}.$$

Now we let $W = W_3$ and we construct $\mathcal{D} :: K \vdash F_1\ F_2 \hookrightarrow W_3$ by an application of the rule $\mathsf{fev\_app}$ to the premisses $\mathcal{D}_1$, $\mathcal{D}_2$, and $\mathcal{D}_3$. Furthermore we let $\mathcal{C}' = \mathcal{C}_4$ and conclude by some elementary reasoning concerning the subcomputation relation that $\mathcal{C}' < \mathcal{C}$.

The representation of this subcase of this case requires three explicit appeals to the transitivity of the subcomputation ordering. In order to make this at all intelligible, we use the name `C2<C1` (one identifier) for the derivation that $\mathcal{C}_2 < \mathcal{C}_1$ and similarly for other such derivations.

```
spl_app : spl (c_app ~ C1)
              (fev_app D3 D2 D1) C4
              (sub_med C4<C1)
               <- spl C1 D1 C2 C2<C1
               <- spl C2 D2 (c_apply ~ C3') C3<C2
               <- spl C3' D3 C4 C4<C3'
               <- trans C3<C2 C2<C1 C3<C1
               <- trans (sub_imm) C3<C1 C3'<C1
               <- trans C4<C3' C3'<C1 C4<C1.
```

□

Now we have all the essential lemmas to prove the main theorem.

Theorem 6.19 *$K \vdash F \hookrightarrow W$ is derivable iff $K \vdash F \stackrel{*}{\Longrightarrow\!\!\!\!\!\!\Rightarrow} W$ is derivable.*

Proof: By definition, $K \vdash F \stackrel{*}{\Longrightarrow\!\!\!\!\!\!\Rightarrow} W$ iff there is a computation

$$\mathcal{C} :: \langle (\cdot; K), F \,\&\, \mathit{done}, \cdot\rangle \stackrel{*}{\Longrightarrow} \langle \cdot, \mathit{done}, (\cdot; W)\rangle.$$

One direction follows immediately from Lemma 6.16: if $K \vdash F \hookrightarrow W$ then

$$\langle (KS; K), F \,\&\, P, S\rangle \overset{*}{\Longrightarrow} \langle KS, P, (S; W)\rangle$$

for *any* KS, P, and S and in particular for $KS = \cdot$, $P = \mathit{done}$ and $S = \cdot$. The implementation of this direction in Elf:

```
cev_complete : feval K F W -> ceval K F W -> type.

cevc : cev_complete D (run C) <- subcomp D C.
```

For the other direction, assume there is a deduction $\mathcal{C}$ of the form shown above. By Lemma 6.18 we know that there exist a W', an evaluation

$$\mathcal{D}' :: K \vdash F \hookrightarrow W'$$

and a computation

$$\mathcal{C}' :: \langle \cdot, \mathit{done}, (\cdot; W')\rangle \overset{*}{\Longrightarrow} \langle \cdot, \mathit{done}, (\cdot; W)\rangle$$

such that $\mathcal{C}' < \mathcal{C}$. Since there is no transition rule for the program *done*, $\mathcal{C}'$ must be id and $W = W'$. Thus $\mathcal{D} = \mathcal{D}'$ fulfills the requirements of the theorem. This is implemented as follows.

```
cls_sound : ceval K F W -> feval K F W -> type.

clss : cls_sound (run C) D <- spl C D (id) Id<C.
```

$\square$

6.4 Complete Induction over Computations

Here we briefly justify the principle of complete induction used in the proof of Lemma 6.18. We repeat the definition of proper subcomputations and also define a general subcomputation judgment which will be useful in the proof.

$\mathcal{C} < \mathcal{C}'$	$\mathcal{C}$ is a proper subcomputation of $\mathcal{C}'$, and
$\mathcal{C} \leq \mathcal{C}'$	$\mathcal{C}$ is a subcomputation of $\mathcal{C}'$.

These judgments are defined via the following inference rules.

$$\frac{}{\mathcal{C} < R \sim \mathcal{C}}\,\mathsf{sub\_imm} \qquad \frac{\mathcal{C} < \mathcal{C}'}{\mathcal{C} < R \sim \mathcal{C}'}\,\mathsf{sub\_med}$$

$$\frac{\mathcal{C} < \mathcal{C}'}{\mathcal{C} \leq \mathcal{C}'}\,\mathsf{leq\_sub} \qquad \frac{}{\mathcal{C} \leq \mathcal{C}}\,\mathsf{leq\_eq}$$

We only need one simple lemma regarding the subcomputation judgment.

Lemma 6.20 *If $\mathcal{C} \leq \mathcal{C}'$ is derivable, then $\mathcal{C} < R \sim \mathcal{C}'$ is derivable.*

Proof: By analyzing the two possibilities for the deduction of the premiss and constructing an immediate deduction for the conclusion in each case. □

We call a property P of computations *complete* if it satisfies:

> For every $\mathcal{C}$, the assumption that P holds for all $\mathcal{C}' < \mathcal{C}$ implies that P holds for $\mathcal{C}$.

Theorem 6.21 (Principle of Complete Induction over Computations) *If a property P of computations is complete, then P holds for all computations.*

Proof: We assume that P is complete and then prove by ordinary structural induction that for every $\mathcal{C}$ and for every $\mathcal{C}' \leq \mathcal{C}$, P holds of $\mathcal{C}$.

Case: $\mathcal{C} = id$. By inversion, there is no $\mathcal{C}'$ such that $\mathcal{C}' < id$. Thus P holds for all $\mathcal{C}' < id$. Since P is complete, this implies that P holds for id.

Case: $\mathcal{C} = R \sim \mathcal{C}_1$. The induction hypothesis states that for every $\mathcal{C}_1' \leq \mathcal{C}_1$, P holds of $\mathcal{C}_1'$. We have to show that for every $\mathcal{C}_2 \leq R \sim \mathcal{C}_1$, property P holds of $\mathcal{C}_2$. By inversion, there are two subcases, depending on the evidence for $\mathcal{C}_2 \leq R \sim \mathcal{C}_1$.

Subcase: $\mathcal{C}_2 = R \sim \mathcal{C}_1$. The induction hypothesis and Lemma 6.20 yield that for every $\mathcal{C}_1' < R \sim \mathcal{C}_1$, P holds of $\mathcal{C}_1$. Since P is complete, P must thus hold for $R \sim \mathcal{C}_1 = \mathcal{C}_2$.

Subcase: $\mathcal{C}_2 < R \sim \mathcal{C}_1$. Then by inversion either $\mathcal{C}_1 = \mathcal{C}_2$ or $\mathcal{C}_1 < \mathcal{C}_2$. In either case $\mathcal{C}_2 \leq \mathcal{C}_1$ by one inference. Now we can apply the induction hypothesis to conclude that P holds of $\mathcal{C}_2$.

□

6.5 Exercises

Exercise 6.1 If we replace the rule ev_app in the natural semantics of Mini-ML (see Section 2.3) by

$$\dfrac{e_1 \hookrightarrow \mathbf{lam}\ x.\ e_1' \qquad e_2 \hookrightarrow v_2 \qquad \begin{array}{c}\overline{x \hookrightarrow v_2}^{\,u} \\ \vdots \\ e_1' \hookrightarrow v\end{array}}{e_1\ e_2 \hookrightarrow v}\ \textsf{ev\_app}'^{x,u}$$

in order to avoid explicitly substituting v_2 for x, something goes wrong. What is it? Can you suggest a way to fix the problem which still employs hypothetical judgments?

(Note: We assume that the third premiss of the modified rule is parametric in x and hypothetical in u which is discharged as indicated. This implies that we assume that x is not already free in any other hypothesis and that all labels for hypotheses are distinct—so this is *not* the problem you are asked to detect.)

Exercise 6.2 Define the judgment W *RealVal* which restricts closures W to Λ-abstractions. Prove that $\cdot \vdash F \hookrightarrow W$ then W *RealVal* and represent this proof in Elf.

Exercise 6.3 In this exercise we try to eliminate some of the non-determinism in compilation.

1. Define a judgment F *std* which should be derivable if the de Bruijn expression F is in the standard form in which the $\uparrow$ operator is not applied to applications or abstractions.

2. Rewrite the translation from ordinary expressions e such that only standard forms can be related to any expression e.

3. Prove the property in item 2.

4. Implement the judgments in items 1, 2, and the proof in item 3.

Exercise 6.4 Restrict yourself to the fragment of the language with variables, abstraction, and application, that is,

$$F \quad ::= \quad 1 \mid F\uparrow \mid \Lambda F \mid F_1\ F_2$$

1. Define a judgment F *Closed* that is derivable iff the de Bruijn expression F is closed, that is, has no free variables at the *object level.*

2. Define a judgment for conversion of de Bruijn expressions F to standard form (as in Exercise 6.3, item 1) in a way that preserves meaning (as given by its interpretation as an ordinary expression e).

3. Prove that, under appropriate assumptions, this conversion results in a de Bruijn expression in standard form equivalent to the original expression.

4. Implement the judgments and correctness proofs in Elf.

Exercise 6.5 Restrict yourself to the same fragment as in Exercise 6.4 and define the operation of substitution as a judgment *subst* F_1 F_2 F. It should be a consequence of your definition that if ΛF_1 represents $\mathbf{lam}x.\ e_1$, F_2 represents e_2, and *subst* F_1 F_2 F is derivable then F should represent $[e_2/x]e_1$. Furthermore, such an F should always exist if F_1 and F_2 are as indicated. With appropriate assumptions about free variables or indices (see Exercise 6.4) prove these properties, thereby establishing the correctness of your implementation of substitution.

Exercise 6.6 Write out the informal proof of Theorem 6.7.

Exercise 6.7 Prove Theorem 6.8 by appropriately generalizing Lemma 6.2.

Exercise 6.8 Standard ML [MTH90] and many other formulations do not contain a **let name** construct. Disregarding problems of polymorphic typing for the moment, it is quite simple to simulate **let name** with **let val** operationally using so-called *thunks*. The idea is that we can prohibit the evaluation of an arbitrary expression by wrapping it in a vacuous **lam**-abstraction. Evaluation can be forced by applying the function to some irrelevant value (we write **z**, most presentations use a unit element). That is, instead of

$$l \quad = \quad \textbf{let name}\ x = e_1\ \textbf{in}\ e_2$$

we write

$$l' \quad = \quad \textbf{let val}\ x' = \textbf{lam}\ y.\ e_1\ \textbf{in}\ [x'\,\mathbf{z}/x]e_2$$

where y is a new variable not free in e_1.

1. Show a counterexample to the conjecture "*If l is closed, $l \hookrightarrow v$, and $l' \hookrightarrow v'$ then $v = v'$ (modulo renaming of bound variables)*".

2. Show a counterexample to the conjecture "$\triangleright\, l : \tau$ *iff* $\triangleright\, l' : \tau$".

3. Define an appropriate congruence $e \cong e'$ such that $l \cong l'$ and if $e \cong e'$, $e \hookrightarrow v$ and $e' \hookrightarrow v'$ then $v \cong v'$.

4. Prove the properties in item 3.

5. Prove that if the values v and v' are natural numbers, then $v \cong v'$ iff $v = v'$.

We need a property such as the last one to make sure that the congruence we define does not identify all expressions. It is a special case of a so-called *observational equivalence* (see **??**).

Exercise 6.9 The rules for evaluation in Section 6.2 have the drawback that looking up a variable in an environment and evaluation are mutually recursive, since the environment contains unevaluated expressions. Such expressions may be added to the environment during evaluation of a **let name** or **fix** construct. In the definition of Standard ML [MTH90] this problem is avoided by disallowing **let name** (see Exercise 6.8) and by syntactically restricting occurrences of the **fix** construct.

When translated into our setting, this restriction states that all occurrences of fixpoint expressions must be of the form **fix** x. **lam** y. e. Then we can dispense with the environment constructor $+$ and instead introduce a constructor $*$ that builds a recursive environment. More precisely, we have

$$\text{Environments} \quad K \quad ::= \quad \cdot \mid K; W \mid K * F$$

The evaluation rules fev_1+, fev_↑+, and fev_fix on page 159 are replaced by

$$\frac{}{K \vdash \mathbf{fix}'\ F \hookrightarrow \{K * F, F\}}\ \mathsf{fev\_fix}*$$

$$\frac{}{K * F \vdash 1 \hookrightarrow \{K * F, F\}}\ \mathsf{fev\_1}*$$

$$\frac{K \vdash F \hookrightarrow W}{K * F' \vdash F\uparrow \hookrightarrow W}\ \mathsf{fev\_}{\uparrow}*$$

1. Implement this modified evaluation judgment in Elf.

2. Prove that under the restriction that all occurrences of $\mathbf{fix}'$ in de Bruijn expressions have the form $\mathbf{fix}'\ \Lambda F$ for some F, the two sets of rules define an equivalent operational semantics. Take care to give a precise definition of the notion of equivalence you are considering and explain why it is appropriate.

3. Represent the equivalence proof in Elf.

4. Exhibit a counterexample which shows that some restriction on fixpoint expressions (as, for example, the one given above) is necessary in order to preserve equivalence.

5. Under the syntactic restriction from above we can also formulate a semantics which requires no new constructor for environments by forming closures over fixpoint expressions. Then we need to add another rule for application of an expression which evaluates to a closure over a fixpoint expression. Write out the rules and prove its equivalence to either the system above or the original evaluation judgment for de Bruijn expressions (under the appropriate restriction).

Exercise 6.10 Show how the effect of the *bind* instruction can be simulated in the CLS machine using the other instructions. Sketch the correctness proof for this simulation.

Exercise 6.11 Complete the presentation of the CLS machine by adding recursion. In particular

1. Complete the computation rules on page 169.
2. Add appropriate cases to the proofs of Lemmas 6.16, and 6.18.

Exercise 6.12 Prove the following carefully.

1. The concatenation operation "$\circ$" on computations is associative.
2. The subcomputation relation "$<$" is transitive (Lemma 6.17).

Show the implementation of your proofs as type families in Elf.

Exercise 6.13 The machine instructions from Section 6.3 can simply quote expressions in de Bruijn form and consider them as instructions. As a next step in the (abstract) compilation process, we can convert the expressions to lower-level code which simulates the effect of instructions on the environment and value stacks in smaller steps.

1. Design an appropriate language of operations.
2. Specify and implement a compiler from expressions to code.
3. Prove the correctness of this step of compilation.
4. Implement your correctness proof in Elf.