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 2.12 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 **??**.

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 η and an expression $\mathbf{lam}\ y.\ e$. η 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

$$\mathbf{lam}\, x.\ \mathbf{lam}\, y.\ x\ x$$

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.

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

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 $\{\eta; \Lambda F'\}$ (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 f.

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

% Environments and values

env    : type.  %name env N.
val    : type.  %name val W 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 η 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.

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

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.

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

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 η, written as $\eta \vdash F \hookrightarrow W$. The variable 1 refers to the first value in the environment (counting from right to left); its evaluation just returns that value.

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

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

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

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

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

In order to evaluate $F_1\ F_2$ in an environment η we evaluate both F_1 and F_2 in that environment, yielding the closure $\{\eta'; \Lambda F_1'\}$ and value W_2, respectively. We then add W_2 to the environment η', 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.

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

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

```
feval : env -> exp' -> val -> type.  %name feval D'.
%mode feval +N +F -W.

% Variables
fev_1 : feval (N , W) 1 W.
fev_^ : feval (N , W') (F ^) W
```

```
            <- feval N F W.

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

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

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

where the rule $\mathsf{tr\_lam}$ 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.

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

As remarked earlier this translation can be non-deterministic if η 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.

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

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.
% can be used in different directions
%mode trans +N +F -E.
%mode vtrans +W -V.

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

% Variables
tr_1  : trans (N , W) 1 E
         <- vtrans W E.
tr_^  : trans (N , W) (F ^) E
         <- trans N F E.

% Values
vtr_lam : vtrans (clo N (lam' F)) (lam E)
           <- trans N (lam' F) (lam E).
```

The judgment

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

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 $\mathcal{D}' :: \eta, W' \vdash F \leftrightarrow [e'/x]e$. 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 [z] z)).

F = app' (lam' (lam' (1 ^))) (lam' 1).
C =
   tr_app (tr_lam ([w:val] [x:exp] [u3:vtrans w x] tr_1 u3))
      (tr_lam
          ([w:val] [x:exp] [u1:vtrans w x]
              tr_lam ([w1:val] [x1:exp] [u2:vtrans w1 x1]
                          tr_^ (tr_1 u1)))).
```

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 η 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 $\eta \vdash F \leftrightarrow e$ 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 $\mathcal{C}_1 :: \eta \vdash F_1 \leftrightarrow e_1$ and $\mathcal{C}_2 :: \eta \vdash F_2 \leftrightarrow e_2$. Applying the rule tr_app to $\mathcal{C}_1$ and $\mathcal{C}_2$ yields the desired deduction $\mathcal{C} :: \eta \vdash F_1\ F_2 \leftrightarrow e_1\ e_2$.

Case: $e = \mathbf{lam}\ x.\ e_1$. Here we apply the induction hypothesis to the expression e_1, environment η, 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

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

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 tr_lam discharges the hypothesis u and we obtain the desired deduction

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

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

$$\mathcal{C} = \quad \dfrac{\dfrac{\dfrac{\overline{w_i \Leftrightarrow x_i}\ u_i}{\cdot, w_n, \ldots, w_i \vdash 1 \leftrightarrow x_i}\ \mathsf{tr\_1}}{\cdots}\ \mathsf{tr\_}\!\uparrow}{\cdot, w_n, \ldots, w_1 \vdash 1\!\uparrow \cdots \uparrow\ \leftrightarrow x_i}\ \mathsf{tr\_}\!\uparrow$$

□

This proof cannot be represented directly 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 η in order to allow a proof by induction.

This property is depicted in the following diagram.

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

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 η, de Bruijn expression F, deductions $\mathcal{D} :: e \hookrightarrow v$ and $\mathcal{C} :: \eta \vdash F \leftrightarrow e$, there exist a value W and deductions $\mathcal{D}' :: \eta \vdash F \hookrightarrow W$ and $\mathcal{U} :: W \Leftrightarrow v$.*

Proof: By induction on the structures of $\mathcal{D} :: e \hookrightarrow v$ and $\mathcal{C} :: \eta \vdash F \leftrightarrow e$. 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 N F E
              -> feval N F W -> vtrans W V -> type.
%mode map_eval +D +C -D' -U.
```

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

$$
\mathcal{C} = \frac{\begin{array}{c}\mathcal{U}_1 \\ W_1 \Leftrightarrow e\end{array}}{\eta_1, W_1 \vdash 1 \leftrightarrow e}\,\mathsf{tr\_1}
$$

$\mathcal{D} :: e \hookrightarrow v$	Assumption
$\mathcal{C}_1 :: \eta_1' \vdash \Lambda F_1' \leftrightarrow e$ and $W_1 = \{\eta_1'; \Lambda F_1'\}$	By inversion on $\mathcal{U}_1$
$e = \mathbf{lam}\, x.\, e_1$	By inversion on $\mathcal{C}_1$
$v = \mathbf{lam}\, x.\, e_1 = e$	By inversion on $\mathcal{D}$

Then $W = W_1$, $\mathcal{U} = \mathcal{U}_1 :: W_1 \Leftrightarrow e$ and $\mathcal{D}' = \mathsf{fev\_1} :: \eta_1, W_1 \vdash 1 \hookrightarrow W_1$ 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 $\mathsf{tr\_}\!\uparrow$ rule.

$$\mathcal{C} = \dfrac{\begin{array}{c}\mathcal{C}_1\\ \eta_1 \vdash F_1 \leftrightarrow e\end{array}}{\eta_1, W_1' \vdash F_1\!\uparrow \leftrightarrow e}\,\mathsf{tr\_}\!\uparrow$$

$\mathcal{D} :: e \hookrightarrow v$ — Assumption
$\mathcal{D}_1' :: \eta_1 \vdash F_1 \hookrightarrow W_1$
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 $\mathcal{D}' :: \eta_1, W_1' \vdash F_1\!\uparrow \hookrightarrow W_1$ by $\mathsf{fev\_}\!\uparrow$ 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 $\mathsf{ev\_lam}$ rule.

$$\mathcal{D} = \dfrac{}{\mathbf{lam}\ x.\ e_1 \hookrightarrow \mathbf{lam}\ x.\ e_1}\,\mathsf{ev\_lam}$$

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

Then we let $W = \{\eta; \Lambda F_1\}$, $\mathcal{D}' = \mathsf{fev\_lam} :: \eta \vdash \Lambda F_1 \hookrightarrow \{\eta; \Lambda F_1\}$, and obtain $\mathcal{U} :: \{\eta; \Lambda F_1\} \Leftrightarrow \mathbf{lam}\ x.\ e_1$ by $\mathsf{vtr\_lam}$ 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 $\mathsf{ev\_app}$ rule.

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

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.

$\mathcal{C} :: \eta \vdash F \leftrightarrow e_1\ e_2$ By assumption
$F = F_1\ F_2$,
$\mathcal{C}_1 :: \eta \vdash F_1 \leftrightarrow e_1$, and
$\mathcal{C}_2 :: \eta \vdash F_2 \leftrightarrow e_2$ By inversion and exclusion
$\mathcal{D}'_2 :: \eta \vdash F_2 \hookrightarrow W_2$ and
$\mathcal{U}_2 :: W_2 \Leftrightarrow v_2$ By ind. hyp. on $\mathcal{D}_2$ and $\mathcal{C}_2$
$\mathcal{D}'_1 :: \eta \vdash F_1 \hookrightarrow W_1$ and
$\mathcal{U}_1 :: W_1 \Leftrightarrow \mathbf{lam}\ x.\ e'_1$ By ind. hyp. on $\mathcal{D}_1$ and $\mathcal{C}_1$
$W_1 = \{\eta_1; \Lambda F'_1\}$ and
$\mathcal{C}'_1 :: \eta_1 \vdash \Lambda F'_1 \leftrightarrow \mathbf{lam}\ x.\ e'_1$ 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,

$$\mathcal{C}'_1 = \begin{array}{c} \overline{\quad w \Leftrightarrow x \quad}\, u \\ \mathcal{C}_3 \\ \eta_1, w \vdash F'_1 \leftrightarrow e'_1 \end{array}$$

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

$$\mathcal{C}'_3 :: \eta_1, W_2 \vdash F'_1 \leftrightarrow [v_2/x]e'_1.$$

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

$\mathcal{D}'_3 :: \eta_1, W_2 \vdash F'_1 \hookrightarrow W_3$ and
$\mathcal{U}_3 :: W_3 \Leftrightarrow v$.

We let $W = W_3$, $\mathcal{U} = \mathcal{U}_3$, and obtain $\mathcal{D}' :: \eta \vdash F_1\ F_2 \hookrightarrow W$ 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 $\mathcal{D}' :: \eta \vdash F \hookrightarrow W$ and a translation $W \Leftrightarrow v$ exist for any translation $\mathcal{C} :: \eta \vdash F \leftrightarrow e$. 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.

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

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 η, value W, deductions $\mathcal{D}' :: \eta \vdash F \hookrightarrow W$ and $\mathcal{C} :: \eta \vdash F \leftrightarrow e$, 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 $\mathcal{D}' :: \eta \vdash F \hookrightarrow W$. 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 N F E
              -> feval N F W -> vtrans W V -> type.
%mode map_eval' -D +C +D' -U.

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.

%terminates D' (map_eval' _ C D' _).
```

□

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 η and de Bruijn expression F, if there exist an e and a translation $\mathcal{C} :: \eta \vdash F \leftrightarrow e$, 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 η, value W, deductions $\mathcal{D}' :: \eta \vdash F \hookrightarrow W$, $\mathcal{C} :: \eta \vdash F \leftrightarrow e$, 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 ($'$).

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

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 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

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

requires that an environment contain only values. We will thus need to add a new environment constructor $\eta + F$ 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.

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

The Elf representation is direct.

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

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

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.

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

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

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

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

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

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

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

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

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

For example, **fix** $x.\ x$ (considered on page 17) is represented by $\mathbf{fix}'\ 1$. Intuitively, evaluation of this expression should not terminate. An attempt to construct an evaluation leads to the sequence

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

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.
%mode feval +N +F -W.

% Variables
fev_1 : feval (N , W) 1 W.
fev_^ : feval (N , W') (F ^) W
           <- feval N F W.

fev_1+ : feval (N + F) 1 W
            <- feval N F W.
fev_^+ : feval (N + F') (F ^) W
            <- feval N F W.

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

% Pairs
fev_pair : feval N (pair' F1 F2) (pair* W1 W2)
             <- feval N F1 W1
             <- feval N F2 W2.
fev_fst  : feval N (fst' F) W1
             <- feval N F (pair* W1 W2).
fev_snd  : feval N (snd' F) W2
             <- feval N F (pair* W1 W2).

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

% Definitions
fev_letv : feval N (letv' F1 F2) W
```

```
            <- feval N F1 W1
            <- feval (N , W1) F2 W.

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

% Recursion
fev_fix  : feval N (fix' F) W
             <- feval (N + (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 $\eta \vdash F \leftrightarrow e$ 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.

$$\frac{}{\eta \vdash \mathbf{z}' \leftrightarrow \mathbf{z}}\,\mathsf{tr\_z} \qquad\qquad \frac{\eta \vdash F \leftrightarrow e}{\eta \vdash \mathbf{s}'\ F \leftrightarrow \mathbf{s}\ e}\,\mathsf{tr\_s}$$

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

where the right premiss of $\mathsf{tr\_let}$ 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 $+$.

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

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

$$\dfrac{\eta \vdash F \leftrightarrow e}{\eta + F \vdash 1 \leftrightarrow e}\,\mathsf{tr\_1+} \qquad \dfrac{\eta \vdash F \leftrightarrow e}{\eta + F' \vdash F\!\uparrow\, \leftrightarrow e}\,\mathsf{tr\_\uparrow+}$$

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

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

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

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.
% can be used in different directions
%mode trans +N +F -E.
%mode vtrans +W -V.

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

```
tr_case : trans N (case' F1 F2 F3) (case E1 E2 E3)
           <- trans N F1 E1
           <- trans N F2 E2
           <- ({w:val} {x:exp}
                 vtrans w x -> trans (N , w) F3 (E3 x)).

% Pairs
tr_pair : trans N (pair' F1 F2) (pair E1 E2)
             <- trans N F1 E1
             <- trans N F2 E2.
tr_fst  : trans N (fst' F1) (fst E1)
             <- trans N F1 E1.
tr_snd  : trans N (snd' F1) (snd E1)
             <- trans N F1 E1.

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

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

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

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

% Variables
tr_1  : trans (N , W) 1 E <- vtrans W E.
tr_^  : trans (N , W) (F ^) E <- trans N F E.
```

```
tr_1+ : trans (N + F) 1 E <- trans N F E.
tr_^+ : trans (N + F') (F ^) E <- trans N 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.

% Function values
vtr_lam : vtrans (clo N (lam' F)) (lam E)
           <- trans N (lam' F) (lam E).
```

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.
%mode vtrans_val +U -P.

val_eval  : value E -> eval E E -> type.
%mode val_eval +P -D.
```

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 η, de Bruijn expression F, deduction $\mathcal{D} :: e \hookrightarrow v$ and $\mathcal{C} :: \eta \vdash F \leftrightarrow e$, there exist a value W and deductions $\mathcal{D}' :: \eta \vdash F \hookrightarrow W$ and $\mathcal{U} :: W \Leftrightarrow v$.*

Proof: By induction on the structure of $\mathcal{D} :: e \hookrightarrow v$ and $\mathcal{C} :: \eta \vdash F \leftrightarrow e$. 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 N F E
               -> feval N F W -> vtrans W V -> type.
%mode map_eval +D +C -D' -U.
```

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 $\mathsf{tr\_1}$ rule.

$$\mathcal{C} = \frac{\begin{array}{c}\mathcal{U}_1\\ W_1 \Leftrightarrow e\end{array}}{\eta_1, W_1 \vdash 1 \leftrightarrow e}\,\mathsf{tr\_1}$$

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 $\mathsf{fev\_1} :: \eta_1, W_1 \vdash 1 \hookrightarrow W_1$. 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 tr_1+ rule.

$$\mathcal{C} = \cfrac{\begin{array}{c}\mathcal{C}_1\\ \eta_1 \vdash F_1 \leftrightarrow e\end{array}}{\eta_1 + F_1 \vdash 1 \leftrightarrow e}\,\mathsf{tr\_1+}$$

$\mathcal{D} :: e \hookrightarrow v$	Assumption
$\mathcal{D}_1' :: \eta_1 \vdash F_1 \hookrightarrow W_1$ and	
$\mathcal{U}_1 :: W_1 \Leftrightarrow v$	By ind. hyp. on $\mathcal{D}$ and $\mathcal{C}_1$
$\mathcal{D}' :: \eta_1 + F_1 \vdash 1 \hookrightarrow W$	By 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 tr_↑+ rule. This case is just like the tr_↑ 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 ev_fix rule.

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

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

By inversion and exclusion (of the previous cases), $\mathcal{C}$ must end in an application of the 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

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

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 $u :: \eta \vdash \mathbf{fix}'\ F_1 \leftrightarrow \mathbf{fix}\ x.\ e_1$ by $\mathcal{C}$! This way we obtain a deduction

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

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

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

Applying fev_fix to $\mathcal{D}_1'$ results in a deduction

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

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 η, value W, deduction $\mathcal{D}' :: \eta \vdash F \hookrightarrow W$ and $\mathcal{C} :: \eta \vdash F \leftrightarrow e$, 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 $\mathcal{D}' :: \eta \vdash F \hookrightarrow W$. 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 η and de Bruijn expression F, if there exist an e and a translation $\mathcal{C} :: \eta \vdash F \leftrightarrow e$, 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 η, value W, deductions $\mathcal{D}' :: \eta \vdash F \hookrightarrow W$, $\mathcal{C} :: \eta \vdash F \leftrightarrow e$, 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.

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

An abstract machine would presumably start in a state where it is given the environment η 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 η, the second phase computes the value of F_2 in environment η, and the third phase combines the two values to form a pair. These phases mean that we have to preserve the environment η 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 Ξ 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:

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

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 Ξ, a program P and a value stack S, written as $\langle \Xi, P, S \rangle$. We have single-step and multi-step transition judgments:

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

We define the transition judgment so that

$$\langle (\cdot; \eta), F \,\&\, \mathit{done}, \cdot \rangle \stackrel{*}{\Longrightarrow} \langle \cdot, \mathit{done}, (\cdot, W) \rangle$$

corresponds to the evaluation of F in environment η 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

$$\eta \vdash F \stackrel{*}{\Longrightarrow}\!\!\!\!\!\!\!\!\!\;\;\;\; W \quad F \text{ computes to } W \text{ in environment } \eta$$

which is defined by the single inference rule

$$\frac{\langle (\cdot; \eta), F \,\&\, \mathit{done}, \cdot \rangle \stackrel{*}{\Longrightarrow} \langle \cdot, \mathit{done}, (\cdot, W) \rangle}{\eta \vdash F \stackrel{*}{\Longrightarrow} W} \,\mathsf{run}.$$

We prove in Theorem 6.19 that $\eta \vdash F \stackrel{*}{\Longrightarrow} W$ iff $\eta \vdash F \hookrightarrow W$. 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

$$\langle(\Xi;\eta), F \,\&\, P, S\rangle \stackrel{*}{\Longrightarrow} \langle\Xi, P, (S, W)\rangle$$

if $\eta \vdash F \hookrightarrow W$. 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 η on the environment stack.

$$\mathsf{c\_pair} :: \langle(\Xi;\eta), \langle F_1, F_2\rangle \,\&\, P, S\rangle \Longrightarrow \langle(\Xi;\eta;\eta), F_1 \,\&\, F_2 \,\&\, \mathit{mkpair} \,\&\, P, S\rangle$$

Here $\mathsf{c\_pair}$ 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

$$\langle(\Xi;\eta), F_2 \,\&\, \mathit{mkpair} \,\&\, P, (S, W_1)\rangle,$$

and the further evaluation of F_2 then leads to a state

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

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:

$$\mathsf{c\_mkpair} :: \langle\Xi, \mathit{mkpair} \,\&\, P, (S, W_1, W_2)\rangle \Longrightarrow \langle\Xi, P, (S, \langle W_1, W_2\rangle^*)\rangle.$$

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*.

$$\begin{array}{rcl}
\mathsf{c_app} & :: & \langle(\Xi;\eta), F_1\ F_2 \,\&\, P, S\rangle \Longrightarrow \langle(\Xi;\eta;\eta), F_1 \,\&\, F_2 \,\&\, \mathit{apply} \,\&\, P, S\rangle \\
\mathsf{c_apply} & :: & \langle\Xi, \mathit{apply} \,\&\, P, (S, \{\eta'; \Lambda F_1'\}, W_2)\rangle \Longrightarrow \langle(\Xi;(\eta', W_2)), F_1' \,\&\, P, S\rangle
\end{array}$$

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

$$\begin{array}{rcl}
\mathsf{c_z} & :: & \langle(\Xi;\eta), \mathbf{z} \,\&\, P, S\rangle \Longrightarrow \langle\Xi, P, (S, \mathbf{z}^*)\rangle \\
\mathsf{c_s} & :: & \langle(\Xi;\eta), \mathbf{s}\ F \,\&\, P, S\rangle \Longrightarrow \langle(\Xi;\eta), F \,\&\, \mathit{add1} \,\&\, P, S\rangle \\
\mathsf{c_add1} & :: & \langle\Xi, \mathit{add1} \,\&\, P, (S, W)\rangle \Longrightarrow \langle\Xi, P, (S, \mathbf{s}^*\ W)\rangle
\end{array}$$

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.

$$\begin{aligned}
\mathsf{c_case} :: {} & \langle(\Xi;\eta), \mathbf{case}\ F_1\ F_2\ F_3 \,\&\, P, S\rangle \\
& \Longrightarrow \langle(\Xi;\eta;\eta), F_1 \,\&\, \mathit{branch} \,\&\, F_2 \,\&\, F_3 \,\&\, P, S\rangle \\
\mathsf{c_branch_z} :: {} & \langle(\Xi;\eta), \mathit{branch} \,\&\, F_2 \,\&\, F_3 \,\&\, P, (S, \mathbf{z}^*)\rangle \Longrightarrow \langle(\Xi;\eta), F_2 \,\&\, P, S\rangle \\
\mathsf{c_branch_s} :: {} & \langle(\Xi;\eta), \mathit{branch} \,\&\, F_2 \,\&\, F_3 \,\&\, P, (S, \mathbf{s}^*\ W)\rangle \\
& \Longrightarrow \langle(\Xi;(\eta, W)), F_3 \,\&\, P, S\rangle
\end{aligned}$$

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

$$\begin{aligned}
\mathsf{c_fst} &:: \langle(\Xi;\eta), \mathbf{fst}\ F \,\&\, P, S\rangle \Longrightarrow \langle(\Xi;\eta), F \,\&\, \mathit{getfst} \,\&\, P, S\rangle \\
\mathsf{c_getfst} &:: \langle\Xi, \mathit{getfst} \,\&\, P, (S, \langle W_1, W_2\rangle^*)\rangle \Longrightarrow \langle\Xi, P, (S, W_1)\rangle \\
\mathsf{c_snd} &:: \langle(\Xi;\eta), \mathbf{snd}\ F \,\&\, P, S\rangle \Longrightarrow \langle(\Xi;\eta), F \,\&\, \mathit{getsnd} \,\&\, P, S\rangle \\
\mathsf{c_getsnd} &:: \langle\Xi, \mathit{getsnd} \,\&\, P, (S, \langle W_1, W_2\rangle^*)\rangle \Longrightarrow \langle\Xi, P, (S, W_2)\rangle
\end{aligned}$$

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).

$$\begin{aligned}
\mathsf{c_let} :: {} & \langle(\Xi;\eta), \mathbf{let}\ F_1\ \mathbf{in}\ F_2 \,\&\, P, S\rangle \Longrightarrow \langle(\Xi;\eta;\eta), F_1 \,\&\, \mathit{bind} \,\&\, F_2 \,\&\, P, S\rangle \\
\mathsf{c_bind} :: {} & \langle(\Xi;\eta), \mathit{bind} \,\&\, F_2 \,\&\, P, (S; W_1)\rangle \Longrightarrow \langle(\Xi;(\eta, W_1)), F_2 \,\&\, P, S\rangle
\end{aligned}$$

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.

$$\begin{aligned}
\mathsf{c_1} &:: \langle(\Xi;(\eta, W)), 1 \,\&\, P, S\rangle \Longrightarrow \langle\Xi, P, (S, W)\rangle \\
\mathsf{c_\uparrow} &:: \langle(\Xi;(\eta, W')), F\!\uparrow \,\&\, P, S\rangle \Longrightarrow \langle(\Xi;\eta), F \,\&\, P, S\rangle \\
\mathsf{c_lam} &:: \langle(\Xi;\eta), \Lambda F \,\&\, P, S\rangle \Longrightarrow \langle\Xi, P, (S, \{\eta; \Lambda F\})\rangle
\end{aligned}$$

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.

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

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 $St \stackrel{*}{\Longrightarrow} St'$ 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.

$$\frac{\langle (\cdot;\eta), F \;\&\; \mathit{done}, \cdot\rangle \stackrel{*}{\Longrightarrow} \langle \cdot, \mathit{done}, (\cdot, W)\rangle}{\eta \vdash F \stackrel{*}{\Longrightarrow\!\!\!\!\!\!\Longrightarrow} W}\,\textsf{run}$$

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 Ns.
state       : type.  %name state St.

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

% Programs
done : program.
&    : instruction -> program -> program.
```

```
%infix right 10 &.

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

% States
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.          %name => R.
%infix none 10 =>.
%mode => +St -St'.

c_z       : st (Ns ; N) (ev z' & P) S => st Ns P (S , z*).

c_app     : st (Ns ; N) (ev (app' F1 F2) & P) S
         => st (Ns ; N ; N) (ev F1 & ev F2 & apply & P) S.
c_apply   : st Ns (apply & P) (S , clo N' (lam' F1') , W2)
         => st (Ns ; (N' , 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.               %name =>* C.
%infix none 10 =>*.
% no mode---this is not operational.

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 $\eta \vdash F \overset{*}{\Longrightarrow} W$.

```
ceval : env -> exp' -> val -> type.
% no mode---this is not operational

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

While this representation is declaratively adequate it has a serious operational defect when used for evaluation, that is, when η 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 $\mathcal{C}'' :: St \stackrel{*}{\Longrightarrow} St''$ to finding a state St' and computation of $\mathcal{C}' :: St' \stackrel{*}{\Longrightarrow} St''$ 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 $\mathcal{C}' :: St' \stackrel{*}{\Longrightarrow} St''$. 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 >=>*.
%mode >=>* +St -St'.

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

>ceval : env -> exp' -> val -> type.
%mode >ceval +N +F -W.

>run   : >ceval N F W
          <- st (emptys ; N) (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 Ns P S =>* st Ns' P' S'
      -> st Ns' P' S' =>* st Ns'' P'' S''
      -> st Ns P S =>* st Ns'' P'' S''
      -> type.
%mode append +C +C' -C''.
```

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 η be an environment, F an expression, and W a value such that $\eta \vdash F \hookrightarrow W$. Then, for any environment stack Ξ, program P and stack S,*

$$\langle (\Xi; \eta), F \,\&\, P, S \rangle \stackrel{*}{\Longrightarrow} \langle \Xi, P, (S, W) \rangle$$

Proof: By induction on the structure of $\mathcal{D} :: \eta \vdash F \hookrightarrow W$. We will construct a deduction of $\mathcal{C} :: \langle (\Xi; \eta), F \,\&\, P, S \rangle \stackrel{*}{\Longrightarrow} \langle \Xi, P, (S, W) \rangle$. 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 N F W
          -> st (Ns ; N) (ev F & P) S =>* st Ns P (S , W)
          -> type.
%mode +{N:env} +{F:exp'} +{W:val} +{Ns:envstack} +{P:program} +{S:env}
    +{D:feval N F W} -{C:st (Ns ; N) (ev F & P) S =>* st Ns P (S , W)}
    subcomp D C.
```

Note that the mode declaration here is in the "full" form in order to express that the implicitly quantified variables `Ns`, `P`, and `S` are input arguments rather than output arguments. The system would have inferred the latter from the simpler declaration `%mode subcomp +D -C`.

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

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

Then the single-step transition

$$\langle(\Xi;\eta), \mathbf{z} \,\&\, P, S\rangle \stackrel{*}{\Longrightarrow} \langle\Xi, P, (S, \mathbf{z}^*)\rangle$$

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.

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

Then

$$\begin{array}{ll}
\langle(\Xi;\eta), F_1\ F_2 \,\&\, P, S\rangle & \\
\Longrightarrow \langle(\Xi;\eta;\eta), F_1 \,\&\, F_2 \,\&\, \mathit{apply} \,\&\, P, S\rangle & \text{By rule } \textsf{c_app} \\
\stackrel{*}{\Longrightarrow} \langle(\Xi;\eta), F_2 \,\&\, \mathit{apply} \,\&\, P, (S, \{\eta'; \Lambda F_1'\})\rangle & \text{By ind. hyp. on } \mathcal{D}_1 \\
\stackrel{*}{\Longrightarrow} \langle\Xi, \mathit{apply} \,\&\, P, (S, \{\eta'; \Lambda F_1'\}, W_2)\rangle & \text{By ind. hyp. on } \mathcal{D}_2 \\
\Longrightarrow \langle(\Xi; (\eta', W_2)), F_1' \,\&\, P, S\rangle & \text{By rule } \textsf{c_apply} \\
\stackrel{*}{\Longrightarrow} \langle\Xi, P, (S, W)\rangle & \text{By ind. hyp. on } \mathcal{D}_3.
\end{array}$$

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 $\mathcal{C} :: St' \stackrel{*}{\Longrightarrow} St''$. 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.

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

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 Ns1 P1 S1) =>* (st Ns P S)
     -> (st Ns2 P2 S2) =>* (st Ns P S)
     -> type.
%infix none 8 <.
sub_imm : C < R ~ C.

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

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*

$$\mathcal{C} :: \langle(\Xi,\eta), F \,\&\, P, S\rangle \stackrel{*}{\Longrightarrow} \langle\cdot, \mathit{done}, (\cdot, W')\rangle$$

there exists a value W*, an evaluation*

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

and a computation

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

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 three deductions: $\mathcal{C}$, $\mathcal{C}'$, and $\mathcal{D}$. We do not explicitly represent the derivation which shows that $\mathcal{C}' < \mathcal{C}$ since the Twelf system can check this automatically using a `%reduces` declaration, placed after all rules for `spl`.

```
spl : (st (Ns ; N) (ev F & P) S) =>* (st emptys done (empty , W'))
       -> feval N F W
       -> (st Ns P (S , W)) =>* (st emptys done (empty , W'))
       -> type.
%mode spl +C -D -C'.
... clauses for spl ...
%reduces C' < C (spl C _ C').
%terminates C (spl C _ _).
```

Now back to the proof.

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.
```

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

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

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

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

and a computation

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

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

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

and a computation

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

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

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

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

$$\mathcal{D}_3 :: \eta', W_2 \vdash F_1' \hookrightarrow W_3$$

and a compuation

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

Now we let $W = W_3$ and we construct $\mathcal{D} :: \eta \vdash F_1\ F_2 \hookrightarrow W_3$ 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
            <- spl C1 D1 C2
            <- spl C2 D2 (c_apply ~ C3')
            <- spl C3' D3 C4.
```

□

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

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

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

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

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

$$\langle (\Xi; \eta), F \,\&\, P, S \rangle \stackrel{*}{\Longrightarrow} \langle \Xi, P, (S, W) \rangle$$

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

```
cev_complete : feval N F W -> ceval N F W -> type.
%mode cev_complete +D -C.
cevc : cev_complete D (run C) <- subcomp D C.
%terminates [] (cev_complete D _).
```

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

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

and a computation

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

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 N F W -> feval N F W -> type.
%mode cls_sound +C -D.
clss : cls_sound (run C) D <- spl C D (id).
%terminates [] (cls_sound C _).
```

□

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.

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

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 as $\mathbf{ev}\ e$, 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.

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

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

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

$$\frac{}{\textbf{init} \diamond \textbf{return}\ v \Longrightarrow \textbf{answer}\ v}\ \mathsf{st\_init}$$

$$\frac{}{K; \lambda x.\ i \diamond \textbf{return}\ v \Longrightarrow K \diamond [v/x]i}\ \mathsf{st\_return}$$

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. First, the constructors:

$$\begin{array}{lllllll}
\mathsf{st_z} & :: & K & \diamond & \textbf{ev z} & \Longrightarrow & K & \diamond & \textbf{return z} \\
\mathsf{st_s} & :: & K & \diamond & \textbf{ev}\ (\textbf{s}\ e) & \Longrightarrow & K; \lambda x.\ \textbf{return}\ (\textbf{s}\ x) & \diamond & e
\end{array}$$

Second, the corresponding destructor:

$$\begin{array}{llllll}
\mathsf{st_case} & :: & K & \diamond & \textbf{ev}\ (\textbf{case}\ e_1\ \textbf{of z} \Rightarrow e_2 \mid \textbf{s}\ x \Rightarrow e_3) \\
& & & & \qquad \Longrightarrow K; \lambda x_1.\ \textbf{case}_1\ x_1\ \textbf{of z} \Rightarrow e_2 \mid \textbf{s}\ x \Rightarrow e_3 \ \diamond\ \textbf{ev}\ e_1 \\
\mathsf{st_case_1_z} & :: & K & \diamond & \textbf{case}_1\ \textbf{z of z} \Rightarrow e_2 \mid \textbf{s}\ x \Rightarrow e_3 \quad \Longrightarrow \quad K \ \diamond\ \textbf{ev}\ e_2 \\
\mathsf{st_case_1_s} & :: & K & \diamond & \textbf{case}_1\ \textbf{s}\ v_1\ \textbf{of z} \Rightarrow e_2 \mid \textbf{s}\ x \Rightarrow e_3 \quad \Longrightarrow \quad K \ \diamond\ \textbf{ev}\ [v_1'/x]e_3
\end{array}$$

We can see that the **case** construct requires a new instruction of the form $\textbf{case}_1\ v_1\ \textbf{of z} \Rightarrow e_2 \mid \textbf{s}\ x \Rightarrow e_3$. This is distinct from $\textbf{case}\ e_1\ \textbf{of z} \Rightarrow e_2 \mid \textbf{s}\ x \Rightarrow e_3$ 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 $\langle v_1, e_2\rangle_1$, $\textbf{fst}_1$, and $\textbf{snd}_1$.

$$\begin{array}{lllllllll}
\mathsf{st_pair} & :: & K & \diamond & \textbf{ev}\ \langle e_1, e_2\rangle & \Longrightarrow & K; \lambda x_1.\ \langle x_1, e_2\rangle_1 & \diamond & \textbf{ev}\ e_1 \\
\mathsf{st_pair_1} & :: & K & \diamond & \langle v_1, e_2\rangle_1 & \Longrightarrow & K; \lambda x_1.\ \textbf{return}\ \langle v_1, x_2\rangle & \diamond & \textbf{ev}\ e_2 \\
\mathsf{st_fst} & :: & K & \diamond & \textbf{ev}\ (\textbf{fst}\ e) & \Longrightarrow & K; \lambda x.\ \textbf{fst}_1\ x & \diamond & \textbf{ev}\ e \\
\mathsf{st_fst_1} & :: & K & \diamond & \textbf{fst}_1\ \langle v_1, v_2\rangle & \Longrightarrow & K & \diamond & \textbf{return}\ v_1 \\
\mathsf{st_snd} & :: & K & \diamond & \textbf{ev}\ (\textbf{snd}\ e) & \Longrightarrow & K; \lambda x.\ \textbf{snd}_1\ x & \diamond & \textbf{ev}\ e \\
\mathsf{st_snd_1} & :: & K & \diamond & \textbf{snd}_1\ \langle v_1, v_2\rangle & \Longrightarrow & K & \diamond & \textbf{return}\ v_2
\end{array}$$

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

$$\begin{array}{lllllll}
\mathsf{st_lam} & :: & K & \diamond & \textbf{ev}\ (\textbf{lam}\ x.\ e) & \Longrightarrow & K \diamond \textbf{return lam}x.\ e \\
\mathsf{st_app} & :: & K & \diamond & \textbf{ev}\ (e_1\ e_2) & \Longrightarrow & K; \lambda x_1.\ \textbf{app}_1\ x_1\ e_2 \diamond \textbf{ev}\ e_1 \\
\mathsf{st_app_1} & :: & K & \diamond & \textbf{app}_1\ v_1\ e_2 & \Longrightarrow & K; \lambda x_2.\ \textbf{app}_2\ v_1\ x_2 \diamond \textbf{ev}\ e_2 \\
\mathsf{st_app_2} & :: & K & \diamond & \textbf{app}_2\ (\textbf{lam}\ x.\ e_1')\ v_2 & \Longrightarrow & K \diamond \textbf{ev}\ ([v_2/x]e_1') \\
\\
\mathsf{st_letv} & :: & K & \diamond & \textbf{let val}\ x = e_1\ \textbf{in}\ e_2 & \Longrightarrow & K; \lambda x_1.\ [x_1/x]e_2 \diamond \textbf{ev}\ e_1 \\
\mathsf{st_letn} & :: & K & \diamond & \textbf{let name}\ u = e_1\ \textbf{in}\ e_2 & \Longrightarrow & K \diamond \textbf{ev}\ ([e_1/x]e_2) \\
\mathsf{st_fix} & :: & K & \diamond & \textbf{fix}\ u.\ e & \Longrightarrow & K \diamond \textbf{ev}\ ([\textbf{fix}\ u.\ e/u]e)
\end{array}$$

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

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

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 : exp -> inst.

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

% Continuations
cont : type.  %name cont K.

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

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

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

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

```
=> : state -> state -> type.          %name => C.
%infix none 6 =>.
%mode => +S -S'.
```

```
% Natural Numbers
st_z : K # (ev z) => K # (return z).
st_s : K # (ev (s E')) => (K ; [x'] return (s x')) # (ev E').
st_case : K # (ev (case E1 E2 E3)) => (K ; [x1] 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] pair1 x1 E2) # (ev E1).
st_pair1 : K # (pair1 V1 E2) => (K ; [x2] return (pair V1 x2)) # (ev E2).
st_fst : K # (ev (fst E')) => (K ; [x'] fst1 x') # (ev E').
st_fst1 : K # (fst1 (pair V1 V2)) => K # (return V1).
st_snd : K # (ev (snd E')) => (K ; [x'] 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] app1 x1 E2) # (ev E1).
st_app1 : K # (app1 V1 E2) => (K ; [x2] 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] 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'))).

% Return Instructions
st_return : (K ; [x] I x) # (return V) => K # (I 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.

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

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

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

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 =>*.
%mode =>* +S -S'.

stop : S =>* S.
<< : S =>* S''
      <- S => S'
      <- S' =>* S''.
%infix left 5 <<.
% Because of evaluation order, computation sequences are displayed
% in reverse, using "<<" as a left-associative infix operator.

ceval : exp -> exp -> type.           %name ceval CE.
%mode ceval +E -V.

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] x) z)) =>* answer V.
Solving...
V = z.
C* =
 stop << st_init << st_z << 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 $e \stackrel{c}{\hookrightarrow} v$. In one direction we have to find a translation from tree-structured derivations $\mathcal{D} :: e \hookrightarrow v$ to sequential computations $\mathcal{C} :: \mathbf{init} \diamond \mathbf{ev}\ e \stackrel{*}{\Longrightarrow} \mathbf{answer}\ v$. 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 $e \stackrel{c}{\hookrightarrow} v$. This immediately reduces to showing that $\mathbf{init} \diamond \mathbf{ev}\ e \stackrel{*}{\Longrightarrow} \mathbf{answer}\ v$. 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 $K \diamond \mathbf{ev}\ e \stackrel{*}{\Longrightarrow} K \diamond \mathbf{return}\ v$, then it follows directly by induction, using some simple lemmas regarding the concatenation of computation sequences (see Exercise 6.17).

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 $K \diamond \mathbf{ev}\ e \stackrel{*}{\Longrightarrow} K \diamond \mathbf{return}\ v$, 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 6.22 *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 $\mathcal{C} :: K \diamond \mathbf{ev}\ e \stackrel{*}{\Longrightarrow}$ $\mathbf{answer}\ w$.*

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.
%mode ccp +D +C' -C.
```

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:

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

$\mathcal{C}' :: K \diamond \mathbf{return}\ \mathbf{lam}\ x.\ e_1 \stackrel{*}{\Longrightarrow} \mathbf{answer}\ w$	Assumption
$\mathcal{C} :: K \diamond \mathbf{ev}(\mathbf{lam}\ x.\ e_1) \Longrightarrow \mathbf{answer}\ w$	By $\mathsf{st\_lam}$ followed by $\mathcal{C}'$

In this case we have added a step st_lam to a computation; in the implementation, this will be an application of the 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:

$$\mathcal{D} = \dfrac{\begin{array}{ccc} \mathcal{D}_1 & \mathcal{D}_2 & \mathcal{D}_3 \\ e_1 \hookrightarrow \mathbf{lam}\, x.\, e_1' & e_2 \hookrightarrow v_2 & [v_2/x]e_1' \hookrightarrow v \end{array}}{e_1\ e_2 \hookrightarrow v}\ \mathsf{ev\_app}$$

$\mathcal{C}' :: K \diamond \mathbf{return}\ v \stackrel{*}{\Longrightarrow} \mathbf{answer}\ w$ — Assumption

$\mathcal{C}_3 :: K \diamond \mathbf{ev}\ ([v_2/x]e_1') \stackrel{*}{\Longrightarrow} \mathbf{answer}\ w$ — 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 $\mathcal{C}_3$

$\mathcal{C}_2 :: K; \lambda x_2.\ \mathbf{app}_2\ (\mathbf{lam}\ x.\ e_1')\ x_2 \diamond \mathbf{ev}\ e_2 \stackrel{*}{\Longrightarrow} \mathbf{answer}\ w$
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 $\mathcal{C}_2$.

$\mathcal{C}_1 :: K; \lambda x_1.\ \mathbf{app}_1\ x_1\ e_2 \diamond \mathbf{ev}\ e_1 \stackrel{*}{\Longrightarrow} \mathbf{answer}\ w$ — By ind. hyp. on $\mathcal{D}_1$ and $\mathcal{C}_1'$

$\mathcal{C} :: K \diamond \mathbf{ev}\ (e_1\ e_2) \stackrel{*}{\Longrightarrow} \mathbf{answer}\ w$ — 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 6.23 (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 6.22 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 $\mathcal{C} :: \mathbf{init} \diamond \mathbf{ev}\ e \stackrel{*}{\Longrightarrow} \mathbf{answer}\ v$. Therefore, by rule cev, $e \stackrel{c}{\hookrightarrow} v$.

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

```
cpm_complete : eval E V -> ceval E V -> type.
%mode cpm_complete +D -C.
cpmcp : cpm_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

$$K \diamond \mathbf{ev}\ (e_1\ e_2) \stackrel{*}{\Longrightarrow} K; \lambda x_1.\, \mathbf{app}_1\ x_1\ e_2 \diamond \mathbf{ev}\ e_1$$

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 $K \diamond \mathbf{ev}\ e$ 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 6.24 *For any continuation K, closed expression e and value w, if $\mathcal{C} :: K \diamond \mathbf{ev}\ e \stackrel{*}{\Longrightarrow} \mathbf{answer}\ w$ then there is a value v a derivation $\mathcal{D} :: e \hookrightarrow v$, and a subcomputation $\mathcal{C}'$ of $\mathcal{C}$ of the form $K \diamond \mathbf{return}\ v \stackrel{*}{\Longrightarrow} \mathbf{answer}\ w$.*

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.
%mode csd +C -D -C'.
```

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

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

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 $\mathcal{C}_1 :: K; \lambda x_1.\ \mathbf{app}_1\ x_1\ e_2 \diamond \mathbf{ev}\ e_1 \stackrel{*}{\Longrightarrow} \mathbf{answer}\ w$, where $e = e_1\ e_2$.

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

$\square$

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

Theorem 6.25 (Soundness of the Continuation Machine) *For any closed expression e and value v, if $e \stackrel{c}{\hookrightarrow} v$ then $e \hookrightarrow v$.*

Proof: By inversion, $\mathcal{C} :: \mathbf{init} \diamond \mathbf{ev}\ e \stackrel{*}{\Longrightarrow} \mathbf{answer}\ v$. By Lemma 6.24 there is a derivation $\mathcal{D} :: e \hookrightarrow v'$ and $\mathcal{C}' :: \mathbf{init} \diamond \mathbf{return}\ v' \stackrel{*}{\Longrightarrow} \mathbf{answer}\ v$ for some v'. By inversion on $\mathcal{C}'$ we see that $v = v'$ and therefore $\mathcal{D}$ satisfies the requirements of the theorem.

```
cpm_sound : ceval E V -> eval E V -> type.
%mode cpm_sound +C -D.

cpmsd : cpm_sound (cev C) D
         <- csd C D (stop << st_init).

%terminates {} (cpm_sound C D).
```

□

6.6 Type Preservation and Progress

So far we have concentrated on operational aspects of the translation from a big-step to a small-step semantics; now we turn to issues of typing. The first property is type preservation—a reprise of the same property for the big-step semantics. But the reformulation of the semantics also allows us to express new language properties, still at a high level of abstraction. One of the most important ones is *progress*. It states that in any valid abstract machine state we can have one of two situations: either we have already computed the final answer of the program, or we can make progress by taking a further step.

Type preservation and progress together express that *well-typed programs cannot go wrong*, a phrase coined by Milner [Mil78]. In our setting, "*going wrong*" corresponds to reaching a machine state in which no further transition rules applicable. Note that type preservation for the big-step semantics does *not* express this, since it only talks about completed evaluations, not about intermediate states. Of course, even in the small-step semantics an expression can fail to have a value, but from the progress theorem we know that this is only due to non-termination.

We begin by giving the typing rules for the continuation-passing machine. One complication as compared to the typing rules for expressions is that certain machine states only make sense when components of instructions are values. For example, the instruction $\mathbf{app}_2\ v_1\ v_2$ requires both arguments to be values. If this restriction is not enforced, the progress theorem clearly fails, because there is no transition for a state $K \diamond \mathbf{app}_2\ ((\mathbf{lam}\ x.\ x)\ (\mathbf{lam}\ y.\ y))\ \mathbf{z}$ even though the arguments to $\mathbf{app}_2$ are correctly typed. This introduces a further complication: since continuations are composed of functions from values to instructions, we need to record that the argument of the continuation is indeed a function. For this we introduce a new context Υ, which is either empty or contains a simple hypothesis x *Value*. The value judgment is appropriately generalized so that $x\ \mathit{Value} \triangleright x\ \mathit{Value}$.

$\Upsilon ::= \cdot \mid x\ \mathit{Value}$	Continuation argument
$\Upsilon; \Delta \triangleright i : \tau$	Instruction i has type τ in context Δ.
$\triangleright K : \tau \Rightarrow \sigma$	Continuation K maps values of type τ to answers of type σ
$\triangleright S : \sigma$	State S has type σ

Typing for continuations keeps track of two types: the type of the value that will be passed to it, and the type of the final answer it produces. States on the other hand record only the type of final answer it may produce. The judgments for continuations and state do not depend on a context because they never contain free variables.

$$\frac{\Delta \triangleright e : \tau}{\Upsilon; \Delta \triangleright \mathbf{ev}\ e : \tau}\ \mathsf{vi\_ev} \qquad \frac{\Delta \triangleright v : \tau \qquad \Upsilon \triangleright v\ \mathit{Value}}{\Upsilon; \Delta \triangleright \mathbf{return}\ v : \tau}\ \mathsf{vi\_return}$$

$$\frac{\Delta \triangleright v_1 : \mathbf{nat} \qquad \Delta \triangleright e_2 : \tau \qquad \Delta, x{:}\mathbf{nat} \triangleright e_2 : \tau \qquad \Upsilon \triangleright v_1\ \mathit{Value}}{\Upsilon; \Delta \triangleright (\mathbf{case}_1\ v_1\ \mathbf{of}\ \mathbf{z} \Rightarrow e_2\ \mid \mathbf{s}\ x \Rightarrow e_3) : \tau}\ \mathsf{vi\_case_1}$$

$$\frac{\Delta \triangleright v_1 : \tau_1 \qquad \Delta \triangleright e_2 : \tau_2 \qquad \Upsilon \triangleright v_1\ \mathit{Value}}{\Upsilon; \Delta \triangleright \langle v_1, e_2 \rangle_1 : \tau_1 \times \tau_2}\ \mathsf{vi\_pair_1}$$

$$\frac{\Delta \triangleright v' : \tau_1 \times \tau_2 \qquad \Upsilon \triangleright v'\ \mathit{Value}}{\Upsilon; \Delta \triangleright \mathbf{fst}_1\ v' : \tau_1}\ \mathsf{vi\_fst_1} \qquad \frac{\Delta \triangleright v' : \tau_1 \times \tau_2 \qquad \Upsilon \triangleright v'\ \mathit{Value}}{\Upsilon; \Delta \triangleright \mathbf{snd}_1\ v' : \tau_2}\ \mathsf{vi\_snd_1}$$

$$\frac{\Delta \triangleright v_1 : \tau_2 \to \tau_1 \qquad \Delta \triangleright e_2 : \tau_2 \qquad \Upsilon \triangleright v_1\ \mathit{Value}}{\Upsilon; \Delta \triangleright \mathbf{app}_1\ v_1\ e_2 : \tau_1}\ \mathsf{vi\_app_1}$$

$$\frac{\Delta \triangleright v_1 : \tau_2 \to \tau_1 \qquad \Delta \triangleright v_2 : \tau_2 \qquad \Upsilon \triangleright v_1\ \mathit{Value} \qquad \Upsilon \triangleright v_2\ \mathit{Value}}{\Upsilon; \Delta \triangleright \mathbf{app}_2\ v_1\ v_2 : \tau_1}\ \mathsf{vi\_app_2}$$

In the typing rules for continuations, we have to make sure that the parts of the continuation are composed properly: the value returned by the last instruction of a continuation matches the type accepted by the remaining continuation. The initial continuation just returns its argument as the final answer and therefore has type $\tau \Rightarrow \tau$.

$$\frac{}{\rhd\ \mathbf{init} : \tau \Rightarrow \tau}\ \mathsf{vk\_init}$$

$$\frac{x\ \mathit{Value}; \cdot, x{:}\tau \rhd i : \tau' \qquad \rhd K : \tau' \Rightarrow \sigma}{\rhd K; \lambda x.\, i : \tau \Rightarrow \sigma}\ \mathsf{vk\_;}$$

For a state we verify that the type of the instruction to be executed matches the one expected by the continuation, and assign the type of the final answer to the state.

$$\frac{\cdot; \cdot \rhd i : \tau \qquad \rhd K : \tau \Rightarrow \sigma}{\rhd (K \diamond i) : \sigma}\ \mathsf{vs\_\diamond}$$

$$\frac{\cdot \rhd v : \sigma \qquad \cdot \rhd v\ \mathit{Value}}{\rhd\ \mathbf{answer}\ v : \sigma}\ \mathsf{vs\_answer}$$

The typing judgments for instructions, continuations, and states admit more states as valid than can be reached from an initial state of the form $\mathbf{init} \diamond \mathbf{ev}\ e$ (see Exercise 6.19). However, they are accurate enough to permit proof of preservation and progress and is therefore appropriate for our purposes.

The implementation of the judgments in the logical framework is straightforward. The fact that we need at most one value variable is not explicitly represented. Instead, we use the usual techniques for parametric and hypothetical judgments by assuming `value x` for a new parameter `x`. The declarations below can be executed in Elf in order to check the validity of instructions, continuations and states and infer their most general types.

```
%%% Instructions
valid : inst -> tp -> type.          %name valid VL.
%mode valid +I *T.

% Evaluation and return
vi_ev : valid (ev E) T
         <- of E T.
vi_return : valid (return V) T
             <- of V T
             <- value V.

% Natural Numbers
vi_case1 : valid (case1 V1 E2 E3) T
            <- of V1 nat
            <- of E2 T
```

```
            <- ({x:exp} of x nat -> of (E3 x) T)
            <- value V1.

% Pairs
vi_pair1 : valid (pair1 V1 E2) (cross T1 T2)
            <- of V1 T1
            <- of E2 T2
            <- value V1.
vi_fst1 : valid (fst1 V') T1
           <- of V' (cross T1 T2)
           <- value V'.
vi_snd1 : valid (snd1 V') T2
           <- of V' (cross T1 T2)
           <- value V'.

% Functions
vi_app1 : valid (app1 V1 E2) T1
           <- of V1 (arrow T2 T1)
           <- of E2 T2
           <- value V1.
vi_app2 : valid (app2 V1 V2) T1
           <- of V1 (arrow T2 T1)
           <- of V2 T2
           <- value V1
           <- value V2.

%%% Continuations
validk : cont -> tp -> tp -> type.    %name validk VK.
%mode validk +K *T *S.

vk_init : validk (init) T T.
vk_; : validk (K ; [x] I x) T S
        <- ({x:exp} value x -> of x T -> valid (I x) T')
        <- validk K T' S.

%%% States
valids : state -> tp -> type.         %name valids VS.

vs_# : valids (K # I) S
        <- valid I T
        <- validk K T S.
vs_answer : valids (answer V) S
             <- of V S
             <- value V.
```

With this preparation, we can now prove preservation and progress. The proofs are quite straightforward, but somewhat tedious.

Theorem 6.26 (One-Step Type Preservation) *If* $\triangleright S : \sigma$ *and* $S \Longrightarrow S'$ *then* $\triangleright S' : \sigma$.

Proof: By cases on the derivation of $\mathcal{C} :: S \Longrightarrow S'$, applying several levels of inversion to the given typing derivation for S. The implementation is via a type family

```
vps : valids S T -> S => S' -> valids S' T -> type.
%mode vps +VS +C -VS'.
```

that relates the three derivation involved in the theorem. We show only a few representative cases; the full implementation can be found in the on-line course material.

Case: $\mathcal{C}$ is st_app:

$$K \diamond \mathbf{ev}\ (e_1\ e_2) \Longrightarrow K; \lambda x_1.\, \mathbf{app}_1\ x_1\ e_2 \diamond \mathbf{ev}\ e_1.$$

$\triangleright K \diamond \mathbf{ev}\ (e_1\ e_2) : \sigma$	Assumption
$\triangleright K : \tau \Rightarrow \sigma$ and	
$\cdot;\cdot \triangleright \mathbf{ev}\ (e_1\ e_2) : \tau$ for some τ	By inversion
$\cdot \triangleright e_1\ e_2 : \tau$	By inversion
$\cdot \triangleright e_1 : \tau_2 \rightarrow \tau$ and	
$\cdot \triangleright e_2 : \tau_2$ for some τ_2	By inversion
$x_1\ \mathit{Value}; x_1 : \tau_2 \rightarrow \tau \triangleright \mathbf{app}_1\ x_1\ e_2 : \tau$	By rule (vi_app$_1$)
$\triangleright (K; \lambda x_1.\, \mathbf{app}_1\ x_1\ e_2) : (\tau_2 \rightarrow \tau) \Rightarrow \sigma$	By rule (vk_;)
$\triangleright (K; \lambda x_1.\, \mathbf{app}_1\ x_1\ e_2) \diamond \mathbf{ev}\ e_1 : \sigma$	By rule (vs_ $\diamond$)

The inversion steps are represented by expanding the structure of the first argument, obtaining access to the subderivations VK of $\triangleright K : \tau \rightarrow \sigma$, P2 of $\cdot \triangleright e_2 : \tau_2$, and P1 of $\cdot \triangleright e_1 : \tau_2 \rightarrow \tau$. The needed derivation for S' is readily constructed from these variables.

```
vps_app : vps (vs_# VK (vi_ev (tp_app P2 P1))) (st_app)
          (vs_# (vk_; VK ([x1] [q1:value x1]
                               [p1:of x1 (arrow T2 T1)]
                                 vi_app1 q1 P2 p1))
             (vi_ev P1)).
```

Case: $\mathcal{C}$ is st_app$_1$:

$$K \diamond \mathbf{app}_1\ v_1\ e_2 \Longrightarrow K; \lambda x_2.\, \mathbf{app}_2\ v_1\ x_2 \diamond \mathbf{ev}\ e_2.$$

$\triangleright K \diamond \mathbf{app}_1\ v_1\ e_2 : \sigma$	Assumption
$\triangleright K : \tau \Rightarrow \sigma$ and	
$\cdot;\cdot \triangleright \mathbf{app}_1\ v_1\ e_2 : \tau$ for some τ	By inversion
$\cdot \triangleright v_1 : \tau_2 \rightarrow \tau$ and	
$\cdot \triangleright e_2 : \tau_2$ form some τ_2 and	
$\cdot \triangleright v_1\ \mathit{Value}$	By inversion
$x_2\ \mathit{Value}; x_2 : \tau_2 \triangleright\!\triangleright \mathbf{app}_2\ v_1\ x_2 : \tau$	By rule ($\mathsf{vi\_app_2}$)
$\triangleright (K; \lambda x_2.\, \mathbf{app}_1\ v_1\ e_2) : \tau_2 \Rightarrow \sigma$	By rule ($\mathsf{vk\_;}$)
$\triangleright (K; \lambda x_1.\, \mathbf{app}_2\ v_1\ x_2) \diamond \mathbf{ev}\ e_2 : \sigma$	By rule $\mathsf{vs\_cpm}$

In the representation we decompose the first argument as before. This now gives as also `Q1` which is the derivation of v_1 *Value*.

```
vps_app1 : vps (vs_# VK (vi_app1 Q1 P2 P1)) (st_app1)
             (vs_# (vk_; VK ([x2] [q2:value x2] [p2:of x2 T2]
                                 vi_app2 q2 Q1 p2 P1))
                (vi_ev P2)).
```

Case: $\mathcal{C}$ is $\mathsf{st\_app_2}$:

$$K \diamond \mathbf{app}_2\ (\mathbf{lam}\ x.\, e_1')\ v_2 \Longrightarrow K \diamond \mathbf{ev}\ ([v_2/x]e_1').$$

$\triangleright (K \diamond \mathbf{app}_2\ (\mathbf{lam}\ x.\, e_1')\ v_2) : \sigma$	Assumption
$\triangleright K : \tau_1 \Rightarrow \sigma$ and	
$\cdot;\cdot \triangleright \mathbf{app}_2\ (\mathbf{lam}\ x.\, e_1')\ v_2) : \tau_1$ for some τ_1	By inversion
$\cdot \triangleright (\mathbf{lam}\ x.\, e_1') : \tau_2 \rightarrow \tau_1$ and	
$\cdot \triangleright v_2 : \tau_2$ for some τ_2 and	
$\cdot \triangleright (\mathbf{lam}\ x.\, e_1')\ \mathit{Value}$ and	
$\cdot \triangleright v_2\ \mathit{Value}$	By inversion
$x{:}\tau_2 \triangleright e_1' : \tau_1$	By inversion
$\cdot \triangleright [v_2/x]e_1' : \tau_1$	By substitution property (2.4)
$\cdot;\cdot \triangleright \mathbf{ev}\ ([v_2/x]e_1') : \tau_1$	By rule ($\mathsf{vi\_ev}$)
$\triangleright K \diamond \mathbf{ev}\ ([v_2/x]e_1') : \sigma$	By rule ($\mathsf{vs\_cpm}$)

By applying inversion as above we obtained

$$P_1' :: (x{:}\tau_2 \triangleright e_1' : \tau_1)$$

which is represented by the variable

```
P1' : {x:exp} of x T2 -> of (E1' x) T1.
```

The appeal to the substitution principle is implemented by applying this function to the representation of v_2 and the typing derivation $\mathcal{P}_2 :: (\cdot \triangleright v_2 : \tau_2)$. Note that we do not need the derivation $\mathcal{Q}_2$ which is evidence that v_2 is a value.

```
vps_app2 : vps (vs_# VK (vi_app2 Q2 (val_lam) P2 (tp_lam P1')))
            (st_app2) (vs_# VK (vi_ev (P1' V2 P2))).
```

□

The multi-step type preservation theorem is a direct consequence of the one-step preservation.

Theorem 6.27 (Multi-Step Type Preservation) *If* $\triangleright S : \sigma$ *and* $S \stackrel{*}{\Longrightarrow} S'$ *then* $\triangleright S' : \sigma$

Proof: By straightforward induction on the structure of the derivation $\mathcal{C}^* :: (S \stackrel{*}{\Longrightarrow} S')$. We only show the implementation.

```
vps* : valids S T -> S =>* S' -> valids S' T -> type.
%mode vps* +VS +C* -VS'.

vps*_stop : vps* VS (stop) VS.
vps*_<< : vps* VS (C2* << C1) VS2
           <- vps VS C1 VS1
           <- vps* VS1 C2* VS2.
```

□

Finally, we come to the progress theorem: we can make a transition from every state that is not a final state. Recall that the only final states are of the form **answer** v.

Theorem 6.28 (Progress) *If* $\triangleright (K \diamond i) : \sigma$ *then there is a state* S' *such that* $K \diamond i \Longrightarrow S'$.

Proof: We know by inversion that

$\triangleright K : \tau \Rightarrow \sigma$ and
$\triangleright i : \tau$ for some τ.

We apply case analysis on i. In each case we can either directly make a transition, or we need to apply several inversions on an available typing or value derivation until each subcase can be seen to be impossible or a transition rule applies. For a **return** instruction, we also need to distinguish cases on the shape of K. We show only a few cases.

Case: $i = \mathbf{ev}\ (e_1\ e_2)$. Then $\mathsf{st\_app}$ applies.

Case: $i = \mathbf{app}_1\ v_1\ e_2$. Then $\mathsf{st\_app}_1$ applies.

Case: $i = \mathbf{app}_2\ v_1\ v_2$. Then

$\cdot \triangleright v_1\ \ \mathit{Value}$ and
$\cdot \triangleright v_2\ \ \mathit{Value}$ and
$\cdot \triangleright v_1 : \tau_2 \to \tau$ and
$\cdot \triangleright v_2\ : \tau_2$ By inversion

Now we distinguish subcases on $\cdot \triangleright v_2\ \ \mathit{Value}$

Subcase: $v_1 = \mathbf{z}$. This is impossible, since there is no rule to conclude $\cdot \triangleright \mathbf{z} : \tau_2 \to \tau$.

Subcase: $v_1 = \mathbf{s}\ v_1'$. This is impossible, since there is no rule to conclude $\cdot \triangleright \mathbf{s}\ v_1' : \tau_2 \to \tau$.

Subcase: $v_1 = \langle v_1', v_1'' \rangle$. This is impossible, since there is not rule to conclude $\cdot \triangleright \langle v_1', v_1'' \rangle : \tau_2 \to \tau$

Subcase: $v_1 = \mathbf{lam}x.\ e_1'$. Then $\mathsf{st\_app}_2$ applies.

The implementation is very similar to progress, except that we don't need to construct the resulting typing derivation. Note that impossible cases are not represented. Also, for simplicity of implementation, we distinguish the cases on the typing derivation rather than the instruction, which is possible since the typing judgment is syntax-directed. We show only the three cases from above.

```
pgs : valids S T -> S => S' -> type.
%mode pgs +VS -C.

pgs_app : pgs (vs_# VK (vi_ev (tp_app P2 P1))) (st_app).

pgs_app1 : pgs (vs_# VK (vi_app1 Q1 P2 P1)) (st_app1).

pgs_app2 : pgs (vs_# VK (vi_app2 Q2 (val_lam) P2 (tp_lam P1')))
               (st_app2).
```

Note that some applications of inversion may be redundant for the sake of uniformity. For example, we could have replaced the last clause by

```
pgs_app2' : pgs (vs_# VK (vi_app2 Q2 (val_lam) P2 P1))
                (st_app2).
```

Nonetheless, the inversion on the value derivation is necessary, and

```
pgs_app2'' : pgs (vs_# VK (vi_app2 Q2 Q1 P2 P1))
                 (st_app2).
```

would be incorrect as a proof case because it is not apparent from the first argument that $\mathsf{st\_app}_2$ indeed applies. □

6.7 Contextual Semantics

[*This section discusses a contextual semantics as an alternative small-step machine to the CPM machine. This still has to be revised from an older version.*]

6.8 Exercises

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

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

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 **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

$$\text{Environments} \quad \eta \quad ::= \quad \cdot \mid \eta, W \mid \eta * F$$

The evaluation rules fev_1+, fev_↑+, and fev_fix on page 161 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 173.

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.

Exercise 6.14 Types play an important role in compilation, which is not reflected in the some of the development of this chapter. Ideally, we would like to take advantage of type information as much as possible in order to produce more compact and more efficient code. This is most easily achieved if the type information is embedded directly in expressions (see Section **??**), but at the very least, we would expect that types can be assigned to intermediate expressions in the compiler.

1. Define typing judgments for de Bruijn expressions, environments, and values for the language of Section 6.2. You may assume that values are always closed.

2. Prove type preservation for your typing judgment and the operational semantics for de Bruijn expressions.

3. Prove type preservation under compilation, that is, well-typed Mini-ML expressions are mapped to well-typed de Bruijn expressions under the translation of Section 6.2.

4. What is the converse of type preservation under compilation. Does your typing judgment satisfy it?

5. Implement the judgments above in Elf.

6. Implement the proofs above in Elf.

Exercise 6.15 As in Exercise 6.14:

1. Define a typing judgment for evaluation contexts. It should only hold for valid evaluation contexts.

2. Prove that splitting a well-typed expression which is not a value always succeeds and produces a unique context and redex.

3. Prove that splitting a well-typed expression results in a valid evaluation context and valid redex.

4. Prove the correctness of contextual evaluation with respect to the natural semantics for Mini-ML.

5. Implement the judgments above in Elf. Evaluation contexts should be represented as functions from expressions to expressions satisfying an additional judgment.

6. Implement the proofs above in Elf.

Exercise 6.16 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 6.17 Carry out the alternative proof of completeness of the continuation machine sketched on page 189. Implement the proof and all necessary lemmas in Elf.

Exercise 6.18 Do the equivalence proof in Lemma 6.22 and the alternative in Exercise 6.17 define the same relation between derivations? If so, exhibit the bijection in the form of a higher-level judgment relating the Elf implementations. Be careful to write out necessary lemmas regarding concatenation. You may restrict yourself to functional abstraction, application, and the necessary computation rules.

Exercise 6.19 Not every valid state of the CPM machine (according to the typing judgments in Section 6.6) can be reached by a computation starting from some initial state of the form $\mathbf{init} \diamond \mathbf{ev}\ e$ where $\cdot \vdash e : \tau$.

1. Exhibit a valid, but unreachable state.
2. Modify the validity judgments so that every valid machine state can in fact be reached from some initial state.
3. Prove this property.
4. Implement your proof in Elf.

Bibliography

[ACCL91] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, October 1991.

[AINP88] Peter B. Andrews, Sunil Issar, Daniel Nesmith, and Frank Pfenning. The TPS theorem proving system. In Ewing Lusk and Russ Overbeek, editors, *9th International Conference on Automated Deduction*, pages 760–761, Argonne, Illinois, May 1988. Springer-Verlag LNCS 310. System abstract.

[All75] William Allingham. *In Fairy Land.* Longmans, Green, and Co., London, England, 1875.

[CCM87] Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The categorical abstract machine. *Science of Computer Programming*, 8, May 1987.

[CDDK86] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. In *Proceedings of the 1986 Conference on LISP and Functional Programming*, pages 13–27. ACM Press, 1986.

[CF58] H. B. Curry and R. Feys. *Combinatory Logic.* North-Holland, Amsterdam, 1958.

[Chu32] A. Church. A set of postulates for the foundation of logic I. *Annals of Mathematics*, 33:346–366, 1932.

[Chu33] A. Church. A set of postulates for the foundation of logic II. *Annals of Mathematics*, 34:839–864, 1933.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Chu41] Alonzo Church. *The Calculi of Lambda-Conversion.* Princeton University Press, Princeton, New Jersey, 1941.

[Coq91] Thierry Coquand. An algorithm for testing conversion in type theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 255–279. Cambridge University Press, 1991.

[Cur34] H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences, U.S.A.*, 20:584–590, 1934.

[dB68] N.G. de Bruijn. The mathematical language AUTOMATH, its usage, and some of its extensions. In M. Laudet, editor, *Proceedings of the Symposium on Automatic Demonstration*, pages 29–61, Versailles, France, December 1968. Springer-Verlag LNM 125.

[dB72] N. G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[DFH+93] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. The Coq proof assistant user's guide. Rapport Techniques 154, INRIA, Rocquencourt, France, 1993. Version 5.8.

[DM82] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Conference Record of the 9th ACM Symposium on Principles of Programming Languages (POPL'82)*, pages 207–212. ACM Press, 1982.

[Dow93] Gilles Dowek. The undecidability of typability in the lambda-pi-calculus. In M. Bezem and J.F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 139–145, Utrecht, The Netherlands, March 1993. Springer-Verlag LNCS 664.

[DP91] Scott Dietzen and Frank Pfenning. A declarative alternative to assert in logic programming. In Vijay Saraswat and Kazunori Ueda, editors, *International Logic Programming Symposium*, pages 372–386. MIT Press, October 1991.

[Ell89] Conal Elliott. Higher-order unification with dependent types. In N. Dershowitz, editor, *Rewriting Techniques and Applications*, pages 121–136, Chapel Hill, North Carolina, April 1989. Springer-Verlag LNCS 355.

[Ell90] Conal M. Elliott. *Extensions and Applications of Higher-Order Unification*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1990. Available as Technical Report CMU-CS-90-134.

[FP91] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the SIGPLAN '91 Symposium on Language Design and Implementation*, pages 268–277, Toronto, Ontario, June 1991. ACM Press.

[Gar92] Philippa Gardner. *Representing Logics in Type Theory.* PhD thesis, University of Edinburgh, July 1992. Available as Technical Report CST-93-92.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.

[Geu92] Herman Geuvers. The Church-Rosser property for $\beta\eta$-reduction in typed λ-calculi. In A. Scedrov, editor, *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 453–460, Santa Cruz, California, June 1992.

[Gol81] Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.

[GS84] Ferenc Gécseg and Magnus Steinby. *Tree Automata.* Akadémiai Kiadó, Budapest, 1984.

[Gun92] Carl A. Gunter. *Semantics of Programming Languages.* MIT Press, Cambridge, Massachusetts, 1992.

[Han91] John J. Hannan. *Investigating a Proof-Theoretic Meta-Language for Functional Programs.* PhD thesis, University of Pennsylvania, January 1991. Available as Technical Report MS-CIS-91-09.

[Han93] John Hannan. Extended natural semantics. *Journal of Functional Programming*, 3(2):123–152, April 1993.

[Har90] Robert Harper. Systems of polymorphic type assignment in LF. Technical Report CMU-CS-90-144, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1990.

[HB34] David Hilbert and Paul Bernays. *Grundlagen der Mathematik.* Springer-Verlag, Berlin, 1934.

[Her30] Jacques Herbrand. Recherches sur la théorie de la démonstration. *Travaux de la Société des Sciences et de Lettres de Varsovic*, 33, 1930.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[HM89] John Hannan and Dale Miller. A meta-logic for functional programming. In H. Abramson and M. Rogers, editors, *Meta-Programming in Logic Programming*, chapter 24, pages 453–476. MIT Press, 1989.

[HM90] John Hannan and Dale Miller. From operational semantics to abstract machines: Preliminary results. In M. Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 323–332, Nice, France, 1990.

[How80] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980. Hitherto unpublished note of 1969, rearranged, corrected, and annotated by Howard.

[HP92] John Hannan and Frank Pfenning. Compiler verification in LF. In Andre Scedrov, editor, *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418, Santa Cruz, California, June 1992.

[HP00] Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. Technical Report CMU-CS-00-148, Department of Computer Science, Carnegie Mellon University, July 2000.

[Hue73] Gérard Huet. The undecidability of unification in third order logic. *Information and Control*, 22(3):257–267, 1973.

[Hue75] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, Germany, January 1987. ACM Press.

[Kah87] Gilles Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer-Verlag LNCS 247, 1987.

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

[Lia95] Chuck Liang. *Object-Level Substitutions, Unification and Generalization in Meta Logic*. PhD thesis, University of Pennsylvania, December 1995.

[Mai92] H.G. Mairson. Quantifier elimination and parametric polymorphism in programming languages. *Journal of Functional Programming*, 2(2):213–226, April 1992.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal Of Computer And System Sciences*, 17:348–375, August 1978.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[ML85] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. Technical Report 2, Scuola di Specializzazione in Logica Matematica, Dipartimento di Matematica, Università di Siena, 1985.

[ML96] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996.

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[New65] Allen Newell. Limitations of the current stock of ideas about problem solving. In A. Kent and O. E. Taulbee, editors, *Electronic Information Handling*, pages 195–208, Washington, D.C., 1965. Spartan Books.

[NGdV94] R.P. Nederpelt, J.H. Geuvers, and R.C. de Vrijer, editors. *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1994.

[NM98] Gopalan Nadathur and Dale Miller. Higher-order logic programming. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, chapter 8. Oxford University Press, 1998.

[NM99] Gopalan Nadathur and Dustin J. Mitchell. System description: Teyjus—a compiler and abstract machine based implementation of lambda Prolog. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 287–291, Trento, Italy, July 1999. Springer-Verlag LNCS.

[Pau86] Lawrence C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer-Verlag LNCS 828, 1994.

[Pfe91a] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[Pfe91b] Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, The Netherlands, July 1991.

[Pfe92] Frank Pfenning, editor. *Types in Logic Programming*. MIT Press, Cambridge, Massachusetts, 1992.

[Pfe93] Frank Pfenning. Refinement types for logical frameworks. In Herman Geuvers, editor, *Informal Proceedings of the Workshop on Types for Proofs and Programs*, pages 285–299, Nijmegen, The Netherlands, May 1993.

[Pfe94] Frank Pfenning. Elf: A meta-language for deductive systems. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 811–815, Nancy, France, June 1994. Springer-Verlag LNAI 814. System abstract.

[Plo75] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

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

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[PM93] Christine Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. In M. Bezem and J.F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 328–345, Utrecht, The Netherlands, March 1993. Springer-Verlag LNCS 664.

[Pra65] Dag Prawitz. *Natural Deduction*. Almquist & Wiksell, Stockholm, 1965.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

[PW90] David Pym and Lincoln Wallen. Investigations into proof-search in a system of first-order dependent function types. In M.E. Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction*, pages 236–250, Kaiserslautern, Germany, July 1990. Springer-Verlag LNCS 449.

[PW91] David Pym and Lincoln A. Wallen. Proof search in the $\lambda\Pi$-calculus. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 309–340. Cambridge University Press, 1991.

[Pym90] David Pym. *Proofs, Search and Computation in General Logic.* PhD thesis, University of Edinburgh, 1990. Available as CST-69-90, also published as ECS-LFCS-90-125.

[Pym92] David Pym. A unification algorithm for the $\lambda\Pi$-calculus. *International Journal of Foundations of Computer Science*, 3(3):333–378, September 1992.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[RP96] Ekkehard Rohwedder and Frank Pfenning. Mode and termination checking for higher-order logic programs. In Hanne Riis Nielson, editor, *Proceedings of the European Symposium on Programming*, pages 296–310, Linköping, Sweden, April 1996. Springer-Verlag LNCS 1058.

[Sal90] Anne Salvesen. The Church-Rosser theorem for LF with $\beta\eta$-reduction. Unpublished notes to a talk given at the First Workshop on Logical Frameworks in Antibes, France, May 1990.

[Sch00] Carsten Schürmann. *Automating the Meta Theory of Deductive Systems.* PhD thesis, Department of Computer Science, Carnegie Mellon University, August 2000. Available as Technical Report CMU-CS-00-146.

[SH84] Peter Schroeder-Heister. A natural extension of natural deduction. *The Journal of Symbolic Logic*, 49(4):1284–1300, December 1984.

[Twe98] Twelf home page. Available at `http://www.cs.cmu.edu/~twelf`, 1998.