Lecture Notes on Call-by-Push-Value

15-816: Substructural Logics
Frank Pfenning

Lecture 21
November 10, 2016

In this lecture we first present natural deduction in its pure form and then *polarize* it into negative and positive proposition in order to make it more directly suitable as the basis for a functional programming language. As we may surmise from earlier lectures, positive propositions type values, while negative propositions type computations. The resulting *call-by-push-value* system [Lev01] can compositionally embed both call-by-value and call-by-name.

1 Natural Deduction

Natural deduction was first introduced by Gentzen [Gen35] as a formalization of ordinary mathematical reasoning with connectives. In contrast, he considered his sequent calculus a technical device for proving consistency via his Hauptsatz. As such one might consider natural deduction as the most fundamental means to define the logical connectives. We will not dwell on this, but the key aspect of *harmony* that we formulated on the sequent calculus was first expressed in natural deduction [Dum91, ML83].

Instead of right and left rules that define the connectives, we have *introduction* and *elimination* rules. Roughly speaking, an introduction rule corresponds to a right rule: it shows how to prove a proposition. An elimination rule corresponds to a left rule and shows how to use a proposition. But rather than decomposing the proposition from the conclusion to the premise, the elimination rule should be read from the premise to the conclusion.

From a judgmental point of view, natural deduction is based on a single basic judgment $A\ \mathit{true}$ and a hypothetical judgment

$$A_1\ \mathit{true} \ldots A_n\ \mathit{true} \vdash A\ \mathit{true}$$

while the sequent calculus uses *two* judgments, A *ante* only as antecedents, and A *succ* only as a succedent

$$A_1\ \mathit{ante} \ldots A_n\ \mathit{ante} \vdash A\ \mathit{succ}$$

Consequently, in natural deduction there is only a hypothesis rule and no cut rule, although we have an admissible rule of substitution from the very nature of hypothetical judgments. Following Levy, we give here the structural form of all judgments, not the ordered or linear ones, but they of course exist as well.

$$\frac{}{\Gamma, A \vdash A}\ \mathsf{hyp} \qquad \frac{\Gamma \vdash A \quad \Gamma, A \vdash C}{\Gamma \vdash C}\ \mathsf{subst}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ {\to}I \qquad \frac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B}\ {\to}E$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \,\&\, B}\ \&I \qquad \frac{\Gamma \vdash A \,\&\, B}{\Gamma \vdash A}\ \&E_1 \qquad \frac{\Gamma \vdash A \,\&\, B}{\Gamma \vdash B}\ \&E_2$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \times B}\ {\times}I \qquad \frac{\Gamma \vdash A \times B \quad \Gamma, A, B \vdash C}{\Gamma \vdash C}\ {\times}E$$

$$\frac{}{\Gamma \vdash \mathbf{1}}\ \mathbf{1}I \qquad \frac{\Gamma \vdash \mathbf{1} \quad \Gamma \vdash C}{\Gamma \vdash C}\ \mathbf{1}E$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A + B}\ {+}I_1 \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A + B}\ {+}I_2 \qquad \frac{\Gamma \vdash A + B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}\ {+}E$$

Since the operational reading will differ, we have two forms of conjunction with the same introduction rule but different elimination rules. We see the characteristic form of eliminations for the negative connectives which turn a proof of a conclusion into hypotheses of its parts. That leads to some anomalies, like the $\mathbf{1}E$ rule in which the conclusion is equal to the second premise. Again, this can have some operational meaning so we include it here despite it apparent logical redundancy.

The name *substitution* expresses that we substitute the proof of A for uses of A in the proof of C. Martin-Löf [ML83] treats this as the very definition of what a hypothetical judgment means. When we come to the functional programming language, this will correspond to substitution of terms which is the engine that drives computation. This is very different from the sequent calculus, where cut reduction proceeds in much smaller steps.

2 Polarizing Natural Deduction

While polarization of propositions is entailed by their very nature and there is no choice, it is not immediately obvious how we should polarize the hypothetical judgments. With a view towards our goal of functional programming and the nature of implication as having the form $A^+ \to B^-$ we decide that *hypotheses should always be positive*, $\Gamma^+ \vdash C$. The conclusion C sometimes must be negative (for example, the $\to I$ and $\& I$) and sometimes positive (for example, $\times I$, $\mathbf{1}I$, and $\oplus I_i$). But let's recall the structural polarized propositions. Since the positive propositions will correspond to values we also call them *value types*, while negative propositions are *computation types*.

$$\begin{array}{lrcl} \text{Computation Types} & A^- & ::= & A^+ \to B^- \mid A^- \,\&\, B^- \mid \uparrow A^+ \\ \text{Value Types} & A^+ & ::= & A^+ \times B^+ \mid \mathbf{1} \mid A^+ + B^+ \mid \downarrow A^- \end{array}$$

The rules can be polarized straightforwardly, after some basic decisions. Besides requiring the hypotheses to be entirely positive (that is, ranging over *values*), we allow the elimination rules only when the conclusion C is

negative since elimination corresponds to some computation, not a value.

$$\frac{\Gamma \vdash A^+ \quad \Gamma, A^+ \vdash C^-}{\Gamma \vdash C^-}\ \mathsf{subst}^- \qquad \frac{\Gamma \vdash A^+ \quad \Gamma, A^+ \vdash C^+}{\Gamma \vdash C^+}\ \mathsf{subst}^+$$

$$\frac{}{\Gamma, A^+ \vdash A^+}\ \mathsf{hyp}$$

$$\frac{\Gamma, A^+ \vdash B^-}{\Gamma \vdash A^+ \to B^-} \to I \qquad \frac{\Gamma \vdash A^+ \to B^- \quad \Gamma \vdash A^+}{\Gamma \vdash B^-} \to E$$

$$\frac{\Gamma \vdash A^- \quad \Gamma \vdash B^-}{\Gamma \vdash A^- \,\&\, B^-} \,\&I \qquad \frac{\Gamma \vdash A^- \,\&\, B^-}{\Gamma \vdash A^-} \,\&E_1 \qquad \frac{\Gamma \vdash A^- \,\&\, B^-}{\Gamma \vdash B^-} \,\&E_2$$

$$\frac{\Gamma \vdash A^+ \quad \Gamma \vdash B^+}{\Gamma \vdash A^+ \times B^+} \times I \qquad \frac{\Gamma \vdash A^+ \times B^+ \quad \Gamma, A^+, B^+ \vdash C^-}{\Gamma \vdash C^-} \times E$$

$$\frac{}{\Gamma \vdash \mathbf{1}}\ \mathbf{1}I \qquad \frac{\Gamma \vdash \mathbf{1} \quad \Gamma \vdash C^-}{\Gamma \vdash C^-}\ \mathbf{1}E$$

$$\frac{\Gamma \vdash A^+}{\Gamma \vdash A^+ + B^+} +I_1 \qquad \frac{\Gamma \vdash B^+}{\Gamma \vdash A^+ + B^+} +I_2$$

$$\frac{\Gamma \vdash A^+ + B^+ \quad \Gamma, A^+ \vdash C^- \quad \Gamma, B^+ \vdash C^-}{\Gamma \vdash C^-} +E$$

The interesting part now pertains to the shifts. Using our restrictions on contexts and conclusion, we derive the following rules. The form of the elimination rule, either direct or with a side formula C, depends on the polarity of the proposition that is exposed, not the shift we eliminate.

$$\frac{\Gamma \vdash A^+}{\Gamma \vdash \uparrow A^+} \uparrow I \qquad \frac{\Gamma \vdash \uparrow A^+ \quad \Gamma, A^+ \vdash C^-}{\Gamma \vdash C^-} \uparrow E$$

$$\frac{\Gamma \vdash A^-}{\Gamma \vdash \downarrow A^-} \downarrow I \qquad \frac{\Gamma \vdash \downarrow A^-}{\Gamma \vdash A^-} \downarrow E$$

3 Term Assignments

We now assign program terms to the various typing rules. We have computations M of type A^- and values V of type A^+. Note that variables are always of positive type and therefore stand for values. This does *not* imply a call-by-value strategy, as we will see in the next lecture. First, the hypothesis rule is straightforward.

$$\frac{}{\Gamma, x{:}A^+ \vdash x : A^+}\ \mathsf{hyp}$$

The substitution principles, when annotated with proof terms, allow us to apply well-typed substitutions. Note that they are admissible rather than primitive typing rules. They tell us the result of the substitution is well-typed whenever we substitute a value of type A^+ for a variable x of type A^+. This is the only substitution we perform, since all variables range over values.

$$\frac{\Gamma \vdash V : A^+ \quad \Gamma, x{:}A^+ \vdash M : C^-}{\Gamma \vdash [V/x]M : C^-}\ \mathsf{subst}^- \qquad \frac{\Gamma \vdash V : A^+ \quad \Gamma, x{:}A^+ \vdash W : C^+}{\Gamma \vdash [V/x]W : C^+}\ \mathsf{subst}^+$$

We now complete the picture by giving all the introduction rules for positive propositions, omitting only the shift for now.

$$\frac{\Gamma \vdash V : A^+ \quad \Gamma \vdash W : B^+}{\Gamma \vdash (V, W) : A^+ \times B^+}\ \times I$$

$$\frac{\Gamma \vdash V : A^+ \times B^+ \quad \Gamma, x{:}A^+, y{:}B^+ \vdash N : C^-}{\Gamma \vdash \mathsf{match}\, V\, \mathsf{as}\, (x, y) \Rightarrow N : C^-}\ \times E$$

$$\frac{}{\Gamma \vdash ( ) : \mathbf{1}}\ \mathbf{1}I \qquad \frac{\Gamma \vdash V : \mathbf{1} \quad \Gamma \vdash N : C^-}{\Gamma \vdash \mathsf{match}\, V\, \mathsf{as}\, ( ) \Rightarrow N : C^-}\ \mathbf{1}E$$

$$\frac{\Gamma \vdash V : A^+}{\Gamma \vdash \mathsf{inl}(V) : A^+ + B^+}\ +I_1 \qquad \frac{\Gamma \vdash W : B^+}{\Gamma \vdash \mathsf{inr}(W) : A^+ + B^+}\ +I_2$$

$$\frac{\Gamma \vdash V : A^+ + B^+ \quad \Gamma, x{:}A^+ \vdash M : C^- \quad \Gamma, y{:}B^+ \vdash N : C^-}{\Gamma \vdash \mathsf{match}\, V\, \mathsf{as}\, (\mathsf{inl}(x) \Rightarrow M \mid \mathsf{inr}(y) \Rightarrow N) : C^-}\ +E$$

For functional programming, just as for session types, it is convenient to

generalize binary disjunction $A + B$ into a labeled sum $+\{l : A_l\}_{l \in L}$.

$$\frac{\Gamma \vdash V : A_k \quad (k \in L)}{\Gamma \vdash k(V) : +\{l : A_l\}_{l \in L}} \, +I_k$$

$$\frac{\Gamma \vdash V : +\{l : A_l\}_{l \in L} \quad \Gamma, x{:}A_l \vdash M_l : C^- \quad (\forall l \in L)}{\Gamma \vdash \mathsf{match}\, V\, \mathsf{as}\, (l(x) \Rightarrow M_l)_{l \in L} : C^-} \, +E$$

At this point we only need the shift to complete the value types. The positive type is $\downarrow A^-$. Since A^- represents a computation, the value of type $\downarrow A^-$ represents a suspended computation. Following tradition, we call this a *thunk*. The elimination rule forces a thunk to be evaluated.

$$\frac{\Gamma \vdash M : A^-}{\Gamma \vdash \mathsf{thunk}\, M : \downarrow A^-} \downarrow I \qquad \frac{\Gamma \vdash V : \downarrow A^-}{\Gamma \vdash \mathsf{force}\, V : A^-} \downarrow E$$

Let's take stock. At this point, we have the elimination rules for values as computations, and also the full set of values. Still missing are the computations for negative types. As we start on the negative types, we generalize $A^- \,\&\, B^-$ to the labeled version $\&\{l : A_l\}_{l \in L}$.

$$\frac{\Gamma, x{:}A^+ \vdash M : B^-}{\Gamma \vdash \lambda x.\, M : A^+ \to B^-} \to I \qquad \frac{\Gamma \vdash M : A^+ \to B^- \quad \Gamma \vdash V : A^+}{\Gamma \vdash M\, V : B^-} \to E$$

$$\frac{\Gamma \vdash M_l : A_l^- \quad (\forall l \in L)}{\Gamma \vdash \{l \Rightarrow M_l\}_{l \in L} : \&\{l : A_l^-\}_{l \in L}} \,\&I \qquad \frac{\Gamma \vdash M : \&\{l : A_l^-\}_{l \in L}}{\Gamma \vdash M.l_k : A_k^-} \,\&E_k$$

The computation type $\uparrow A^+$ just embeds a value, which we write as $\mathsf{return}\ V$. The elimination for decomposes this and substitutes the value for a bound variable.

$$\frac{\Gamma \vdash V : A^+}{\Gamma \vdash \mathsf{return}\, V : \uparrow A^+} \uparrow I \qquad \frac{\Gamma \vdash M : \uparrow A^+ \quad \Gamma, x{:}A^+ \vdash N : C^-}{\Gamma \vdash \mathsf{let\ val}\, x = M\, \mathsf{in}\, N : C^-} \uparrow E$$

In summary, including the type on which each construct operates, ei-

ther by introduction or elimination:

$$
\begin{array}{llll}
\text{Computations} & M & ::= & \mathsf{match}\, V\, \mathsf{as}\, (x,y) \Rightarrow M & A^+ \times B^+ \\
& & \mid & \mathsf{match}\, V\, \mathsf{as}\, (\,) \Rightarrow M & \mathbf{1} \\
& & \mid & \mathsf{match}\, V\, \mathsf{as}\, (l(x) \Rightarrow M_l)_{l\in L} & +\{l : A_l^+\}_{l\in L} \\
& & \mid & \mathsf{force}\, V & \downarrow A^- \\
& & \mid & \lambda x.M \mid M\, V & A^+ \to B^- \\
& & \mid & \{l \Rightarrow M_l\}_{l\in L} \mid M.k & \&\{l : A_l^-\}_{l\in L} \\
& & \mid & \mathsf{return}\, V \mid \mathsf{let\ val}\, x = M\, \mathsf{in}\, N & \uparrow A^+ \\
\text{Values} & V & ::= & x & \\
& & \mid & (V,W) & A^+ \times B^+ \\
& & \mid & (\,) & \mathbf{1} \\
& & \mid & l(V) & +\{l : A_l^+\}_{l\in L} \\
& & \mid & \mathsf{thunk}\, M & \downarrow A^-
\end{array}
$$

4 Local Reduction

The analogue of a cut reduction is a *local reduction*. It will call upon *substitution* $[V/x]M$. A local reduction arises when an introduction of a proposition is immediately followed by its elimination.

$$
\begin{array}{lcl}
\mathsf{match}\, (V,W)\, \mathsf{as}\, (x,y) \Rightarrow M & \longrightarrow & [V/x, W/y]M \\
\mathsf{match}\, (\,)\, \mathsf{as}\, (\,) \Rightarrow M & \longrightarrow & M \\
\mathsf{match}\, k(V)\, \mathsf{as}\, (l(x) \Rightarrow M_l)_{l\in L} & \longrightarrow & [V/x]M_k \\
\mathsf{force}\, (\mathsf{thunk}\, M) & \longrightarrow & M \\
(\lambda x.\, M)\, V & \longrightarrow & [V/x]M \\
\{l \Rightarrow M_l\}_{l\in L}.k & \longrightarrow & M_k \\
\mathsf{let\ val}\, x = \mathsf{return}\, V\, \mathsf{in}\, N & \longrightarrow & [V/x]N
\end{array}
$$

Local reductions can be also found in the operational semantics, which imposes a certain strategy on the reductions which give the call-by-push-value strategy its name.

5 Substructural Operational Semantics

Interestingly, we can specify the substructural operational semantics entirely on ordered logic. It takes the form of a stack machine and uses substitution, although other techniques are certainly possible.

As is typical for functional languages, we use three predicates:

- $\mathsf{eval}(M)$, which means computation M should be evaluated.
- $\mathsf{retn}(T)$, which means we return a *terminal computation* T.
- $\mathsf{cont}(K)$, which specifies continuation K waiting for a returned value.

Globally, we start with $\mathsf{eval}(M)$ for a closed term M and apply ordered inference to eventually obtain $\mathsf{retn}(T)$. Here, T stands for a terminal computation, that is, a computation that does not take any further steps. We have

$$\begin{array}{lrcll} \text{Terminal Computations} & T & ::= & \lambda x.\, M & A^+ \to B^- \\ & & \mid & \{l \Rightarrow M_l\}_{l \in L} & A^- \,\&\, B^- \\ & & \mid & \mathsf{return}\, V & \uparrow A \end{array}$$

During the computation, the configuration will always have one of the forms

$$\begin{array}{l} \mathsf{eval}(M)\ \mathsf{cont}(K_n)\ \ldots\ \mathsf{cont}(K_1) \\ \mathsf{retn}(T)\ \mathsf{cont}(K_n)\ \ldots\ \mathsf{cont}(K_1) \end{array}$$

which means that the continuations form a stack. Rather than pre-specify, we will write up the operational semantics and see which kind of continuations we need. It helps to make us aware how evaluations, returns, and continuations are typed. Note that all computations and values are closed, so no hypotheses are needed in their typing.

$$\frac{\cdot \vdash M : A^-}{\mathsf{eval}(M) : A^-} \qquad \frac{\cdot \vdash T : A^-}{\mathsf{retn}(T) : A^-} \qquad \frac{A^- \vdash K : B^-}{A^- \vdash \mathsf{cont}(K) : B^-}$$

The only interesting part here are the continuation stack frames. They expect a terminal computation on the left and return a terminal computation to the right, to the next stack frame.

The first key inside is that values of positive type do not need to be further evaluated. One can see that simply by looking at the typing rules.

So we have

$$\frac{\mathsf{eval}(\mathsf{match}\ (V, W)\ \mathsf{as}\ (x, y) \Rightarrow M)}{\mathsf{eval}([V/x, W/y]M)}\ \times C$$

$$\frac{\mathsf{eval}(\mathsf{match}\ (\,)\ \mathsf{as}\ (\,) \Rightarrow M)}{\mathsf{eval}(M)}\ \mathbf{1}C$$

$$\frac{\mathsf{eval}(\mathsf{match}\ k(V)\ \mathsf{as}\ (l(x) \Rightarrow M_l)_{l \in L})}{\mathsf{eval}([V/x]M_k)}\ +C$$

$$\frac{\mathsf{eval}(\mathsf{force}\ (\mathsf{thunk}\ M))}{\mathsf{eval}(M)}\ \downarrow C$$

The cases for negative types are more complicated, since they require continuations. Fortunately, there are only 3. We start with functions.

$$\frac{\mathsf{eval}(M\ V)}{\mathsf{eval}(M) \quad \mathsf{cont}(\_\ V)}\ \rightarrow C_1 \qquad \frac{\mathsf{eval}(\lambda x.\, M)}{\mathsf{retn}(\lambda x.\, M)}\ \rightarrow C_2 \qquad \frac{\mathsf{retn}(\lambda x.\, M) \quad \mathsf{cont}(\_\ V)}{\mathsf{eval}([V/x]M)}\ \rightarrow C_3$$

We can see the continuation must accept a function from the left and pass its return value to the right. So we have

$$\frac{\cdot \vdash V : A^+}{A^+ \rightarrow B^- \vdash \mathsf{cont}(\_\ V) : B^-}$$

Now we can appreciate why this scheme is called *call-by-push-value*. In the purely functional fragment (only type $A^+ \rightarrow B^-$) the configuration will have one of the two forms

$$\begin{array}{l} \mathsf{eval}(M)\ \mathsf{cont}(\_\ V_n)\ \ldots\ \mathsf{cont}(\_\ V_1) \\ \mathsf{retn}(T)\ \mathsf{cont}(\_\ V_n)\ \ldots\ \mathsf{cont}(\_\ V_1) \end{array}$$

that is, the continuations form a stack of values, and function application will push a value onto the stack. The reason we can still easily represent call-by-name, by the way, is because $\mathsf{thunk}\, M$ is a possible value. We will see that in the next lecture.

Next, products $\&\{l : A_l\}_{l \in L}$. We see that they correspond to *lazy pairs*, because the components are not evaluated until the value of that compo-

nent is requested.

$$\frac{\mathsf{eval}(M.k)}{\mathsf{eval}(M) \quad \mathsf{cont}(\_.k)}\ \&C_1$$

$$\frac{\mathsf{eval}(\{l \Rightarrow M_l\}_{l \in L})}{\mathsf{retn}(\{l \Rightarrow M_l\}_{l \in L})}\ \&C_2$$

$$\frac{\mathsf{retn}(\{l \Rightarrow M_l\}_l \in L) \quad \mathsf{cont}(\_.k)}{\mathsf{eval}(M_k)}\ \&C_3$$

Again, we can derive the typing of the new form of continuation from the computation rules.

$$\frac{}{\&\{l : A_l\}_{l \in L} \vdash \mathsf{cont}(\_.k) : A_k}$$

There is a strong analogy between the typing of this continuation and the typing of a message in SILL.

Finally, values as they are included in computations.

$$\frac{\mathsf{eval}(\mathsf{let\ val}\, x = M\ \mathsf{in}\ N)}{\mathsf{eval}(M) \quad \mathsf{cont}(\mathsf{let\ val}\, x = \_\ \mathsf{in}\ N)}\ \uparrow C_1$$

$$\frac{\mathsf{eval}(\mathsf{return}\, V)}{\mathsf{retn}(\mathsf{return}\, V)}\ \uparrow C_2$$

$$\frac{\mathsf{retn}(\mathsf{return}\, V) \quad \mathsf{cont}(\mathsf{let\ val}\, x = \_\ \mathsf{in}\ N)}{\mathsf{eval}([V/x]N)}\ \uparrow C_3$$

The requisit typing:

$$\frac{x{:}A^+ \vdash N : C^-}{\uparrow A^+ \vdash \mathsf{cont}(\mathsf{let\ val}\, x = \_\ \mathsf{in}\ N) : C^-}$$

6 Example: A Map Function

As a simple example we consider a higher-order function map that applies a function to each element of a given list to construct a new list. The interesting aspect of this exercise it the polarization.

Lists are entirely positive. Not unexpected, given we already encounter this property in the concurrent setting. Of course, we could have *lazy lists*, in which case the type would be quite different (see Exercise 3).

$$\mathsf{list}\, A^+ = +\{\mathsf{cons} : A^+ \times \mathsf{list}\, A^+, \mathsf{nil} : 1\}$$

The map function has to account for this particular polarization.

$$\mathsf{map} : \downarrow(A^+ \to \uparrow B^+) \to \mathsf{list}\, A^+ \to \uparrow \mathsf{list}\, B^+$$

The we define

$$\begin{array}{l}
\mathsf{map} = \lambda f.\, \lambda l. \\
\quad \mathsf{match}\ l\ \mathsf{as}\ (\ \mathsf{cons}(p) \Rightarrow \mathsf{match}\ p\ \mathsf{as}\ (x, l') \Rightarrow \\
\qquad\qquad \mathsf{let\,val}\ y = (\mathsf{force}\ f)\ x\ \mathsf{in} \\
\qquad\qquad \mathsf{let\,val}\ k = (\mathsf{map}\ f)\ l'\ \mathsf{in} \\
\qquad\qquad\quad \mathsf{return}\ \mathsf{cons}(y, l') \\
\qquad\quad \mid \mathsf{nil}(p) \Rightarrow \mathsf{match}\ p\ \mathsf{as}\ (\,) \Rightarrow \\
\qquad\qquad \mathsf{return}\ \mathsf{nil}(\,))
\end{array}$$

Exercises

Exercise 1 The operational semantics uses substitution of values for variables. We can instead *bind* variables to values using a persistent predicate $\underline{\mathsf{bind}}(x, V)$ in operational semantics.

Rewrite the substructural operational semantics using $\underline{\mathsf{bind}}$ so that only (fresh) parameters are substituted for variables, not arbitrary values.

Exercise 2 We can refactor the syntax using *patterns* which are defined as

$$\text{Patterns} \quad p, q \quad ::= x \mid (p, q) \mid () \mid l(p)$$

where variables x always have type $\downarrow A^-$. Then, there is only a single matching construct

$$\mathsf{match}\ V\ \mathsf{as}\ (p_i \Rightarrow M_i)_i$$

Rewrite the typing rules and the operational semantics using patterns.

Exercise 3 Define *lazy lists* as the negative type

$$\mathsf{list}\, A^- = \uparrow + \{\mathsf{cons} : \downarrow \& \{\mathsf{hd} : A^-, \mathsf{tl} : \mathsf{list}\, A^-\}, \mathsf{nil} : \downarrow \& \{\,\}\}$$

Rewrite the map function to work on lazy lists.

References

[Dum91] Michael Dummett. *The Logical Basis of Metaphysics*. Harvard University Press, Cambridge, Massachusetts, 1991. The William James Lectures, 1976.

[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.

[Lev01] Paul Blain Levy. *Call-by-Push-Value*. PhD thesis, University of London, 2001.

[ML83] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. Notes for three lectures given in Siena, Italy. Published in *Nordic Journal of Philosophical Logic*, 1(1):11-60, 1996, April 1983.