4.6 Recursive Types

The language so far lacks basic data types, such as natural numbers, integers, lists, trees, etc. Moreover, except for finitary ones such as booleans, they are not definable with the mechanism at our disposal so far. At this point we can follow two paths: one is to define each new data type in the same way we defined the logical connectives, that is, by introduction and elimination rules, carefully checking their local soundness and completeness. The other is to enrich the language with a general mechanism for defining such new types. Again, this can be done in different ways, using either *inductive types* which allow us to maintain a clean connection between propositions and types, or *recursive types* which are more general, but break the correspondence to logic. Since we are mostly interested in programming here, we chose the latter path.

Recall that we defined the booleans as $\mathbf{1} \oplus \mathbf{1}$. It is easy to show by the definition of values, that there are exactly two values of this type, to which we can arbitrarily assign true and false. A finite type with n values can be defined as the disjoint sum of n observable singleton types, $\mathbf{1} \oplus \cdots \oplus \mathbf{1}$. The natural numbers would be $\mathbf{1} \oplus \mathbf{1} \oplus \cdots$, except that this type is infinite. We can express it finitely as a recursive type $\mu\alpha.\ \mathbf{1} \oplus \alpha$. Intuitively, the meaning of this type should be invariant under unrolling of the recursion. That is,

$$\begin{array}{rcl} \mathrm{nat} & = & \mu\alpha.\ \mathbf{1} \oplus \alpha \\ & \cong & [(\mu\alpha.\ \mathbf{1} \oplus \alpha)/\alpha]\mathbf{1} \oplus \alpha \\ & = & \mathbf{1} \oplus \mu\alpha.\ \mathbf{1} \oplus \alpha \\ & = & \mathbf{1} \oplus \mathrm{nat} \end{array}$$

which is the expected recursive definition for the type of natural numbers.

In functional languages such as ML or Haskell, recursive type definitions are not directly available, but the results of elaborating syntactically more pleaseant definitions. In addition, recursive type definitions are *generative*, that is, they generate new constructors and types every time they are invoked. This is of great practical value, but the underlying type theory can be seen as simple recursive types combined with a mechanism for generativity. Here, we will only treat the issue of recursive types.

Even though recursive types do not admit a logical interpretation as propositions, we can still define a term calculus using introduction and elimination rules, including local reduction and expansions. In order maintain the property that a term has a unique type, we annotate the introduction constant fold with the recursive type itself.

$$\frac{\Gamma; \Delta \vdash M : [\mu\alpha.\ A/\alpha]A}{\Gamma; \Delta \vdash \mathrm{fold}^{\mu\alpha.\ A} M : \mu\alpha.\ A}\,\mu\mathrm{I} \qquad \frac{\Gamma; \Delta \vdash M : \mu\alpha.\ A}{\Gamma; \Delta \vdash \mathrm{unfold}\, M : [\mu\alpha.\ A/\alpha]A}\,\mu\mathrm{E}$$

The local reduction and expansions, expressed on the terms.

$$\begin{array}{rcl} \mathrm{unfold}\,\mathrm{fold}^{\mu\alpha.\ A}\, M & \longrightarrow_\beta & M \\ M : \mu\alpha.\ A & \longrightarrow_\eta & \mathrm{fold}^{\mu\alpha.\ A}\,(\mathrm{unfold}\, M) \end{array}$$

It is easy to see that uniqess of types and subject reduction remain valid properties (see Exercise 4.11). There are also formulation of recursive types where the term M in the premiss and conclusion is the same, that is, there are no explicit constructor and destructors for recursive types. This leads to more concise programs, but significantly more complicated type-checking (see Exercise 4.12).

We would like recursive types to represent data types. Therefore the values of recursive type must be of the form $\mathrm{fold}^{\mu\alpha.\,A}\, v$ for values v—otherwise data values would not be observable.

$$\frac{M\ \mathit{Value}}{\mathrm{fold}^{\mu\alpha.\,A}\, M\ \mathit{Value}}\,\mu\mathrm{val}$$

$$\frac{M \hookrightarrow v}{\mathrm{fold}^{\mu\alpha.\,A}\, M \hookrightarrow \mathrm{fold}^{\mu\alpha.\,A}\, v}\,\mu\mathrm{Iv} \qquad \frac{M \hookrightarrow \mathrm{fold}^{\mu\alpha.\,A}\, v}{\mathrm{unfold}\, M \hookrightarrow v}\,\mu\mathrm{Ev}$$

In order to write interesting programs simply, it is useful to have a general recursion operator $\mathbf{fix}\, u{:}A.\ M$ at the level of terms. It is not associated with an type constructor and simply unrolls its definition once when executed. In the typing rule we have to be careful: since the number on unrollings generally unpredictable, no linear variables are permitted to occur free in the body of a recursive definition. Moreover, the recursive function itself may be called arbitrarily many times—one of the characteristics of recursion. Therefore its uses are unrestricted.

$$\frac{(\Gamma, u{:}A); \cdot \vdash M : A}{\Gamma; \cdot \vdash \mathbf{fix}\, u{:}A.\ M : A}\,\mathbf{fix}$$

The operator does not introduce any new values, and one new evaluation rules which unrolls the recursion.

$$\frac{[\mathbf{fix}\, u{:}A.\ M/u]M \hookrightarrow v}{\mathbf{fix}\, u{:}A.\ M \hookrightarrow v}\,\mathbf{fix}\mathrm{v}$$

In order to guarantee subject reduction, the type of whole expression, the body M of the fixpoint expression, and the bound variable u must all have the same type A. This is enforced in the typing rules.

We now consider a few examples of recursive types and some example programs.

Natural Numbers.

$$\begin{array}{rcl} \mathrm{nat} & = & \mu\alpha.\, \mathbf{1} \oplus \alpha \\ \mathrm{zero} & : & \mathrm{nat} \\ & = & \mathrm{fold}^{\mathrm{nat}}\,(\mathrm{inl}^{\mathrm{nat}}\,\star) \\ \mathrm{succ} & : & \mathrm{nat} \multimap \mathrm{nat} \\ & = & \hat{\lambda} x{:}\mathrm{nat}.\ \mathrm{fold}^{\mathrm{nat}}\,(\mathrm{inr}^{\mathbf{1}}\, x) \end{array}$$

With this definition, the addition function for natural numbers is linear in both argument.

$$\begin{array}{rcl} \text{plus} & : & \text{nat} \multimap \text{nat} \multimap \text{nat} \\ & = & \mathbf{fix}\, p{:}\text{nat} \multimap \text{nat} \multimap \text{nat}. \\ & & \quad \hat{\lambda} x{:}\text{nat}.\ \hat{\lambda} y{:}\text{nat}.\ \mathbf{case}\ \text{unfold}\, x \\ & & \qquad\qquad\qquad\qquad\qquad \mathbf{of}\ \text{inl} \star \Rightarrow y \\ & & \qquad\qquad\qquad\qquad\qquad\ \mid \text{inr}\, x' \Rightarrow \text{succ} \hat{\ } (p \hat{\ } x' \hat{\ } y) \end{array}$$

It is easy to ascertain that this definition is well-typed: x occurs as the case subject, y in both branches, and x' in the recursive call to p. On the other hand, the natural definition for multiplication is *not* linear, since the second argument is used twice in one branch of the case statement and not at all in the other.

$$\begin{array}{rcl} \text{mult} & : & \text{nat} \multimap \text{nat} \rightarrow \text{nat} \\ & = & \mathbf{fix}\, m{:}\text{nat} \multimap \text{nat} \rightarrow \text{nat} \\ & & \quad \hat{\lambda} x{:}\text{nat}.\ \lambda y{:}\text{nat}.\ \mathbf{case}\ \text{unfold}\, x \\ & & \qquad\qquad\qquad\qquad\qquad \mathbf{of}\ \text{inl} \star \Rightarrow \text{zero} \\ & & \qquad\qquad\qquad\qquad\qquad\ \mid \text{inr}\, x' \Rightarrow \text{plus} \hat{\ } (m \hat{\ } x'\, y) \hat{\ } y \end{array}$$

Interestingly, there is also a linear definition of mult (see Exercise 4.13), but its operational behavior is quite different. This is because we can explicitly copy and delete natural numbers, and even make them available in an unrestricted way.

$$\begin{array}{rcl} \text{copy} & : & \text{nat} \multimap \text{nat} \otimes \text{nat} \\ & = & \mathbf{fix}\, c{:}\text{nat} \multimap \text{nat} \otimes \text{nat} \\ & & \quad \hat{\lambda} x{:}\text{nat}.\ \mathbf{case}\ \text{unfold}\, x \\ & & \qquad\qquad\quad \mathbf{of}\ \text{inl} \star \Rightarrow \text{zero} \otimes \text{zero} \\ & & \qquad\qquad\quad\ \mid \text{inr}\, x' \Rightarrow \mathbf{let}\ x'_1 \otimes x'_2 = c \hat{\ } x'\ \mathbf{in}\ (\text{succ} \hat{\ } x'_1) \otimes (\text{succ} \hat{\ } x'_2) \\ \text{delete} & : & \text{nat} \multimap \mathbf{1} \\ & = & \mathbf{fix}\, d{:}\text{nat} \multimap \mathbf{1} \\ & & \quad \hat{\lambda} x{:}\text{nat}.\ \mathbf{case}\ \text{unfold}\, x \\ & & \qquad\qquad\quad \mathbf{of}\ \text{inl} \star \Rightarrow \mathbf{1} \\ & & \qquad\qquad\quad\ \mid \text{inr}\, x' \Rightarrow \mathbf{let}\ \star = d \hat{\ } x'\ \mathbf{in}\ \mathbf{1} \\ \text{promote} & : & \text{nat} \multimap !\text{nat} \\ & = & \mathbf{fix}\, p{:}\text{nat} \multimap !\text{nat} \\ & & \quad \hat{\lambda} x{:}\text{nat}.\ \mathbf{case}\ \text{unfold}\, x \\ & & \qquad\qquad\quad \mathbf{of}\ \text{inl} \star \Rightarrow !\text{zero} \\ & & \qquad\qquad\quad\ \mid \text{inr}\, x' \Rightarrow \mathbf{let}\ !u' = p \hat{\ } x'\ \mathbf{in}\ !(\text{succ}\, u') \end{array}$$

Lazy Natural Numbers. Lazy natural numbers are a simple example of *lazy data types* which contain unevaluated expressions. Lazy data types are useful in applications with potentially infinite data such as streams. We encode such lazy data types by using the $!A$ type constructor.

$$
\begin{array}{rcl}
\mathrm{lnat} & = & \mu\alpha.\,!(\mathbf{1} \oplus \alpha) \\
\mathrm{lzero} & : & \mathrm{lnat} \\
 & = & \mathrm{fold}^{\mathrm{lnat}}\,!\,(\mathrm{inl}^{\mathrm{lnat}}\,\star) \\
\mathrm{lsucc} & : & \mathrm{lnat} \to \mathrm{lnat} \\
 & = & \lambda u{:}\mathrm{lnat}.\ \mathrm{fold}^{\mathrm{lnat}}\,!\,(\mathrm{inr}^{\mathbf{1}}\,u)
\end{array}
$$

There is also a linear version of successor of type, $\mathrm{lnat} \multimap \mathrm{lnat}$, but it is not as natural since it evaluates its argument just to build another lazy natural number.

$$
\begin{array}{rcl}
\mathrm{lsucc}' & : & \mathrm{lnat} \multimap \mathrm{lnat} \\
 & = & \hat{\lambda} x{:}\mathrm{lnat}.\ \mathbf{let}\ !u = \mathrm{unfold}\,x\ \mathbf{in}\ \mathrm{fold}^{\mathrm{lnat}}\,(!\,\mathrm{inr}^{\mathbf{1}}\,(\mathrm{fold}^{\mathrm{lnat}}\,(!u)))
\end{array}
$$

The "infinite" number number ω can be defined by using the fixpoint operator. We can either use lsucc as defined above, or define it directly.

$$
\begin{array}{rcl}
\omega & : & \mathrm{lnat} \\
 & = & \mathbf{fix}\,u{:}\mathrm{lnat}.\ \mathrm{lsucc}\,u \\
 & \cong & \mathbf{fix}\,u{:}\mathrm{lnat}.\ \mathrm{fold}^{\mathrm{lnat}}\,!\,(\mathrm{inr}^{\mathbf{1}}\,u)
\end{array}
$$

Note that lazy natural numbers are not directly observable (except for the $\mathrm{fold}^{\mathrm{lnat}}$), so we have to decompose and examine the structure of a lazy natural number successor by successor, or we can convert it to an observable natural number (which might not terminate).

$$
\begin{array}{rcl}
\mathrm{toNat} & : & \mathrm{lnat} \multimap \mathrm{nat} \\
 & = & \mathbf{fix}\,t{:}\mathrm{lnat} \multimap \mathrm{nat} \\
 & & \quad \hat{\lambda} x{:}\mathrm{lnat}.\ \mathbf{case}\ \mathrm{unfold}\,x \\
 & & \qquad\qquad \mathbf{of}\ !\mathrm{inl}^{\mathrm{lnat}}\,\star \Rightarrow \mathrm{zero} \\
 & & \qquad\qquad\ \mid\ !\mathrm{inr}^{\mathbf{1}}\,x' \Rightarrow \mathrm{succ}\,\hat{}\,(t\,\hat{}\,x')
\end{array}
$$

Lists. To avoid issues of polymorphism, we define a family of data types list_A for an arbitrary type A.

$$
\begin{array}{rcl}
\mathrm{list}_A & = & \mu\alpha.\,\mathbf{1} \oplus (A \otimes \alpha) \\
\mathrm{nil}_A & : & \mathrm{list}_A \\
 & = & \mathrm{fold}^{\mathrm{list}_A}\,(\mathrm{inl}^{\mathrm{list}_A}\,\star) \\
\mathrm{cons}_A & : & A \otimes \mathrm{list}_A \multimap \mathrm{list}_A \\
 & = & \hat{\lambda} p{:}A \otimes \mathrm{list}_A.\ \mathrm{fold}^{\mathrm{list}_A}\,(\mathrm{inr}^{\mathbf{1}}\,p)
\end{array}
$$

We can easily program simple functions such as append and reverse which are linear in their arguments. We show here reverse; for other examples see Exercise 4.14. we define an auxiliary tail-recursive function rev which moves element from it first argument to its second.

$$
\begin{array}{rcl}
\mathrm{rev}_A & : & \mathrm{list}_A \multimap \mathrm{list}_A \multimap \mathrm{list}_A \\
 & = & \mathbf{fix}\, r{:}\mathrm{list}_A \multimap \mathrm{list}_A \multimap \mathrm{list}_A \\
 & & \quad \hat{\lambda} l{:}\mathrm{list}_A.\ \hat{\lambda} k{:}\mathrm{list}_A. \\
 & & \quad\quad \mathbf{case}\ \mathrm{unfold}\, l \\
 & & \quad\quad\quad \mathbf{of}\ \mathrm{inl}^{A\otimes \mathrm{list}_A} \star \Rightarrow k \\
 & & \quad\quad\quad\ \mid \mathrm{inr}^{\mathbf{1}}\,(x \otimes l') \Rightarrow r\,\hat{}\,l'\,\hat{}\,(\mathrm{cons}_A\,(x \otimes k)) \\
\mathrm{reverse}_A & : & \mathrm{list}_A \multimap \mathrm{list}_A \\
 & = & \hat{\lambda} l{:}\mathrm{list}_A.\ \mathrm{rev}\,\hat{}\,l\,\hat{}\,\mathrm{nil}_A
\end{array}
$$

To make definitions like this a bit easier, we can also define a case for lists, in analogy with the conditional for booleans. It is a family indexed by the type of list elements A and the type of the result of the conditional C.

$$
\begin{array}{rcl}
\mathrm{listCase}_{A,C} & : & \mathrm{list}_A \multimap (C \,\&\, (A \otimes \mathrm{list}_A \multimap C)) \multimap C \\
 & = & \hat{\lambda} l{:}\mathrm{list}_{\mathrm{A}}.\ \hat{\lambda} n{:}C \,\&\, (A \otimes \mathrm{list}_A \multimap C). \\
 & & \quad\quad \mathbf{case}\ \mathrm{unfold} l \\
 & & \quad\quad\quad \mathbf{of}\ \mathrm{inl}^{A\otimes \mathrm{list}_A} \star \Rightarrow \mathrm{fst}\, n \\
 & & \quad\quad\quad\ \mid \mathrm{inr}^{\mathbf{1}}\, p \Rightarrow (\mathrm{snd}\, n)\,\hat{}\,p
\end{array}
$$

Lazy Lists. There are various forms of lazy lists, depending of which evaluation is postponed.

$\mathrm{llist}^1_A = \mu\alpha.\ !(\mathbf{1} \oplus (A \otimes \alpha))$. This is perhaps the canonical lazy lists, in which we can observe neither head nor tail.

$\mathrm{llist}^2_A = \mu\alpha.\ \mathbf{1} \oplus\, !(A \otimes \alpha)$. Here we can observe directly if the list is empty or not, but not the head or tail which remains unevaluated.

$\mathrm{llist}^3_A = \mu\alpha.\ \mathbf{1} \oplus (A \otimes\, !\alpha)$. Here we can observe directly if the list is empty or not, and the head of the list is non-empty. However, we cannot see the tail.

$\mathrm{llist}^4_A = \mu\alpha.\ \mathbf{1} \oplus (!A \otimes \alpha)$. Here the list is always eager, but the elements are lazy. This is the same as $\mathrm{list}_{!A}$.

$\mathrm{llist}^5_A = \mu\alpha.\ \mathbf{1} \oplus (A \& \alpha)$. Here we can see if the list is empty or not, but we can access only either the head or tail of list, but not both.

$\mathrm{infStream}_A = \mu\alpha.\ !(A \otimes \alpha)$. This is the type of infinite streams, that is, lazy lists with no nil constructor.

Functions such as append, map, etc. can also be written for lazy lists (see Exercise 4.15).

Other types, such as trees of various kinds, are also easily represented using similar ideas. However, the recursive types (even without the presence of the fixpoint operator on terms) introduce terms which have no normal form. In the pure, untyped λ-calculus, the classical examples of a term with no normal form is $(\lambda x.\ x\,x)\,(\lambda x.\ x\,x)$ which β-reduces to itself in one step. In the our typed λ-calculus (linear or intuitionistic) this cannot be assigned a type, because x is used as an argument to itself. However, with recursive types (and the fold and unfold constructors) we can give a type to a version of this term which β-reduces to itself in two steps.

$$\begin{array}{rcl} \Omega & = & \mu\alpha.\ \alpha \to \alpha \\ \omega & : & \Omega \to \Omega \\ & = & \lambda x{:}\Omega.\ (\mathrm{unfold}\,x)\,x \end{array}$$

Then

$$\begin{array}{l} \omega\,(\mathrm{fold}^{\Omega}\,\omega) \\ \quad \longrightarrow_{\beta} (\mathrm{unfold}\,(\mathrm{fold}^{\Omega}\,\omega))\,(\mathrm{fold}^{\Omega}\,\omega) \\ \quad \longrightarrow_{\beta} \omega\,(\mathrm{fold}^{\Omega}\,\omega). \end{array}$$

At teach step we applied the only possible β-reduction and therefore the term can have no normal form. An attempt to evaluate this term will also fail, resulting in an infinite regression (see Exercise 4.16).

4.7 Termination

As the example at the end of the previous section shows, unrestricted recursive types destroy the normalization property. This also means it is impossible to give all recursive types a logical interpretation. When we examine the inference rules we notice that recursive types are *impredicative*: the binder $\mu\alpha$ in $\mu\alpha.\ A$ ranges over the whole type. This means in the introduction rule, the type in the premiss $[\mu\alpha.\ A/\alpha]A$ generally will be larger than the type $\mu\alpha.\ A$ in the conclusion. That alone is not responsible for non-termination: there are other type disciplines such as the polymorphic λ-calculus which retain a logical interpretation and termination, yet are impredicative.

In this section we focus on the property that all well-typed terms in the linear λ-calculus *without* recursive types and fixpoint operators evaluate to a value. This is related to the normalization theorem for natural deductions (Theorem 2.19): if $\Gamma; \Delta \vdash A$ then $\Gamma; \Delta \vdash A \ \uparrow$. We proved this by a rather circuitous route: unrestricted natural deductions can be translated to sequent derivations with cut from which we can eliminate cut and translate the result cut-free derivation back to a noraml natural deduction.

Here, we prove directly that every term evaluates using the proof technique of *logical relations* also called *Tait's method*. Because of the importance of this technique, we spend some time motivating its form. Our ultimate goal is to prove:

If $\cdot;\cdot \vdash M : A$ *then* $M \hookrightarrow v$ *for some value* v.

The first natural attempt would be to prove this by induction on the typing derivation. Surprisingly, case for $\multimap$I works, even though we cannot apply the inductive hypothesis, since every linear λ-abstraction immediately evaluates to itself.

In the case for $\multimap$E, however, we find that we cannot complete the proof. Let us examine why.

Case: $$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \cdot;\cdot \vdash M_1 : A_2 \multimap A_1\end{array} \qquad \begin{array}{c}\mathcal{D}_2 \\ \cdot;\cdot \vdash M_2 : A_2\end{array}}{\cdot;\cdot \vdash M_1 \hat{\ } M_2 : A_1} \multimap\mathrm{E}.$$

We can make the following inferences.

$M_1 \hookrightarrow v_1$ for some v_1	By ind. hyp. on $\mathcal{D}_1$
$v_1 = \hat{\lambda} w{:}A_2.\ M_1'$	By type preservation and inversion
$M_2 \hookrightarrow v_2$ for some v_2	By ind. hyp. on $\mathcal{D}_2$

At this point we cannot proceed: we need a derivation of

$$[v_2/w]M_1' \hookrightarrow v \textit{ for some } v$$

to complete the derivation of $M_1\, M_2 \hookrightarrow v$. Unfortunately, the induction hypothesis does not tell us anything about $[v_2/w]M_1'$. Basically, we need to extend it so it makes a statement about the *result* of evaluation ($\hat{\lambda} w{:}A_2.\ M_1'$, in this case).

Sticking to the case of linear application for the moment, we call a term M "good" if it evaluates to a "good" value v. A value v is "good" if it is a function $\hat{\lambda} w{:}A_2.\ M_1'$ and if substituting a "good" value v_2 for w in M_1' results in a "good" term. Note that this is not a proper definition, since to see if v is "good" we may need to substitute any "good" value v_2 into it, possibly including v itself. We can make this definition inductive if we observe that the value v_2 will be of type A_2, while the value v we are testing has type $A_2 \multimap A_1$, and that the resulting term as type A_1. That is, we can fashion a definition which is inductive on the structure of the *type*. Instead of saying "good" we say $M \in \|A\|$ and $v \in |A|$. Still restricting ourselves to linear implication only, we define:

$$\begin{array}{rcl} M \in \|A\| & \text{iff} & M \hookrightarrow v \text{ and } v \in |A| \\ M \in |A_2 \multimap A_1| & \text{iff} & M = \hat{\lambda} w{:}A_2.\ M_1 \text{ and } [v_2/w]M_1 \in \|A_1\| \text{ for any } v_2 \in |A_2| \end{array}$$

From $M \in \|A\|$ we can immediately infer $M \hookrightarrow v$, so when proving that $\cdot;\cdot \vdash M : A$ implies $M \in \|A\|$ we do indeed have a much stronger induction hypothesis.

While the case for application now goes through, the case for linear λ-abstraction fails, since we cannot prove the stronger property for the value.

Case: $$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \cdot;w{:}A_2 \vdash M_1 : A_1\end{array}}{\cdot;\cdot \vdash \hat{\lambda} w{:}A_2.\ M_1 : A_2 \multimap A_1} \multimap\mathrm{I}.$$

Then $\hat{\lambda} w{:}A_2.\ M_1 \hookrightarrow \hat{\lambda} w{:}A_2.\ M_1$ and it remains to show that for every $v_2 \in |A_2|$, $[v_2/w]M_2 \in \|A_1\|$.

This last statement should follow from the induction hypothesis, but presently it is too weak since it only allows for closed terms. The generalization which suggests itself from this case (ignoring the unrestricted context for now) is:

> If $\Delta \vdash M : A$, then for any substitution θ which maps the linear variables $w{:}A$ in Δ to values $v \in |A|$, $[\theta]M \in \|A\|$.

This generalization indeed works after we also account for the unrestricted context. During evaluation we substitute values for linear variables and expressions for unrestricted variables. Therefore, the substitutions we must consider for the induction hypothesis have to behave accordingly.

$$\begin{array}{rrcl} \textit{Unrestricted Substitution} & \eta & ::= & \cdot \mid \eta, M/u \\ \textit{Linear Substitution} & \theta & ::= & \cdot \mid \theta, v/w \end{array}$$

We write $[\eta;\theta]M$ for the simultaneous application of the substitutions η and θ to M. For our purposes here, the values and terms in the substitutions are always closed, but we do not need to postulate this explicitly. Instead, we only deal with substitution satisfying the property necessary for the generalization of the induction hypothesis.

$$\begin{array}{rcl} \theta \in |\Delta| & \text{iff} & [\theta]w \in |A| \text{ for every } w{:}A \text{ in } \Delta \\ \eta \in \|\Gamma\| & \text{iff} & [\eta]u \in \|A\| \text{ for every } u{:}A \text{ in } \Gamma \end{array}$$

We need just one more lemma, namely that values evaluate to themselves.

Lemma 4.11 (Value Evaluation) *For any value v, $v \hookrightarrow v$*

Proof: See Exercise 4.18. □

Now we have all ingredients to state the main lemma in the proof of termination, the so called *logical relations lemma* [**?**]. The "logical relations" are $\|A\|$ and $|A|$, seen as unary relations, that is, predicates, on terms and values, respectively. They are "logical" since they are defined by induction on the structure of the type A, which corresponds to a proposition under the Curry-Howard isomorphism.

Lemma 4.12 (Logical Relations) *If $\Gamma;\Delta \vdash M : A$, $\eta \in \|\Gamma\|$ and $\theta \in |\Delta|$ then $[\eta;\theta]M \in \|A\|$.*

Before showing the proof, we extend the definition of the logical relations to all the types we have been considering.

$$
\begin{array}{rll}
M \in \|A\| & \text{iff} & M \hookrightarrow v \text{ and } v \in |A| \\
v \in |A_2 \multimap A_1| & \text{iff} & v = \hat{\lambda} w{:}A_2.\ M_1 \text{ and } [v_2/w]M_1 \in \|A_1\| \text{ for any } v_2 \in |A_2| \\
v \in |A_1 \otimes A_2| & \text{iff} & v = v_1 \otimes v_2 \text{ where } v_1 \in |A_1| \text{ and } v_2 \in |A_2| \\
v \in |\mathbf{1}| & \text{iff} & v = \star \\
v \in |A_1 \& A_2| & \text{iff} & v = \langle M_1, M_2 \rangle \text{ where } M_1 \in \|A_1\| \text{ and } M_2 \in \|A_2\| \\
v \in |\top| & \text{iff} & v = \langle\,\rangle \\
v \in |A_1 \& A_2| & \text{iff} & \text{either } v = \mathrm{inl}^{A_2}\, v_1 \text{ and } v_1 \in |A_1|, \\
 & & \text{or } v = \mathrm{inr}^{A_1}\, v_2 \text{ and } v_2 \in |A_2| \\
v \in |\mathbf{0}| & & \text{never} \\
v \in |!A| & \text{iff} & v = !M \text{ and } M \in \|A\| \\
v \in |A_2 \to A_1| & \text{iff} & v = \lambda u{:}A_2.\ M_1 \text{ and } [M_2/u]M_1 \in \|A_1\| \text{ for any } M_2 \in \|A_2\|
\end{array}
$$

These definitions are motivated directly from the form of values in the language. One can easily see that it is indeed inductive on the structure of the type. If we tried to add recursive types in a similar way, the proof below would still go through, except that the definition of the logical relation would no longer be well-founded.

Proof: (of the logical relations lemma 4.12). The proof proceeds by induction on the structure of the typing derivation $\mathcal{D} :: (\Gamma; \Delta \vdash M : A)$. We show three cases—all others are similar.

Case: $\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \Gamma; \Delta \vdash M_1 : A_2 \multimap A_1 & \Gamma; \Delta \vdash M_2 : A_2\end{array}}{\Gamma; \Delta \vdash M_1 \,\hat{}\, M_2 : A_1} \multimap \mathrm{E}.$

$$
\begin{array}{ll}
\eta \in \|\Gamma\| & \text{by assumption} \\
\theta \in |\Delta| & \text{by assumption} \\
[\eta;\theta]M_1 \in \|A_2 \multimap A_1\| & \text{by ind. hyp. on } \mathcal{D}_1 \\
\mathcal{E}_1 :: ([\eta;\theta]M_1 \hookrightarrow v_1) \text{ and } v_1 \in |A_2 \multimap A_1| & \text{by definition of } \|A_2 \multimap A_1\| \\
v_1 = \hat{\lambda} w{:}A_1.\ M_1' \text{ and } [v_2/w]M_1' \in \|A_1\| \text{ for any } v_2 \in |A_2| & \text{by definition of } |A_2 \multimap A_1| \\
[\eta;\theta]M_2 \in \|A_2\| & \text{by ind. hyp. on } \mathcal{D}_2 \\
\mathcal{E}_2 :: ([\eta;\theta]M_2 \hookrightarrow v_2) \text{ and } v_2 \in |A_2| & \text{by definition of } \|A_2\| \\
[v_2/w]M_1' \in \|A_1\| & \text{since } v_2 \in |A_2| \\
\mathcal{E}_3 :: ([v_2/w]M_1' \hookrightarrow v) \text{ and } v \in |A_1| & \text{by definition of } \|A_1\| \\
\mathcal{E} :: ([\eta;\theta](M_1 \,\hat{}\, M_2) \hookrightarrow v) & \text{by } \multimap \mathrm{Ev} \text{ from } \mathcal{E}_1, \mathcal{E}_2, \text{ and } \mathcal{E}_3 \\
[\eta;\theta](M_1 \,\hat{}\, M_2) \in \|A_1\| & \text{by definition of } \|A_1\|
\end{array}
$$

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Gamma; (\Delta, w{:}A_2) \vdash M_1 : A_1\end{array}}{\Gamma; \Delta \vdash \hat{\lambda} w{:}A_2.\ M_1 : A_2 \multimap A_1} \multimap \mathrm{I}.$

$$
\begin{array}{ll}
\eta \in \|\Gamma\| & \text{by assumption}
\end{array}
$$

$\theta \in \|\Delta\|$	by assumption
$\mathcal{E} :: ([\eta;\theta](\hat{\lambda} w{:}A_2.\, M_1) \hookrightarrow [\eta;\theta](\hat{\lambda} w{:}A_2.\, M_1))$	by $\multimap$Iv
$v_2 \in \|A_2\|$	assumption
$(\theta, v_2/w) \in \|\Delta, w{:}A_2\|$	by definition of $\|\Delta\|$
$[\eta; (\theta, v_2/w)]M_1 \in \\|A_1\\|$	by ind. hyp. on $\mathcal{D}_1$
$[v_2/w]([\eta; (\theta, w/w)]M_1) \in \\|A_1\\|$	by properties of substitution
$(\hat{\lambda} w{:}A_2.\, [\eta; (\theta, w/w)]M_1) \in \|A_2 \multimap A_1\|$	by definition of $\|A_2 \multimap A_1\|$
$[\eta;\theta](\hat{\lambda} w{:}A_2.\, M_1) \in \|A_2 \multimap A_1\|$	by properties of substitution
$[\eta;\theta](\hat{\lambda} w{:}A_2.\, M_1) \in \\|A_2 \multimap A_1\\|$	by definition of $\\|A_2 \multimap A_1\\|$

Case: $\mathcal{D} = \dfrac{}{\Gamma; (\cdot, w{:}A) \vdash w : A}\, w.$

$\theta \in \|\cdot, w{:}A\|$	by assumption
$[\theta]w \in \|A\|$	by definition of $\|\cdot, w{:}A\|$
$\mathcal{E} :: ([\eta;\theta]w \hookrightarrow [\eta;\theta]w)$	by Lemma 4.11
$[\eta;\theta]w \in \\|A\\|$	by definition of $\\|A\\|$

$\square$

The termination theorem follows directly from the logical relations lemma. Note that the theorem excludes recursive types and the fixpoint operator by a general assumption for this section.

Theorem 4.13 (Termination) *If* $\cdot;\cdot \vdash M : A$ *then* $M \hookrightarrow v$ *for some* v.

Proof: We have $\cdot \in \|\cdot\|$ and $\cdot \in |\cdot|$ since the conditions are vacuously satisfied. Therefore, by the logical relations lemma 4.12, $[\cdot;\cdot]M \in \|A\|$. By the definition of $\|A\|$ and the observation that $[\cdot;\cdot]M = M$, we conclude that $M \hookrightarrow v$ for some v. $\square$

4.8 Exercises

Exercise 4.1 Prove that if $\Gamma; \Delta \vdash M : A$ and $\Gamma; \Delta \vdash M : A'$ then $A = A'$.

Exercise 4.2 A function in a functional programming language is called *strict* if it is guaranteed to use its argument. Strictness is an important concept in the implementation of lazy functional languages, since a strict function can evaluate its argument eagerly, avoiding the overhead of postponing its evaluation and later memoizing its result.

In this exercise we design a λ-calculus suitable as the core of a functional language which makes strictness explicit at the level of types. Your calculus should contain an unrestricted function type $A \to B$, a strict function type $A \twoheadrightarrow B$, a vacuous function type $A \dashrightarrow B$, a full complement of operators refining product and disjoint sum types as for the linear λ-calculus, and a modal operator to internalize the notion of closed term as in the linear λ-calculus. Your calculus should not contain quantifiers.

1. Show the introduction and elimination rules for all types, including their proof terms.
2. Given the reduction and expansions on the proof terms.
3. State (without proof) the valid substitution principles.
4. If possible, give a translation from types and terms in the strict λ-calculus to types and terms in the linear λ-calculus such that a strict term is well-typed if and only if its linear translation is well-typed (in an appropriately translated context).
5. Either sketch the correctness proof for your translation in each direction by giving the generalization (if necessary) and a few representative cases, or give an informal argument why such a translation is not possible.

Exercise 4.3 Give an example which shows that the substitution $[M/w]N$ must be capture-avoiding in order to be meaningful. *Variable capture* is a situation where a bound variable w' in N occurs free in M, and w occurs in the scope of w'. A similar definition applies to unrestricted variables.

Exercise 4.4 Give a counterexample to the conjecture that if $M \longrightarrow_\beta M'$ and $\Gamma; \Delta \vdash M' : A$ then $\Gamma; \Delta \vdash M : A$. Also, either prove or find a counterexample to the claim that if $M \longrightarrow_\eta M'$ and $\Gamma; \Delta \vdash M' : A$ then $\Gamma; \Delta \vdash M : A$.

Exercise 4.5 The proof term assignment for sequent calculus identifies many distinct derivations, mapping them to the same natural deduction proof terms. Design an alternative system of proof terms from which the sequent derivation can be reconstructed uniquely (up to weakening of unrestricted hypotheses and absorption of linear hypotheses in the $\top$R rule).

1. Write out the term assignment rules for all propositional connectives.
2. Give a calculus of reductions which corresponds to the initial and principal reductions in the proof of admissibility of cut.
3. Show the reduction rule for the dereliction cut.
4. Show the reduction rules for the left and right commutative cuts.
5. Sketch the proof of the subject reduction properties for your reduction rules, giving a few critical cases.
6. Write a translation judgment $S \Longrightarrow M$ from faithful sequent calculus terms to natural deduction terms.
7. Sketch the proof of type preservation for your translation, showing a few critical cases.

Exercise 4.6 Supply the missing rules for $\oplus$E in the definition of the judgment $\Gamma; \Delta_I \setminus \Delta_O \vdash_i M : A$ and show the corresponding cases in the proof of Lemma 4.5.

Exercise 4.7 In this exercise we explore the syntactic expansion of *extended case expressions* of the form **case** M **of** m.

1. Define a judgment which checks if an extended case expression is valid. This is likely to require some auxiliary judgments. You must verify that the cases are exhaustive, circumscribe the legal patterns, and check that the overall expression is linearly well-typed.

2. Define a judgment which relates an extended case expression to its expansion in terms of the primitive **let** , **case** , and abort constructs in the linear λ-calculus.

3. Prove that an extended case expression which is valid according to your criteria can be expanded to a well-typed linear λ-term.

4. Define an operational semantics directly on extended case expressions.

5. Prove that your direct operational semantics is correct on valid patterns with respect to the translational semantics from questions 2.

Exercise 4.8 Define the judgment $M \longrightarrow_\beta^* M'$ via inference rules. The rules should directly express that it is the congruent, reflexive and transitive closure of the β-reduction judgment $M \longrightarrow_\beta M'$. Then prove the generalized subject reduction theorem 4.7 for your judgment. You do not need to show all cases, but you should carefully state your induction hypothesis in sufficient generality and give a few critical parts of the proof.

Exercise 4.9 Define *weak β-reduction* as allows simple β-reduction under $\otimes$, inl, and inr constructs and in all components of the elimination form. Show that if M weakly reduces to a value v then $M \hookrightarrow v$.

Exercise 4.10 Prove type preservation (Theorem 4.9) directly by induction on the structure of the evaluation derivation, using the substitution lemma 4.2 as necessary, but without appeal to subject reduction.

Exercise 4.11 Prove the subject reduction and expansion properties for recursive type computation rules.

Exercise 4.12 [*An exercise exploring the use of type conversion rules without explicit term constructors.*]

Exercise 4.13 Define a linear multiplication function mult : $\mathrm{nat} \multimap \mathrm{nat} \multimap \mathrm{nat}$ using the functions copy and delete.

Exercise 4.14 Defined the following functions on lists. Always explicitly state the type, which should be the most natural type of the function.

1. append to append two lists.

2. concat to append all the lists in a list of lists.

3. map to map a function f over the elements of a list. The result of mapping f over the list $x_1, x_2, \ldots, x_n$ should be the list $f(x_1), f(x_2), \ldots f(x_n)$, where you should decide if the application of f to its argument should be linear or not.

4. foldr to reduce a list by a function f. The result of folding f over a list $x_1, x_2, \ldots x_n$ should be the list $f(x_1, f(x_2, \ldots, f(x_n, \mathit{init})))$, where *init* is an initial value given as argument to foldr. You should decide if the application of f to its argument should be linear or not.

5. copy, delete, and promote.

Exercise 4.15 For one of the form of lazy lists on Page 109, define the functions from Exercise 4.14 plus a function toList which converts the lazy to an eager list (and may therefore not terminate if the given lazy lists is infinite). Make sure that your functions exhibit the correct amount of laziness. For example, a map function applied to a lazy list should not carry out any non-trivial computation until the result is examined.

Further for your choice of lazy list, define the infinite lazy list of eager natural numbers $0, 1, 2, \ldots$.

Exercise 4.16 Prove that there is no term v such that $\omega\,(\mathrm{fold}^{\Omega}\,\omega) \hookrightarrow v$.

Exercise 4.17 [*An exercise about the definability of fixpoint operators at various type.*]

Exercise 4.18 Prove Lemma 4.11 which states that all values evaluate to themselves.