$\mathrm{llist}^2_A = \mu\alpha.\, \mathbf{1} \oplus {!}(A \otimes \alpha)$**.** 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 6.15).

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 6.16).

6.5 Exercises

Exercise 6.1 Prove that if $\Gamma; \Delta \vdash M : A$ and $\Gamma; \Delta \vdash M : A'$ then $A = A'$.

Exercise 6.2 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 $A \rightarrow B$, 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 6.3 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 6.4 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 6.5 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 6.6 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 6.4.

Exercise 6.7 In this exercise we explore the syntactic expansion of *extended case expressions* of the form $\mathsf{case}\, M \,\mathsf{of}\, m$.

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 6.8 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 6.6 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 6.9 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 6.10 Prove type preservation (Theorem 6.8) directly by induction on the structure of the evaluation derivation, using the substitution lemma 6.2 as necessary, but without appeal to subject reduction.

Exercise 6.11 Prove the subject reduction and expansion properties for recursive type computation rules.

Exercise 6.12 [*An exercise exploring the use of type conversion rules without explicit term constructors.*]

Exercise 6.13 Define a linear multiplication function mult : nat $\multimap$ nat $\multimap$ nat using the functions copy and delete.

Exercise 6.14 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 6.15 For one of the form of lazy lists on Page 130, define the functions from Exercise 6.14 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 6.16 Prove that there is no term v such that $\omega\,(\mathrm{fold}^{\Omega}\,\omega) \hookrightarrow v$.

Exercise 6.17 [*An exercise about the definability of fixpoint operators at various type.*]

Exercise 6.18 Prove Lemma **??** which states that all values evaluate to themselves.

Exercise 6.19 In this exercise we explore strictness as a derived, rather than a primitive concept. Recall that a function is *strict* if it uses its argument at least once. The strictness of a function from A to B can be enforced by the type $(A \otimes {!A}) \multimap B$.

1. Show how to represent a strict function $\lambda^s x{:}A.\ M$ under this encoding.

2. Show how to represent an application of a strict function M to an argument N.

3. Give natural evaluation rules for strict functions and strict applications.

4. Show the corresponding computation under the encoding of strict functions in the linear λ-calculus.

5. Discuss the merits and difficulties of the given encoding of the strict in the linear λ-calculus.

Exercise 6.20 In the exercise we explore the *affine* λ-calculus. In an affine hypothetical judgment, each assumption can be used at most once. Therefore, it is like linear logic except that affine hypotheses need not be used.

The affine hypothetical judgment $\Gamma; \Delta \vdash^a A\ \mathit{true}$ is characterized by the hypothesis rule

$$\frac{}{\Gamma; \Delta, x{:}A\ \mathit{true} \vdash^a A\ \mathit{true}}$$

and the substitution principle

if $\Gamma; \Delta \vdash^a A\ \mathit{true}$ *and* $\Gamma; \Delta', A\ \mathit{true} \vdash^a C\ \mathit{true}$ *then* $\Gamma; \Delta, \Delta' \vdash^a C\ \mathit{true}$.

1. State the remaining hypothesis rule and substitution principle for unrestricted hypotheses.

2. Give introduction and elimination rules for affine implication ($A \leadsto B$) simultaneous conjunction, alternative conjunction, and truth. Note that there is only one form of truth: since assumptions need not be used, the multiplicative and additive form coincide.

3. Give a proof term assignment for affine logic.

4. We can map affine logic to linear logic by translating every affine function $A \leadsto B$ to a linear function $(A \& \mathbf{1}) \multimap B$. Give a corresponding translation for all proof terms from the affine logic to linear logic.

5. We can also map affine logic to linear logic by translating every affine function $A \leadsto B$ into function $A \multimap (B \otimes \top)$. Again give a corresponding translation for all proof terms from affine logic to linear logic.

6. Discuss the relative merits of the two translations.

7. [Extra Credit] Carefully formulate and prove the correctness of one of the two translations.