

Assignment 2

Deadline for solutions: 01.06.2019

Exercise 1 Intuitionistic Tautologies

(6 Points)

The logic of Martin-Löf type theory underlying Agda is *intuitionistic*, and is, very roughly, obtained from the *classical* logic by removing the *law of excluded middle (LEM)*: $\phi \vee \neg\phi$, which is also equivalent to the principle of *double negation elimination (DNE)*: $\neg\neg\phi \rightarrow \phi$.

Formalize and prove the following in Agda:

1. equivalence of LEM and DNE;
2. $\neg\neg\neg\phi \rightarrow \neg\phi$ (i.e. DNE with $\neg\phi$ instead of ϕ);
3. $\exists x. \neg\phi(x) \rightarrow \neg\forall x. \phi(x)$ (a one-sided De Morgan law for quantifiers).

Here $\neg\phi$ encodes $\phi \rightarrow \perp$ and by “equivalence” we mean precisely mutual implication.

Exercise 2 Filtering Vectors

(6 Points)

Implement a `filter` function for vectors in Agda by adapting the corresponding function for lists and by using dependent sums for the output type.

Formalize and prove the following properties in Agda.

1. Filtering a vector by a predicate yields a vector whose all elements satisfy the predicate.
2. If all the elements of a vector satisfy a predicate then filtering this vector by this predicate yields the original vector.
3. Filtering by the same predicate is an idempotent operation.

Hint: Take inspiration from the corresponding properties of filtering for lists. Beware of **Ill-typed with abstractions**.

Exercise 3 Matrices

(6 Points)

1. Using the vector type \mathbb{V} in a nested fashion, fill in the hole below to define a type of integer matrices

```
_by_matrix : ℕ → ℕ → Set
n by m matrix = ?
```

2. Define the following basic operations on matrices. You should first figure out the types of the operations, of course, and then write code for them (possibly using helper functions).

- (a) `matrix-at`, which takes in an n by m matrix and a row and column index within those bounds, and returns the element stored at that position in the matrix.
- (b) `identity-matrix`, which takes in a dimension n , and returns the n by n matrix which has zero everywhere except 1 down the diagonal of the matrix.
- (c) `*matrix_`, which multiplies an n by k matrix and a k by m matrix to obtain an n by m matrix.

Exercise 4 Braun Trees and Merge Sort (12 Points)

1. Formalize the ordering property of Braun trees and prove that `bt-insert` preserves it.

Hint: One possible way to formulate the requested ordering property is in two stages as follows:

```
-- the top element is greater than the given element
-- and for every subtree the top element is greater than the descendants
bt-ord-h : ∀ {n : ℕ} (a : A) (t : braun-tree n) → Set
bt-ord-h _ bt-empty = ⊤
bt-ord-h a (bt-node b l r _) = a ≤A b ≡ tt ∧ bt-ord-h b l ∧ bt-ord-h b r

-- for every subtree the top element is greater than the descendants
bt-ord : ∀ {n : ℕ} (t : braun-tree n) → Set
bt-ord bt-empty = ⊤
bt-ord (bt-node a l r _) = bt-ord-h a l ∧ bt-ord-h a r
```

2. `list-merge-sort.agda` of the Iowa Agda library contains an implementation of merge sort using Braun trees. Prove that the length of the input list is equal to the length of the output list.

3. Analogously, formalize and prove that the resulting list, sorted by `merge-sort` is indeed sorted.

Hint: You need to assume totality of the comparison relation \leq , i.e. that the type `total _≤A_` is inhabited. This can be done elegantly using the module mechanism (as e.g. in `bst.agda`). Consider proving the following lemma:

```
sorted : ∀ (l : ℒ A) → Set
sorted l = ∀ (i : ℕ) (a b : A)
  → just a ≡ nth i l
  → just b ≡ nth (suc i) l
  → (a ≤A b) ≡ tt

merge-sorted : ∀ (l r : ℒ A) → (sorted l) → (sorted r) → sorted (merge l r)
```

(most of the complexity is concentrated in the base of induction, which amounts to an extensive case distinction over possible mutual arrangements of the initial list elements.