Type Inference
Abstract Data Types

Exercise 1. Derive the equations and solve them to find the type for:

`let cadr l = List.hd (List.tl l) in cadr (1::2::[]), cadr (true::false::[])`

in environment $\Gamma = \{$`List.hd`$: \forall \alpha . \alpha \, \mathrm{list} \rightarrow \alpha;$ `List.tl`$: \forall \alpha . \alpha \, \mathrm{list} \rightarrow \alpha \, \mathrm{list}\}$. You can take "shortcuts" if it is too many equations to write down.

Exercise 2. *Terms* $t_1, t_2, \ldots \in T(\Sigma, X)$ are built out of variables $x, y, \ldots \in X$ and function symbols $f, g, \ldots \in \Sigma$ the way you build values out of functions:

- $X \subset T(\Sigma, X)$ – variables are terms; usually an infinite set,
- for terms $t_1, \ldots, t_n \in T(\Sigma, X)$ and a function symbol $f \in \Sigma_n$ of arity n, $f(t_1, \ldots, t_n) \in T(\Sigma, X)$ – bigger terms arise from applying function symbols to smaller terms; $\Sigma = \dot{\cup}_n \Sigma_n$ is called a signature.

In OCaml, we can define terms as: `type term = V of string | T of string * term list` , where for example `V("x")` is a variable x and `T("f", [V("x"); V("y")])` is the term $f(x, y)$.

By *substitutions* σ, ρ, ... we mean finite sets of variable, term pairs which we can write as $\{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$ or $[x_1 := t_1; \ldots; x_k := t_k]$, but also functions from terms to terms $\sigma: T(\Sigma, X) \rightarrow T(\Sigma, X)$ related to the pairs as follows: if $\sigma = \{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$, then

- $\sigma(x_i) = t_i$ for $x_i \in \{x_1, \ldots, x_k\}$,
- $\sigma(x) = x$ for $x \in X \backslash \{x_1, \ldots, x_k\}$,
- $\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$.

In OCaml, we can define substitutions σ as: `type subst = (string * term) list`, together with a function `apply : subst -> term -> term` which computes $\sigma(\cdot)$.

We say that a substitution σ is *more general* than all substitutions $\rho \circ \sigma$, where $(\rho \circ \sigma)(x) = \rho(\sigma(x))$. In type inference, we are interested in most general solutions: the less general type judgement `List.hd`: $\mathrm{int} \, \mathrm{list} \rightarrow \mathrm{int}$, although valid, is less useful than `List.hd`$: \forall \alpha . \alpha \, \mathrm{list} \rightarrow \alpha$ because it limits the usage of `List.hd`.

A *unification problem* is a finite set of equations $S = \{s_1 =^? t_1, \ldots, s_n =^? t_n\}$ which we can also write as $s_1 \doteq t_1 \wedge \ldots \wedge s_n \doteq t_n$. A solution, or *unifier* of S, is a substitution σ such that $\sigma(s_i) = \sigma(t_i)$ for $i = 1, \ldots, n$. A *most general unifier*, for short *MGU*, is a most general such substitution.

A substitution is *idempotent* when $\sigma = \sigma \circ \sigma$. If $\sigma = \{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$, then σ is idempotent exactly when no t_i contains any of the variables $\{x_1, \ldots, x_n\}$; i.e. $\{x_1, \ldots, x_n\} \cap \mathrm{Vars}(t_1, \ldots, t_n) = \varnothing$.

1. Implement an algorithm that, given a set of equations represented as a list of pairs of terms, computes an idempotent most general unifier of the equations.
2. * (Ex. 4.22 in *Franz Baader and Tobias Nipkov "Term Rewriting and All That"*, p. 82.) Modify the implementation of unification to achieve linear space complexity by working with what could be called iterated substitutions. For example, the solution to $\{x =^? f(y), y =^? g(z), z =^? a\}$ should be represented as variable, term pairs $(x, f(y))$, $(y, g(z))$, (z, a). (Hint: iterated substitutions should be unfolded lazily, i.e. only so far that either a non-variable term or the end of the instantiation chain is found.)

Exercise 3.

1. What does it mean that an implementation has junk (as an algebraic structure for a given signature)? Is it bad?
2. Define a monomorphic algebraic specification (other than, but similar to, nat_p or string_p, some useful data type).
3. Discuss an example of a (monomorphic) algebraic specification where it would be useful to drop some axioms (giving up monomorphicity) to allow more efficient implementations.

Exercise 4.

1. Does the example ListMap meet the requirements of the algebraic specification for maps? Hint: here is the definition of List.remove_assoc; compare a x equals 0 if and only if a = x.

```
 let rec remove_assoc x = function
| [] -> []
| (a, b as pair) :: l ->
    if compare a x = 0 then l else pair :: remove_assoc x l
```

2. Trick question: what is the computational complexity of ListMap or TrivialMap?

3. * The implementation MyListMap is inefficient: it performs a lot of copying and is not tail-recursive. Optimize it (without changing the type definition).

4. Add (and specify) isEmpty: $(\alpha,\ \beta)$ map $\rightarrow$ bool to the example algebraic specification of maps without increasing the burden on its implementations (i.e. without affecting implementations of other operations). Hint: equational reasoning might be not enough; consider an equivalence relation $\approx$ meaning "have the same keys", defined and used just in the axioms of the specification.

Exercise 5. Design an algebraic specification and write a signature for first-in-first-out queues. Provide two implementations: one straightforward using a list, and another one using two lists: one for freshly added elements providing efficient queueing of new elements, and "reversed" one for efficient popping of old elements.

Exercise 6. Design an algebraic specification and write a signature for sets. Provide two implementations: one straightforward using a list, and another one using a map into the unit type.

- To allow for a more complete specification of sets here, augment the maps ADT with generally useful operations that you find necessary or convenient for map-based implementation of sets.

Exercise 7.

1. (Ex. 2.2 in *Chris Okasaki "Purely Functional Data Structures"*) In the worst case, member performs approximately $2d$ comparisons, where d is the depth of the tree. Rewrite member to take no mare than $d+1$ comparisons by keeping track of a candidate element that *might* be equal to the query element (say, the last element for which $<$ returned false) and checking for equality only when you hit the bottom of the tree.

2. (Ex. 3.10 in *Chris Okasaki "Purely Functional Data Structures"*) The balance function currently performs several unnecessary tests: when e.g. ins recurses on the left child, there are no violations on the right child.

   a. Split balance into lbalance and rbalance that test for violations of left resp. right child only. Replace calls to balance appropriately.

   b. One of the remaining tests on grandchildren is also unnecessary. Rewrite ins so that it never tests the color of nodes not on the search path.