MP 4 – User Defined Datatypes and Recursion

CS 421 – su 2011

Assigned June 26, 2011
Due July 3, 2011 23:59
Extension 48 hours (20% penalty)

1 Change Log

1.0 Initial Release.

2 Objectives and Background

The purpose of this MP is to help the student master:

1. user-defined datatypes
2. recursion over those datatypes

3 Instructions

You may use any and all library functions. We highly recommend the `List` library, for which documentation can be found at http://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html .

In particular, you may want to look at:

- `map`
- `fold_left`
- `fold_right`
- `mem`
- `filter`

You may also find other `List` library functions useful.

4 Problems

4.1 `Records`

In `mp4common.ml`, you are given the following record type:

```
type 'a payroll_record = {id : 'a; salary : float}
```

1. (3 pts) Write a function `dept_payroll : 'a payroll_record list -> 'a list payroll_record` such that when `dept_payroll` is given a list of payroll records, with the `id` field containing a string (presumed to be the name of a department member), and the `salary` field containing a `float` (presumed to be the department member's salary), it returns a payroll record whose `id` field contains a listing of department memebrs, in the order given, and the sum of the salary entries for the members. The salary of an empty department is `0.0`.

```
# let dept_payroll dept_list = ...
val dept_payroll :
  'a Mp4common.payroll_record list -> 'a list Mp4common.payroll_record =
  <fun>
# dept_payroll [{id = "Jose Miguel"; salary = 26543.70};
               {id = "Sue Wright"; salary = 25794.35}];;
 - : string list Mp4common.payroll_record =
{id = ["Jose Miguel"; "Sue Wright"]; salary = 52338.05}
```

4.2 `'a option` Datatype

The following problem is intended to familiarize you with using the given datatype from class:

```
type 'a option = None | Some of 'a
```

2. (6 pts) Write a function `sum_smallest : int list list -> int option` such that if `sum_smallest` is given a non-empty list of non-empty lists of integers, it returns `(Some s)` where `s` is the sum of the smallest element from each of the lists in the given list, and otherwise it returns `None`.

```
let sum_smallest list_list = ...;;
val sum_smallest : int list list -> int option = <fun>
# sum_smallest [[2;5;-3]; [12; 16]; [5]];;
- : int option = Some 14
```

Hint: you may want to write a helper function.

4.3 An Enumeration Type: `ord`

In `mp4common.ml`, you are given the following enumeration type:

```
type ord = GT (* greater *) | EQ (* equal *) | LT (* less *)
```

This type can be used as the return type of comparison functions.

3. (3 pts) Write a function `gencompare : 'a -> 'a -> ord` that compares its two arguments, and returns `EQ` if they are equal, `LT` if the first one is less than the second one and `GT` otherwise. Such a function will only be usable on types not involving function types.

```
# let gencompare a b = ...
val gencompare : 'a -> 'a -> Mp4common.ord = <fun>
# gencompare "hello" "world";;
- : Mp4common.ord = LT
```

4. (5 pts) Write a function `insert : ('a -> 'a -> ord) -> 'a -> 'a list -> 'a list` that inserts an element in a list passed as a first argument, which may be assumed to be sorted in ascending order with respect to the comparison function. `insert cmp x l` inserts `x` in `l` right before the first element `e` that is greater than `x` (`cmp e x` returns `GT`). However, if the function encounters an element that is equal to `x` (`cmp e x` returns `EQ`) before reaching one that is greater then `x`, it discards `x` and returns the original list. If no element in the list is greater than or equal to `x`, `x` is inserted as the last element in the list.

   ```
   # let rec insert cmp e l = ...
   val insert : ('a -> 'a -> Mp4common.ord) -> 'a -> 'a list -> 'a list = <fun>
   # insert gencompare (-1) [-2; 0; 2];;
   - : int list = [-2; -1; 0; 2]
   ```

4.4 `'a prop` Datatype

In the next few problems, you will be working with two additional datatypes designed to model propositional formulae and to facilitate logical calculations over them. These datatypes are as follows:

```
type 'a atom = Pos of 'a | Neg of 'a
type 'a prop =
  | Atom of 'a atom
  | Not of 'a prop
  | Or of 'a prop * 'a prop
  | And of 'a prop * 'a prop
  | Implies of 'a prop * 'a prop
```

Propositional formulae (or propositions, for short) are represented by the type `'a prop`. The only non-self-explanatory case is that of `Atom`. As the type says, `Atom` injects the type of `atom` in propositions. Atoms are boolean variables or negated boolean variables. A negated boolean variable `Neg x` is logically equivalent to `Not(Pos x)`, the negation of the boolean variable. We distinguish this special case of a negated boolean variable for algorithmic reasons to be used in the extra-credit problem.

Both `atom` and `prop` are polymorphic in a type of boolean (or truth-valued) variables. Our ultimate goal is to assign a value of `true` or `false` to some subset of the given variables, thus making a given formula true. In `mp4common.ml`, we have supplied an example type `boolVar` our truth-valued variables. To help you test your code it is as follows:

```
type boolVar = A | B | C | D | E
```

5. (5 pts) Write a function `compare: 'a atom -> 'a atom -> ord` that compares two atoms. Atoms having different bases are ordered by `gencompare` on their bases. For `Neg x` and `Pos x` (same `x`), `Neg x` is strictly less than `Pos x`.

   ```
   # let compare a1 a2 = ...
   val compare : 'a Mp4common.atom -> 'a Mp4common.atom -> Mp4common.ord = <fun>
   # compare (Pos 1) (Neg 1);;
   - : Mp4common.ord = GT
   ```

6. (5 pts) Write a function `eliminateImplies : 'a prop -> 'a prop` that takes in a proposition and eliminates all uses of implication within that proposition, using the fact that $A \Rightarrow B \equiv (\neg A) \vee B$.

```
# let rec eliminateImplies p = ...
val eliminateImplies : 'a Mp4common.prop -> 'a Mp4common.prop = <fun>
# eliminateImplies (Implies (Atom (Pos 0), Atom (Pos 1)));;
- : int Mp4common.prop = Or (Not (Atom (Pos 0)), Atom (Pos 1))
```

7. (5 pts) Write a function `eliminateNot : 'a prop -> 'a prop` that takes in a proposition and eliminates all uses of `Not`, pushing all negations down to the atoms. Use De Morgan's laws and double negation elimination: $\neg\neg A \equiv A$, $\neg(A \vee B) \equiv (\neg A) \wedge (\neg B)$, $\neg(A \wedge B) \equiv (\neg A) \vee (\neg B)$, and the fact that "not" of positive atom is a negative one and vice versa.
Note: You may assume that there's no use of the `Implies` constructor in the input proposition. You will be tested only on this kind of input.

```
# let rec eliminateNot p = ...
val eliminateNot : 'a Mp4common.prop -> 'a Mp4common.prop = <fun>
# eliminateNot (Not (And (Atom (Pos 0), Atom (Pos 1))));;
- : int Mp4common.prop = Or (Atom (Neg 0), Atom (Neg 1))
```

8. (5 pts) Write a function `dnf : 'a prop -> 'a prop` that takes in a proposition and returns a Disjunctive Normal Form (DNF) of this proposition. A proposition is in DNF iff it is a disjunction of one or more conjunctions of one or more atoms. Use distributive law: $A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$
Note: You may assume that there's no use of the `Implies` and `Not` constructors in the input proposition. You will be tested only on this kind of inputs.

```
# let rec dnf p = ...
val dnf : 'a Mp4common.prop -> 'a Mp4common.prop = <fun>
# dnf (And (Or(Atom (Pos 0), Atom (Pos 1)), Atom (Pos 2)));;
- : int Mp4common.prop =
Or (And (Atom (Pos 0), Atom (Pos 2)), And (Atom (Pos 1), Atom (Pos 2)))
```

4.5 Building a SAT Solver

In the previous 4 problems, you have incrementally implemented a function for putting propositional formulae into disjunctive normal form. To reflect this in the type system, we introduce two new types to represent a conjunct of atoms, and a disjunct of conjuncts, as follows: Type `disj` represents a proposition in DNF. `disj` is a list of conjunctions, and `conj`, each of which is a list of atoms.

```
type 'a conj = Conjunction of 'a atom list
type 'a disj = Disjunction of 'a conj list
```

9. (6 pts) Write a function `dnf_compact : 'a Mp4common.prop -> 'a disj` that takes a proposition and returns its DNF. The lists of atoms should be sorted with respect to `compare` function from Problem 5.
Hint: you may use functions from previous sections.

```
# let dnf_compact p =
val dnf_compact : 'a Mp4common.prop -> 'a disj = <fun>
# dnf_compact (Implies (Atom (Pos A), Atom (Pos B)));;
- : boolVar disj = Disjunction [Conjunction [Neg A]; Conjunction [Pos B]]
```

10. (8 pts) Write a function `solve : 'a disj -> ('a * bool) list option` that takes a proposition and if that proposition can be satisfied, it returns a `Some(l)`, where `l` is a mapping of variables to boolean (list of pairs); and returns `None` if no such mapping exists. `l` should contain a list of variable-boolean pairs, such that no variable occurs more than once and if the variables in the given formula are assigned the values in the list, then the formula will be logically equivalent to True no matter what values the variables not in the list are given. You may return the list of pairs in any order you wish.

```
# let solve d = ...
val solve : 'a Mp4common.disj -> ('a * bool) list option = <fun>
# solve (Disjunction [Conjunction [Pos A; Neg B]; Conjunction [Pos C]]);;
- : (Mp4common.boolVar * bool) list option = Some [(A, true); (B, false)]
```

4.6 Extra credit

11. (5 pts) Write a function `satisfies : 'a list -> 'a prop -> bool` that takes a valuation and a proposition, and returns true if the valuation makes the proposition true, and false otherwise. The valuation is represented by a list, where each variable that is present in the list is set to true, and all other variables are assumed to be false.

```
# let satisfies m p = ...
val satisfies : 'a list -> 'a Mp4common.prop -> bool = <fun>
# satisfies [A; B] (And(Atom(Pos A),Atom(Pos B)));;
- : bool = true
# satisfies [A] (And(Atom(Pos A),Atom(Neg B)));;
- : bool = true
```