Programming Assignment #7 (SML)

CS–671

due 27 April 2014 11:59 PM

The objective of this assignment is to implement structures of signature `PROP` of boolean formulas, or propositional formulas. It illustrates the use of user-defined datatypes (SML's `datatype`).

1 States

In the first part of the assignment, we wish to implement a notion of *state*. A state is a mapping from variable names to values (i.e., in a given state, a variable name corresponds to at most one value). We are especially interested in boolean states, in which variable names are associated with boolean values.

We consider the following signature of generic state and its specialization to boolean states.

```
signature STATE =
  sig
    type value
    type state
    val blankState : state
    val set : state -> string * value -> state
    val unset : state -> string -> state
    val get : state -> string -> value option
    val dumpState : state -> unit
  end

signature BOOL_STATE = STATE where type value = bool
```

`value` is the type of what is stored in the state; `state` is the state itself. The signature includes a blank state (a state in which no names are associated to values) and functions to set or unset a name, to get the current value for a name and to print all the names and values on the terminal:

- `set s (x,v)` returns a new state identical to state `s` except that variable `x` now has value `v`.
- `unset s x` returns a new state identical to state `s` except that variable `x` now has no value.
- `get s x` returns the value of `x` in state `s` as an option (it returns `NONE` if `x` in unset in state `s`).
- `dumpState s` prints all the names that are set in state `s` along with their values. Names are listed in alphabetical (`String.<`) order.

1. Write a structure `BoolListState` of signature `BOOL_STATE` in which states are implemented as lists of pairs (*name*, *value*) (i.e., a poor man's map). 20 pts

2. Write a structure `BoolPairState` of signature `BOOL_STATE` in which states are implemented as pairs of lists of strings: one list contains all the names set to *true*; the other list contains all the names set to *false*. 20 pts

2 Proposition Validity and Satisfiability

In the second part on the assignment, we want to implement a structure that can parse propositional formulas, evaluate them in a given state, and decide their validity or satisfiability by building truth tables.

Formulas are based on propositional (i.e., boolean) variables, conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$), implication ($\rightarrow$) and the two constants *true* and *false*. They are represented by the datatype `prop`:

```
datatype prop = Ident of string         (* a variable *)
              | T | F                   (* True and False *)
              | And of prop * prop      (* conjunction *)
              | Or of prop * prop       (* disjunction *)
              | Implies of prop * prop  (* implication *)
              | Not of prop             (* negation *)
```

A (boolean) state assigns boolean values to variable names. In a given state, a proposition can have the value *true*, the value *false*, or be unset (no value). These three possibilities are represented by the datatype `value`:

```
datatype value = True | False | Unknown
```

The remainder of the signature `PROP` is as follows:

- `type state` is the type of states used by this propositional calculator.
- `parse` parses a string into a value of type `prop`. It raises the exception `Parse` (with a message) if the string cannot be parsed as a proposition.
- `identifiers` returns a list of all the variable names that appear in the formula, in order.
- `eval` evaluates (as a `value`) a proposition in a given state.
- `satisfy` returns a state that satisfies the proposition (if any) or `NONE` if the proposition is unsatisfiable. A satisfying state is a state in which the proposition evaluates to `True`.
- `isValid` tests the validity of a proposition. A proposition is valid iff it evaluates to `True` in all possible states. If a proposition is not valid, the function displays a "counterexample" state (i.e., a state in which the formula evaluates to `False`) before it returns *false*.

Functions `satisfy` and `isValid` work by enumerating all the possible states (i.e., by building a truth table). This can be achieved in the following way:

- Build the list of all names in the formula; if the list is empty, there are no states to enumerate.
- Set the first name to *true* and recursively enumerate all the possible states on a shorter list on names.
- Set the first name to *false* and recursively enumerate all the possible states on a shorter list on names.

Function `satisfy` stops as soon as a state is found that satisfies the proposition; function `isValid` stops as soon as a state is found that does not satisfy the proposition. In the worst case, each function may enumerate all the possible states (i.e., this is not an efficient way to decide the satisfiability and validity of propositional formulas).

3. Write a structure `ListProp` of signature `PROP` in which — 30 pts

   ```
   type state = BoolListState.state
   ```

4. Write a structure `PairProp` of signature `PROP` in which — 30 pts

   ```
   type state = BoolPairState.state
   ```

Notes

- This assignment must be submitted in a file named `7.sml` in the `sml` directory of your repository. This file can load other files using function `use`, if necessary.
- The signature `PROP` is in the file `prop-sig.sml`; the signature `BOOL_STATE` is in the file `state-sig.sml`.
- Function `dumpState` must list variables in alphabetical order of names. For instance:

```
let
  open BoolPairState
  val s = set (set (set blankState ("a",true)) ("b",false)) ("c",true)
in
  dumpState s
end
```

  should produce the output:

```
a = true
b = false
c = true
```

- If at least one variable in the formula is unset in the state, the formula should evaluate to `Unknown`, even if the truth value could in theory be decided. For instance, the following code should return `Unknown`, even though the formula will evaluate to *true* for any possible values of `B` and `C`:

```
let
  open BoolPairState
  open PairProp
  val e = parse "A->B&C"
  val s = set blankState ("A",false)
in
  eval s e
end
```

- The function `parse : string -> prop` parses strings into formulas according to the following syntax:
  - `&` is conjunction, `|` is disjunction, `->` is implication, `~` is negation, `True` is *true* and `False` is *false*
  - any sequence of letters (letters only) other than `True` or `False` is a variable name
  - precedence of operators, from high to low is as follows: `~`, `&`, `|`, `->`
  - precedence can be overridden using parentheses
  - whitespaces are ignored

  It is implemented as a recursive descent in the file `pairPropStub.sml`. However, the implementation relies on a function `tokenize`, which must be implemented as part of the assignment. This function breaks a string into a list of symbols. For instance,

```
tokenize  "A&B|C -> (A->True)|B"
```

  returns

```
["A","&","B","|","C","->","(","A","->","True",")","|","B"]
```

- Exceptionally for this assignment, it is allowed (and expected) that some code is duplicated between structures `BoolListState` and `BoolPairState` and between structures `ListProp` and `Pairprop`. It is recommended (but not required) to use a functor here.