Technical University of Munich
Chair for Logic and Verification
Prof. Tobias Nipkow, Ph.D.
J. Rädle, L. Stevens, K. Kappelmann, MC M. Eberl

WS 2019/20
13.12.2019
Deadline: 30.12.2019, 23:59

Functional Programming and Verification

Sheet 9

The deadline for this exercise sheet is Monday night 30.12.2019.

Tutorial Exercises

Exercise T9.1 Nonempty Lists

When working with lists, there are often situations in which one needs to make sure that they contain at least one element. When trying to access the head of a list, for instance, one could wrap the call to `head` like so:

```
if length xs > 0 then
  ... head xs ...
else
  error "something went utterly wrong"
```

This approach still causes run time errors when the programmer makes a mistake, however. In Haskell, one uses the type system to catch these errors at *compile time* by constructing a custom data type that simply does not include invalid values.

1. Define a data type `NonEmptyList`, which represents a list that contains at least one element.
2. Write conversions between `[a]` and `NonEmptyList a`.

   ```
   fromList :: [a] -> Maybe (NonEmptyList a)
   toList :: NonEmptyList a -> [a]
   ```

3. Implement the functions `nHead`, `nTail` and `nAppend` in analogy to `head`, `tail` and `(++)`.
4. Write a function `nTake :: Integer -> NonEmptyList a -> Maybe (NonEmptyList a)` that takes the first n elements of a non-empty list. If the list does not contain enough elements, `nTake` should return `Nothing`.

Note that your definitions need to fulfill the following criteria:

- They may not use library functions.
- They may not cause runtime errors or loop indefinitely for any inputs.

Exercise T9.2 (Mirror . Mirror) On the Wall = On the Wall

Given the following definitions

```
data Tree a = Leaf | Node (Tree a) a (Tree a)

mirror Leaf = Leaf
mirror (Node l v r) = Node (mirror r) v (mirror l)

id x = x

(f . g) x = f (g x)
```

show that `mirror . mirror = id`. You will have to use structural induction on trees.

Exercise T9.3 Sounds Logical, Am I Right?

You have already seen a data type of propositional formulas in the lecture. Moreover, you have seen a function to transform those formulas into negation normal form (NNF). In this exercise sheet, we change the definition of our formula data type such that one can only build formulas in NNF to begin with.

An *atom* is either a variable labelled by a string or the value "T" representing truth. A *literal* is a positive or negative atom. Finally, a *formula* is either a literal, a conjunction of formulas, or a disjunction of formulas.

1. Define data types for atoms, literals, and formulas. Make your definitions derive instances for `Eq` and `Show`.
2. Define values `top :: Literal` and `bottom :: Literal` representing truth and falsity, respectively.
3. A *clause* is a disjunction of literals. A formula is in *conjunctive normal form* (CNF) if it is a conjunction of clauses. We define the corresponding types

   ```
   type Clause = [Literal]
   type ConjForm = [Clause]
   ```

   By convention, an empty clause corresponds to falsity since it is the neutral element of the disjunction operator. Similarly, `[] :: ConjForm` corresponds to truth – as it is the neutral element of the conjunction operator.

   Define a function `conjToForm :: ConjForm -> Form` that transforms a formula given as a list of clauses to a formula as encoded in subtask 1. Examples:

   ```
   conjToForm [] = L top
   conjToForm [[]] = L bottom
   conjToForm [[], []] = L bottom :&: L bottom
   conjToForm [[Pos $ Var "v1", Neg $ Var "v2"], [Neg $ Var "v1", top]]
     = (L (Pos (Var "v1")) :|: L (Neg (Var "v2"))) :&:
       (L (Neg (Var "v1")) :|: L top)
   ```

   Hint: It might be useful to first define a function of type `Clause -> Form`.

4. (Optional) In the lecture, we defined the type of *valuations*: `type Valuation = [(Name,Bool)]`. Write a function `substConj :: Valuation -> ConjForm -> ConjForm` that replaces the variables of a formula by the values as specified in the passed valuation. Examples:

```
substConj [("v",True)] [[Neg $ Var "v"],[Pos $ Var "w"]]
  = [[bottom],[Pos (Var "w")]]
substConj [("v",False)] [[Neg $ Var "v"],[Pos $ Var "w"]]
  = [[top],[Pos (Var "w")]]
substConj [("v",True),("w",False)]
  [[Neg $ Var "v"],[Pos $ Var "w", Neg $ Var "w"]]
  = [[bottom],[bottom, top]]
```

Homework

You need to collect 5 out of 7 points (P) to pass this sheet.

Exercise H9.1 Sum Sum Sumsidumbibum! [2P]

Show that the `sumTree` function from T8.2 indeed works as expected, i.e. prove the equivalence

```
sum (inorder t) = sumTree t
```

using structural induction on trees.

Exercise H9.2 I Have It So SAT! [1: 1P, 2: 1P, 3: 1P, 4+5: 2P]

We extend our work from the tutorial. Our goal will be to build a *SAT-solver*. A SAT-solver takes a propositional formula F and decides whether F is satisfiable by some valuation or not. You have already seen a simple brut-force SAT-solver in the lecture. In this exercise, we will create a more efficient version. We provide you with some QuickCheck generators in a separate file on moodle that you can use to test your functions (or to stress test your competition submission).

Trivia

Though, in general, it is rather expensive to decide whether a propositional formula is satisfiable (unless **P=NP**), modern SAT-solvers are amazingly fast for many problem instances and were even used to solve longstanding mathematical conjectures.

Just 3 years ago, Marijn Heule used a SAT-solver to answer the boolean Pythagorean triples problem with a mind-boggling 200 terabytes proof: it is not possible to colour each of the positive integers either red or blue so that no three $a, b, c \in \mathbb{N}_+$ satisfying $a^2 + b^2 = c^2$ are all the same colour.

We begin by writing some utility functions.

1. Write a function `simpConj :: ConjForm -> ConjForm` that simplifies a formula F in CNF in the following way:
    a) For every clause C in F containing `top`, simplify C to `[top]`.
    b) For every clause C in F, remove every occurrence of `bottom`.
    c) Remove every clause C in F that has been simplified to `[top]`.
    d) If F contains a clause C that has been simplified to the empty clause, simplify F to `[[]]`.

   Examples:

   ```
   simpConj [[top], [top, Pos $ Var "v"]] = []
   simpConj [[bottom], [Neg $ Var "v"]] = [[]]
   simpConj [[Neg $ Var "v"], [bottom, Neg $ Var "v"]]
     = [[Neg (Var "v")],[Neg (Var "v")]]
   ```

```
simpConj [[Pos $ Var "v", bottom], [Pos $ Var "v", top]]
  = [[Pos (Var "v")]]
simpConj [[Neg $ Var "v"],[Neg $ Var "v"],
          [Pos $ Var "w", Neg $ Var "w"]]
  = [[Neg (Var "v")],[Neg (Var "v")],[Pos (Var "w"),Neg (Var "w")]]
```

2. Write a function `cnf :: Form -> ConjForm` that transforms a formula into CNF. Note that if $\phi = \phi_1 : \& : \cdots : \& : \phi_n$ and $\psi = \psi_1 : \& : \cdots : \& : \psi_m$ are in CNF, then the CNF of $\psi : | : \phi$ can be obtained by computing

$$\begin{aligned} &(\phi_1 : | : \psi_1) : \& : \cdots : \& : (\phi_1 : | : \psi_m) \\ : \& : &(\phi_2 : | : \psi_1) : \& : \cdots : \& : (\phi_2 : | : \psi_m) \\ &\vdots \\ : \& : &(\phi_n : | : \psi_1) : \& : \cdots : \& : (\phi_n : | : \psi_m) \end{aligned}$$

Make sure that, for every `cf :: ConjForm`, your functions satisfy the following property: `simpConj cf == simpConj $ cnf $ conjToForm cf`. Example:

```
let (a,b,c,d) = (Pos (Var "a"), Pos (Var "b"),
  Pos (Var "c"), Pos (Var "d"))
cnf $ (L a :&: L b) :|: (L c :&: L d)
  = [[a, c], [a, d], [b, c], [b, d]]
```

3. A typical, simple implementation of a SAT-solver for formulas in CNF can be given as follows:

Algorithm 1: Simple SAT-solver for formulas in CNF

```
1 Function sat:
     Input  : Formula $f$ in CNF
     Output: True if $f$ is satisfiable and False otherwise
2    let $f' = $ simplify $f$ in          -- as described in H9.3.1
3    if $f'$ contains no clauses then True
4    else if $f'$ contains the empty clause then False
5    else
6        let $v$ be a variable in $f$ and $b$ be a boolean value in
7        sat (subst $v$ $b$ $f$) or sat (subst $v$ (not $b$) $f$)      -- subst as described in T9.3.4
8    end
```

The efficiency of this SAT-solver greatly depends on its *selection strategy* for variables (cf. line 5 above). Such a strategy returns a variable v contained in f and a boolean value to substitute for v. In the best case, assigning v to b in f keeps f satisfiable (if it is satisfiable to begin with). If not, the algorithm has to backtrack and substitute the negation of b for v in f.

Write a selection strategy `selectV :: ConjForm -> Maybe (Name, Bool)` such that `selectV f` returns a variable v occurring in f and a boolean value b to assign to v. If f contains no variables, `Nothing` must be returned.

Note: Your strategy does not have to be sophisticated, but of course, we encourage you to implement some good heuristics.

4. (**Competition**) Next, implement a function

   ```
   satConj :: ConjForm -> Maybe Valuation
   ```

   that takes a formula f in CNF and returns an assignment satisfying f if such an assignment exists and `Nothing` otherwise. You might want to orientate yourself by the pseudo-algorithm given above, which can easily be extended to also return a valuation in case of a positive result.

   It suffices to return a valuation `val` such that `simpConj $ substConj val f = []`. Moreover, you can assume that all variables are labelled by integers n with $1 \leq n \leq u$ for some `u :: Int`.

   For the competition, the MC Jr will evaluate your solver against different kind of problem instances, increasing the size of the instance in each step if needed. Some of the formulas that he will generate are, for example:

   a) Formulas containing at least one positive literal for each clause
   b) Formulas containing variables that occur strictly positively or negatively
   c) Formulas where certain variables occur very frequently
   d) Formulas containing very short and very long clauses
   e) Formulas that have at most one positive literal for each clause (so-called Horn clauses)
   f) Formulas that are completely random

   For each class, the MC Jr creates a ranking and assigns a weight to the given class. The list above is ranked by importance, that is winning run a) gives you the fewest points, running run f) the most. However, the list may be incomplete and further classes of formulas will be tested.

   Some hints to optimise your implementation:

   a) Search for "DPLL" to read about simple but efficient selection strategies.
   b) List are simple but can be slow (e.g. for lookups) – use efficient data structures.[1]
   c) Optimise your pre-processing and simplification steps.
   d) Tune your selection strategy. For example, prioritisation of short clauses and frequently occuring variables can be useful.
   e) Check out conflict-driven clause learning (CDCL) for an improvement of DPLL (this will require some more work).

[1] But first makes sure that they are allowed by the submission system here

Please leave some comments for the MC Jr in order to understand your code. He is looking forward to all your submissions!

Important: If you submit a competition exercise, you agree that we are allowed to publish your name as part of the competition on our website. If you just want to submit a competition exercise as part of your homework without taking part in the competition, you can just remove the `{-WETT-}`...`{-TTEW-}` comments of your submission.

5. Lastly, define a function `sat :: Form -> Maybe Valuation` that solves the satisfiability problem for an arbitrary formula.

The best defense against logic is ignorance.

— Blaise Pascal