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

WS 2020/21
08.01.2021
Deadline: 18.01.2021, 23:59

Functional Programming and Verification

Sheet 9

Exercise T9.1 But wait, there is more!

It is common that one faces a situation in which one needs to make sure that a given list contains 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"
```

However, this approach causes run time errors when the programmer makes a mistake and passes an empty list to given code fragment. In Haskell, one can use the type system to catch these errors at *compile time*: we can construct a custom data type that simply does not include invalid values.

a) Define a data type `NonEmptyList`, which represents a list that contains at least one element.

b) Write conversions between `[a]` and `NonEmptyList a`.

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

c) Implement the functions `nHead`, `nTail`, and `nAppend` in analogy to `head`, `tail`, and `(++)`.

d) 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 or is called with a non-positive value, `nTake` should return `Nothing`.

Your definitions need to fulfill the following criteria:

- They may not use library functions (note that non-empty lists are already included in base).
- 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 Logical Foreplay

You have already seen a data type for propositional formulas in the lecture. However, these formulas are rather complicated to work with as they allow for arbitrary nesting of logical operators.

In this exercise, we again use Haskell's type system to fix this problem. We define a new data type that only allows one to create propositional formulas in conjunctive normal form (CNF). Such Formulas are particularly useful for automated theorem proving purposes as you will see in the homework.

a) Create data types or type aliases for the following definitions. Make use of lists where sensible.

   a) A variable *name* is a string.

   b) We have two *polarities*: positive and negative.

   c) A *literal* combines a polarity and a variable.

   d) A *clause* is a (possibly empty) disjunction of literals.

   e) A formula in *conjunctive normal form* is a (possibly empty) conjunction of clauses.

   Make your data type definitions derive instances for `Eq`.

b) Create `Show` instances for polarities and literals such that a negative literal $\neg a$ is transformed to `"~a"` and a positive literal a to `"a"`.

c) Implement the following utility functions:

   a) `lName` returning the name of a literal.

   b) `lPos` and `lNeg` transforming a variable to its corresponding positive and negative literal, respectively.

   c) The predicates `lIsPos` and `lIsNeg` returning `True` if and only if a given literal is positive and negative, respectively.

   d) `lNegate` negating a literal.

   e) `complements` such that ``l1 `complements` l2`` holds if and only if `l1` is the negation of `l2` (or vice versa).

d) We define the type of valuations as `type Valuation = [Name]`. In contrast to the lecture, we only save those variable names in our valuations that should be assigned to truth.

   Implement a function `eval :: Valuation -> ConjForm -> Bool` that evaluates a formula under a given valuation. Note that an empty clause corresponds to falsity since it is the neutral element of the disjunction operator. Similarly, an empty conjunction corresponds to truth as it is the neutral element of the conjunction operator.

Advice: create auxiliary functions that evaluate a literal and clause first.

e) Write a predicate `clauseIsTauto` that checks whether a given clause is a tautology without using `eval`.

Homework

You need to collect 9 out of 14 points (P) to collect a coin.

Exercise H9.1 Stop, Lemma Time! [5P]

Prove the lemma as stated in `h91.cthy` using *structural* induction.

Exercise H9.2 (Wettbewerb) Talkin' Bout A Resolution [c: 2P, e: 7P]

We are interested in the propositional satisfiability problem (SAT) that asks whether a given propositional formula is satisfiable. Indeed, you have already seen a simple brut-force approach to solve this problem in the lecture. However, the function there falls short in two categories:

1. It is very slow as it simply evaluates the formula under all possible valuations, answering positively if some satisfying valuation is found.
2. It returns no proof certificate, that is a piece of data that confirms its answer and is easy to check by external programs. Note that the function can readily be extended to return a satisfying valuation, that is a *model*, in case of a positive answer. But what about certificates for negative answers? How should such a certificate look like?

In this exercise, we solve the latter problem – and if you are motivated, also the former – using *resolution*.

Resolution is a very successful theorem proving method operating on formulas in CNF (cf tutorial exercise 9.3). Note that a clause can naturally be identified as a set of literals and a formula in CNF as a set of clauses. We will jump between both representations freely.

The core of propositional resolution is the following binary resolution rule:

$$\frac{L_1 \vee \cdots \vee L_n \vee A \qquad L'_1 \vee \cdots \vee L'_m \vee \neg A}{L_1 \vee \cdots \vee L_n \vee L'_1 \vee \cdots L'_m}\ (resolve)$$

The idea of the rule is as follows: Assume you are interested in finding a model M for the two premise clauses containing A and $\neg A$, respectively. If A is true, then $\neg A$ is false and hence one of $L'_1, \dots, L'_m$ must be true in M. Likewise, if $\neg A$ is true, then A is false and hence one of $L_1, \dots, L_n$ must be true in M. Now as either A or $\neg A$ must be true in M, we can conclude that also one of $L_1, \dots, L_n, L'_1, \dots, L'_m$ must be true in M.

Simply speaking, a resolution prover then applies this rule to a given set of clauses as many times as possible, potentially deriving a new clause in each step until either

1. the empty clause (falsity) is derived, and hence the original formula must be unsatisfiable, or
2. no new clause can be derived ("the set of clauses is saturated").

As a matter of fact, it is proven that in the second case, the orginal formula is indeed satisfiable and one can extract a model using the saturated set of clauses. You can trust us on this one or get started reading about it here.

In summary: a resolution prover can either return a list of resolution steps leading to the empty clause or a model for the formula. Both are easily checkable by external programs and hence good proof certificates. This solves the second of our aforementioned problems.

In practice, when saturating a set of clauses, one wants to avoid blindly re-checking all pairs of clauses in each iteration. One can do so by splitting the set of clauses into a processed set P and an unprocessed set U. In the beginning, all clauses are unprocessed. In each iteration, one then selects and removes a clause from U, only resolves it with clauses from P, and then adds the newly derived clauses to U and the selected clause to P. This way, one never checks whether two clauses can be resolved with each other more than once.

Putting all these thoughts together, we arrive at the following pseudo-code:

Algorithm 1: Simple propositional resolution-solver

```
Function resolution:
    Input  : Set of clauses $C$
    Output: Resolution proof if $C$ is unsatisfiable and a model otherwise
    $U = C$        -- unprocessed clauses
    $P = \emptyset$        -- processed clauses
    $R = []$        -- list of resolution steps
    while $U \neq \emptyset$ do
        $uc = \text{selClause}(U)$        -- select next best clause
        if $uc = \emptyset$ then return $R$        -- found the empty clause $\rightarrow$ return resolution steps
        $U = U \setminus \{uc\}$        -- remove selected clause
        $(NR, NU) = \text{resolvants}(uc, P)$   -- get resolvants & steps using $uc$ and clauses in $P$
        $R = \text{append}(R, NR)$        -- add new resolution steps
        $U = U \cup NU$        -- add new clauses
        $P = P \cup \{uc\}$        -- $uc$ is now processed
    end
    return extractModel($P$)
```

Okay, enough talking. Let's get serious and implement such a solver.

For the *Wettbewerb*, your goal of course is to submit the most efficient resolution solver. To give you a great deal of flexibility while avoiding duplicate efforts, you can change any of the used data structures in `Types.hs` provided that you adapt the corresponding mapping functions to our simple default representation. You can also change the behaviour and signature of most functions if they preserve the correctness of your algorithm.

For the tests and Wettbewerb benchmarks, you can also assume that all variables are labelled by integers n with $1 \leq n \leq u$ for some `u :: Int`. Further Wettbewerb-remarks can be found at the end of the sheet.

a) Implement the function `resolve` that takes a variable `n` and clauses `cp` and `cn` and resolves `cp` with `cn` on the variable `n` while assuming that `n` occurs positively in `cp` and negatively in `cn`.

b) In order to identify clauses in our resolution certificate R, we pair each clause with a unique, incremental key (see type `KeyClause`). Each clause in the initial formula C will be paired with keys $\{0, \ldots, \text{size}(C) - 1\}$, where $\text{size}(C)$ returns the number of clauses in C (without removing duplicates). A resolution step $r \in R$ then consists of the name of the resolved variable as well as the keys of the clauses containing the positive and negative occurence of the variable (in this order).

   Write a function `resolvants :: KeyClause -> KeyClause -> [(Resolve, Clause)]` that returns a list of all possible resolution steps and resulting clauses for two given clauses.

   Note that the function call resolvants(uc, P) in above pseudo-code just represents a call to `resolvants uc pc` for each `pc` in P.

c) Write a function `proofCheck :: ConjForm -> Proof -> Bool` that checks a proof against a given formula in CNF. Note that for resolution proofs, as stated above, each clause in the initial formula C is paired with keys $\{0, \ldots, \text{size}(C) - 1\}$ (in order as given by the passed list). Each resolution step then adds the next, corresponding key,clause-pair and increments the key.

d) Write a clause selection strategy `selClause :: SelClauseStrategy` that picks the next best clause from U. If U is empty, `Nothing` must be returned.

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

e) Implement `resolutionParam` that runs the resolution algorithm as sketched in the pseudo-code on a passed formula using the passed clause selection strategy. It should return a proof as well as all processed and unprocessed clauses (the latter is useful for debugging).

   You can use the function `extractModel` from the template to obtain a model from a (up to redundancy) saturated set of clauses.

   The more efficient your solver is, the more points you will be awarded.

For the Wettbewerb, the MC Jr will evaluate your solver as instantiated in `resolution` against different kind of problem instances, increasing the size of the instance in each step if needed. The focus will be on contradictory formulas as this is where resolution provers really shine.

The MC Jr also provides you with some QuickCheck generators that you can use to test your functions (or to stress test your competition submission). Feel free to share more generators and particularly interesting formulas on Zulip and discuss your performance on them with your peers. Think about formulas with with more than just a few hundred variables and clauses.

Here are some hints to optimise your implementation:

1. Lists are simple but can be slow (e.g. for lookups) – use efficient data structures.
2. Implement subsumption checking: a clause c is subsumed by a clause c' iff c' logically entails c. Similarly, a set of clauses C' subsumes a clause c if some clause $c' \in C'$ subsumes c. Note that subsumed clauses can safely be skipped and removed.
3. Tune your clause selection strategy.
4. Profile your program on large inputs. Here is a really nice video – part of an amazing video series done by one of our tutors – to get started.
5. Duckduckgo for more efficient resolution variants.
6. Duckduckgo "literal selection strategies".
7. Get a PhD in logic.

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

There can be no doubt that the knowledge of logic is of considerable practical importance for everyone who desires to think and infer correctly.

— Alfred Tarski