CS2002-Logic-Report

170011474

2019 March

1 Overview

This practical requires students to implement applications of laws of Boolean algebra and use them to find derivations of proofs of tautologies. Besides, the completeness of a set of laws will be considered and studied during this practical.

2 Part 1 Finding shortest proofs of tautologies

In `simplify.c`, there are 3 methods: `find_derivations_for_strings()`, `apply()`, and `min_deri()`. The `apply()` mainly covers applications of each logic rule and collect the smallest number needed to prove the tautology. Thus, critical and error-prone parts of implementation all come from `apply()` and they have been demonstrated below:

2.1 Design & Implementation

Consider every rewrite rule and apply it in each possible path. A for loop is adopted to iterate over each rule. For each rule, applicable paths will be found by using a set of search methods. The current rule will be applied on paths one by one until the searching methods return a "NULL", indicating there is no applicable path found. The difficult part is how to use searching methods correctly to find those possible paths. *Note that since we assign a new address to the pointer `cur_path`, the old address of it should also be freed.* The old address is easily ignored by programmers and the potential overwhelming usage of memory can occur.

```
int *cur_path = searches[i](expr_tree, path);
    while (cur_path != NULL) // keep searching for the next applicable path
    {
      int *temp_path = cur_path; // reserve the old address for next free()
      struct Expr *cur_expr = applies[i](cur, cur_path);
      int temp_res = apply(cur_expr, max_depth - 1, searches, applies, n_laws);
      ......
      cur_path = searches[i](expr_tree, cur_path);

      free_expr(cur_expr);
      free(temp_path); // the old path should be freed
    }
```

Recursively repeat applications. Once the rule has been applied, the resulting expression will be passed as an argument in another `apply()` to continue the recursion. *Note that max depth should be subtracted by 1 for every iteration. 6 is the starting number while 0 is the threshold.*

- **Base Case:** When the max_depth reaches 0 or when the expr is already proved to be **T**, then returns -1 or `6-max_depth` respectively.
- **Recursive Case:** For each iteration in `n_laws` and each possible path, each law will be applied and a `apply()` will be called.

Difficulty: How to collect the shortest proofs? The codes shown below solve this problem.

```
int temp_res = apply(cur_expr, max_depth - 1, searches, applies, n_laws); // the proof steps
      if (deri[i] == -1)
      {
        if (temp_res != -1) // when there is no valid proof yet, store the current valid proof
            directly
          deri[i] = temp_res;
      }
      else if (temp_res != -1 && temp_res < deri[i]) // when there is a valid proof, compare and
          store the smaller one
      {
        deri[i] = temp_res;
      }
```

There is an array storing the returned results called `int deri[n_laws]`. The index `i` refers to the corresponding law in the `law_searches`. That is, for `searchs[i]`, `applies[i]`, the shortest proofs will be stored in the `deri[i]`. However, for each law, there might be several applicable paths. We can only store the shortest one from them, which requires a comparison after application and before the storage process. Thus, above codes are intended to implement this function.
But another round of comparisons is required since we need to find the least number in `deri[]` so the temporary shortest proof can be returned to the previous call of `apply()`. The method `min_deri()` was implemented to select the smallest proof.

2.2 Testing

2.2.1 Stacscheck: 3/3 passed

```
stacscheck
Testing CS2002 Logic
- Looking for submission in a directory called 'LogicDir': Already in it!
* BUILD TEST - build-clean : pass
* BUILD TEST - part1/build : pass
* COMPARISON TEST - part1/prog-tautologies_part1.out : pass
3 out of 3 tests passed
```

2.2.2 Valgrind Memory Leak Check

```
==21648==
==21648== LEAK SUMMARY:
==21648==    definitely lost: 0 bytes in 0 blocks
==21648==    indirectly lost: 0 bytes in 0 blocks
==21648==      possibly lost: 0 bytes in 0 blocks
==21648==    still reachable: 120 bytes in 1 blocks
==21648==         suppressed: 0 bytes in 0 blocks
==21648==
==21648== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==21648== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

From the valgrind memory check tool, we can know that all blocks are freed. The only 120 reachable bytes come from the given code:

```
while (getline(&line, &len, stdin) != -1){...}
```

Since it does not matter, it can be safely ignored.

2.3 Evaluation

The full functionality of the standard specification satisfies requirements.

- All stacschecks have been passed.

- Codes are clear and simple to follow, with sufficient comments and explanation.
- There is no duplicate method.

3 Part 2 Adding rules

3.1 (a) Explain why there are no successful derivations for $T \vee a$, $\neg F$ and $\neg\neg T$ using the rewrite rules from Figure 3.

Since we cannot easily realise neither complementation backward nor absorption backward e.g.:

$$T \Rightarrow a \vee \neg a \textit{ compl disj backward}$$
$$a \Rightarrow a \wedge (a \vee b) \textit{ abs conj backward}$$

for an arbitrary a. Otherwise, $T \vee a$ can be proved as below:
$T \vee a = (a \vee \neg a) \vee a$ (compl backward) $= (a \vee a) \vee a$ (associativity) $= (a \vee (a \wedge (a \vee b))) \vee \neg a$ (absorp backward) $= a \vee \neg a$ (absorp) $= T$ (compl)
Due to the absence of double negation and negation of constant e.g.:

$$\neg\neg a = a$$
$$\neg F = T$$
$$\neg T = F$$

for an anbitrary a, $\neg F$ and $\neg\neg T$ cannot be proved to be tautologies.

3.2 (b) Determine a small number of additional rewrite rules, so that we are able to find successful derivations for the above three tautologies. Explain why you choose them.

There are only 3 rewrite rules which can help find successful derivations for the 3 tautologies:

1. $a \vee T = T$ *disjunctive domination*
2. $\neg\neg a = a$ *double negation*
3. $\neg F = T$ *negation of constant F (forward and backward)*

Reasons why I choose these 3: They can directly obtain successful derivations. Also, an interesting thing can be mentioned is that for $\neg F$, it can be turned to $\neg\neg T$ and then use double negation rule to turn it into T.

3.3 (d) Verify that we can now find successful derivations for the three derivations.

So for $T \vee a$, the result is 2; for $\neg F$, the result is 1; for $\neg\neg T$, the result is 1.

3.4 (e) Can you find more tautologies for which still no successful derivations exist? If so, add more rewrite rules to the code. Discuss this.

1. $\neg(a \wedge F)$ has no successful derivations. Add domination rule $a \wedge F = F$.
2. $T \wedge T$ has no successful derivations. Add idempotence conj forward rule. $A \wedge A = A$

4 Part 3 A sufficient set of rules

4.1 (a)

Rewrite rules needed:

1. The original 14 rewrite rules listed in the Part 1.
2. De Morgan rules (conjunctive and disjunctive with forward methods).

$$\neg(a \vee b) \Rightarrow \neg a \wedge \neg b \textit{ disj forward}$$
$$\neg(a \wedge b) \Rightarrow \neg a \vee \neg b \textit{ conj forward}$$

3. double negation
4. negation of constant
5. domination
6. idempotence conjunctive

How this rewriting (to CNF) is done

1. move $\neg$ in wards by *De Morgan* and *double negation laws.*
2. distribute $\vee$ over $\wedge$ by *distribute disj forward* and *distribute conj backward.*
3. get rid of superfluous T and F (while in this practical, we can safely ignore this process). But eventually, we need to get only one T, so use *idempotence law.*

How this rewriting (to T) is done (the details and proof are given in (d))

1. **rearrange:** commutativity, associativity.
2. **reduce:** idempotence, domination, absorption, complementation.
3. **combine Ts and Fs:** idempotence, negation of constant

4.2 (c) Running Result

4.3 (d) Describe your findings, proof

Phase 1: From tautology to CNF. For rewriting formulas into CNF, de morgan, double negation and domination are enough. If people want to eliminate $\neg F$ and obtain T instead, then negation of constant is good to be introduced. As we can see from procedures, the step 1 ensures that no $\neg$ outside the clause. Step 2 ensures that no $\vee$ will connect two clauses: since distribute laws will detect $\vee$ and if conditions are satisfied, $\wedge$ will replace $\vee$.
So here we can know that those 5 laws can easily convert any tautology to the CNF form.

Phase 2: From CNF to T. For a tautology in CNF form, we can mainly use the original 14 rewriting rules to convert it. Since all double $\neg$ has been simplified in the phase 1 (From tautologies to CNF), or been moved inwards brackets, so there will be much less situations requiring additional rules in this phase. For instance, if A represents an atom:

1. $\neg(A \vee B)$ will be considered as $\neg A \wedge \neg B$. By applying distribute laws or community rules, A will be combined with other As or $\neg A$s in phase 2, B will be combined with other Bs or $\neg B$s. Thus the T can easily be reached.
2. $\neg(A \vee \neg A)$ will be converted to $\neg A \wedge \neg\neg A$, and then $\neg A \wedge A$. When phase 2 comes, this clause will be transformed to T within mere 1 step.
3. $\neg(A \vee T)$ becomes$\neg A \wedge \neg T$. When phase 2 comes, $\neg T$ will be grouped with other T or F, and A will also be grouped by other As or $\neg A$s. Thus the T can easily be reached.
4. Likewise, $\neg(A \vee F)$ will have the similar development as the above one.

From this example, we can take a quick glimpse of the whole procedure from CNF to T: CNF provides us with convenient proof methods so that we can break the whole expression into several clauses. Afterwards, we can group some clauses together by using commutativity laws or others, then applicable laws will be applied to get a "local" T or F.

If one clause contains several atoms. For example, A and B. If the current expression is a tautology, there must be other clauses with A and B (or $\neg A$ or $\neg B$) inside. *Since it is a tautology, indicating that atoms will be eliminated eventually. Otherwise, T can never be achieved.* So the problem is: how can we eliminate them? The most straightforward way is to reduce them by laws. Here, we can look at laws listed above.

1. The first thing we can do is to **rearrange clauses**, to make it easier to find suitable patterns for laws. In this step, we can use *commutativity, associativity to change the order.* For example, here we reorder $(A \vee B \vee \neg A) \wedge (\neg A \vee A)$ to $(A \vee \neg A \vee B) \wedge (\neg A \vee A)$
2. The second thing we can do is to **reduce to T and F (or negatives).** Then we may find absorption laws, complementation laws, domination laws very useful here. Also distributivity assists this step. They can reduce the amount of clauses as well as the amount of atoms, simplifying the expression step by step. For example, we can use complementation then $(A \vee \neg A \vee B) \wedge (\neg A \vee A)$ to $(T \vee B) \wedge (T)$ to $T \wedge T$
3. repeat the 1, 2 until there are only Ts or $\neg F$s.
4. **The last thing is to reduce to only one T.** Since there is a negation of constant law inside this set, so we can convert $\neg F$ to T and the domination will reduce them again. There might be several Ts coexisting, so we can use idempotence conj forward rule to reduce them to one T. Given that if the current expression is a tautology, it means the occurrence of $T \wedge F$ is not possible (otherwise, it is not a tautology). Finally, the one T is achieved. For example, then $T \wedge T$ is reduced to T.

So here we can prove that every tautology in CNF forms can be rewritten as T.