1. Part 1 : Code Evaluation

1-1. TseitinTransformer

(1)

```
static @NotNull Set<Set<Literal>> toSetRepr(@NotNull Exp formula) {
    TseitinTransformer transformer = new TseitinTransformer();
    transformer.extensionVarIndex = 0;
    transformer.extensionVarIndex = formula.vars().size();
```

```
TseitinTransformer() {
    extensionVarIndex = 0;
    setOfClauses = new HashSet<>();
    variableList = new ArrayList<>();
    ExpOfVar = new HashMap<>();
}
```

함수 toSetRepr 안에서 extensionVarIndex를 0으로 초기화하는 의미없는 코드가 있었다. TseitinTransformer의 Constructor에서 이미 초기화를 진행하였기에 제거해야 한다.

(2) Variable을 반환하는 용도로 미리 만들어둔 list에서 받아오는 방식으로 구현했는데, 이 방식을 유지하기 위해서는 list의 값을 변경하는 코드가 필요하므로 side effect의 위험성이 있다. 또한 list 내의 모든 variable은 value가 true로 설정되기 때문에 variable return을 호출 시 true를 가진 값으로 return하도록 변경하였다.

=> variableList 제거, return new Literal(idx, true), Literal makeNewVar(boolean value) -> makeNewVar(void)

```
private Literal transform(Exp formula){
    if (formula instanceof Variable variable){
        return variableList.get(variable.identifier() - 1);
    }
```

```
for(int idx : formula.vars()){
    transformer.variableList.add(new Literal(idx,  value: true));
}
```

```
private Literal makeNewVar(boolean value){
    extensionVarIndex++;
    Literal newVariable = new Literal(extensionVarIndex, value);
    variableList.add(newVariable);
    return newVariable;
```

(3) ExpOfVar 을 참조하는 함수가 없어 변수를 제거하였다.

(4) 함수에서 주고받는 대상은 Literal 클래스이지만, 함수 이름에는 Variable이 사용된 경우가 있어 코드 이해에 혼동을 줄 수 있다고 판단하여 이름을 Literal으로 변경하였다.

makeNewVar -> makeNewLiteral, subExpVars -> subExpLiteral, produceSubExpressionVariable -> makeSubExpressionLiteral

(5) addClause 에서 `else return makeNewLiteral();` 다음 코드가 잘못되었음을 발견하였다. negation, conjunction, disjunction이 아닌 경우 exception을 throw하도록 변경하였다.

(6) 모든 clause를 대체하는 literal을 cnf에 추가하도록 하였다.

1-2. AbstractDPLLState

(1) propagate 중 assignedValue의 이름이 헷갈릴 여지가 있기에 totalAssignedValue 로 바꿈

1-3. DPLLSolver

(1)

```
while(!state.getRemainingVars().isEmpty() && !state.isSatisfiable()){
    if(!state.propagate()){
        if(state.isInconsistent()){
            if(state.getDecisionLevel()==0) return Optional.empty();
            state.backtrack();
        }
        else{
            state.decide(state.getRemainingVars().iterator().next(), value: true);
        }
    }
    else{
        if(state.getRemainingVars().isEmpty()){
            if(state.isInconsistent()){
                if(state.getDecisionLevel()==0) return Optional.empty();
                state.backtrack();
            }
        }
```

Inconsistent 확인을 propagate가 성공한 후에 실행하도록 변경하였다.

(2) 중첩된 if 문을 하나의 조건문을 확인하도록 변경하였다.

1-4. AdvancedDPLLState

(1)

```java
@Override
public void backtrack() {
    checkBackTrackValid();
    Set<Literal> conflictClause = findConflictClause();
    int lastDecisionVar = 0;
    int lastDecisionLevel = 0;
    for(Literal literal : conflictClause){
        int decisionLevelOfLiteral = getDecisionLevelOfVariable(literal.identifier());
        if (decisionLevelOfLiteral > lastDecisionLevel) {
            lastDecisionVar = literal.identifier();
        }
        lastDecisionLevel = getDecisionLevelOfVariable(lastDecisionVar);
    }
    removeVariables();
    propagateVarList = new ArrayList<>();
    changeValueAndAssign(lastDecisionVar);
```

길어진 코드를 알아보기 쉽도록 getLastDecisionVar 으로 변경하였다.

2. Part 2 : Code Refactoring

2-1. TseitinTransformer

(2) cnf 안의 Literal의 값이 변하지 않도록 하였고, 서로 다른 clause 안의 같은 identifier을 가진 literal 이라도 다른 개체로 취급되기 때문에 side effect의 위험성이 줄어들었다.

(4) 이름을 Literal으로 바꾸어 formula의 variable class와 혼동하지 않게 되어 코드 해석이 더 쉬워졌다.

(5) addClause 에서 `else return makeNewLiteral();` 다음 코드가 잘못되었음을 발견하였다. negation, conjunction, disjunction이 아닌 경우 exception을 throw하도록 변경하였다.

refactor 이전에는 negation, conjunction, disjunction인 경우에 적절한 return을 실행해주었기에 위의 코드가 발동하지 않았지만, 위의 3가지가 아닌 formula가 추가될 경우를 대비하여 exception을 throw하도록 바꾸었고, 이후에 코드를 추가할 때 예외 케이스를 더 쉽게 찾아낼 수 있을 것이다.

(6) 모든 clause를 대체하는 literal을 cnf에 추가하도록 하였다.

DPLL solver에서 satisfiable하지 않은 formula를 satisfiable 하다고 판단하는 경우가 자주 발생했다. 이는 코드 구현상의 문제로, 주어진 formula 전체를 치환하는 마지막 literal이 만들어지도록 구현하였고 이 literal의 값이 true가 되어야 satisfiable 한데, 해당 literal을 cnf에 추가하지 않아 마지막 literal의 값이 false 인 채로 cnf가 satisfiable 한 경우가 생겼다. 따라서 오류를 발생시키지 않도록 마지막 literal 혼자 존재하는 set을 추가하였다.

2-2. AbstractDPLLState

(1) totalAssignedValue 로 바꾸어 clause 중 assigned된 variable들의 value임을 확인하기 쉬워졌다.

2-3. DPLLSolver

(1) Inconsistent 확인을 propagate가 성공한 후에 실행하도록 변경하였다. propagate에 성공하고, remaining variable이 없을 때 cnf가 satisfiable 하지 않음에도 함수가 종료되는 경우가 발생하여 propagate 성공한 후에 satisfiability를 확인도록 변경하였다.

(2) 중첩된 if 문을 하나의 조건문을 확인하도록 변경하였다. 이로 인해 코드를 읽기 더욱 쉬워졌다.

2-4. AdvancedDPLLState

(1) 길어진 코드를 알아보기 쉽도록 getLastDecisionVar 으로 변경하였다. 따라서 코드를 읽기 더욱 쉬워졌다.