Implementation of a solver for the union of the theories of equality, lists, and arrays

Riccardo Fidanza
VR516130

February 2025

1 Introduction

This project presents a prototype solver that determines the satisfiability of a set of literals in the union of the quantifier-free fragments of three theories.

- **Equality with free symbols**
- **Non-empty, possibly cyclic lists**
- **Arrays without extensionality**

The concept of the congruence closure algorithm is derived from the course textbook, The Calculus of Computation.
The core of the solver is the congruence closure algorithm applied to DAGs.

This report describes the implementation of the solver, focusing on the main classes and the design pattern used, along with its advantages. It also covers the syntax supported by the solver, the testing process, and an analysis of the results.

2 Implementation Details

The project was implemented in Java, which aligns with my preference for the language. Java's strong object-oriented principles facilitate a well-structured and maintainable design, ensuring the project's components remain functionally distinct and easy to manage.

2.1 Interface TheorySolver

```
public interface Solver {
    boolean solve(String formula);
    void setForbiddenListHToFalse();
    void setForbiddenListHToTrue();
}
```

setForbiddenListHToTrue, setForbiddenListHToFalse

This methods are used to set the parameter of the forbidden list heuristic to true or false. By default, the theory solver uses the forbidden list, but this method allows us to decide whether to use this heuristic.

Solve

This method is used to solve the input formula within the current theory.

2.1.1 Class EqualitySolver

The **EqualitySolver** class implements the `TheorySolver` interface, within the theory of equality. It begins by processing the formula using utility functions to remove quantifiers and rewrite predicates. Then, it initializes a directed acyclic graph (DAG) using the extracted subterm set from the formula. Next, it constructs a DAG based on the extracted subterm set from the formula. The DAG is then utilized to implement the congruence closure algorithm, leveraging its structure and provided methods to determine satisfiability. If the forbidden list heuristic is enabled, the necessary information is passed to the DAG, allowing it to handle constraints accordingly.

2.1.2 Class ListSolver

The class implements the `TheorySolver` interface. It preprocesses the formula by removing quantifiers, rewriting predicates, and substituting negated atomic predicates (`-atom`) with a structured cons function. The DAG is then constructed from extracted subterms, and equality rules are inferred to merge equivalent nodes. Additionally, special handling is applied for cons terms, introducing `car` and `cdr` nodes to maintain structural integrity. The congruence closure algorithm is applied, followed by checks for equality and disequality constraints to determine satisfiability. Finally, atomic terms are validated against the DAG to verify that no contradictions arise. If all constraints are satisfied, the method returns true; otherwise, it returns false, indicating an unsatisfiable formula.

2.1.3 Class ArraySolver

The `ArraySolver` class implements the `TheorySolver` interface and provides methods for solving logical formulas, particularly those involving array-like structures, such as those with `select` and `store` operations. The class begins by processing a formula using a `rewriteFormula` method that either rewrites it directly or handles formulas containing `select` and `store`. It handles these operations by breaking down the formula into parts, extracting and manipulating terms (such as handling `select-store` patterns), and then recursively rewriting the formula. If the formula does not contain a `store`, it replaces `select` operations with a `f_array(index)` format.

The solver applies congruence closure algorithms using a DAG to process the formula. The `solve` method attempts to check the satisfiability of the formula by repeatedly calling a helper function, `solvePartial`, for each rewritten formula. It checks if all constraints are satisfied and returns `true` if the formula is satisfiable (SAT), or `false` if not (UNSAT).

2.2 Class DAG

The `Dag` class represents a Directed Acyclic Graph (DAG) that is a key component for performing congruence closure for solving logical formulas with terms and their equivalences. The DAG in this implementation is built around nodes, where each node represents a term.

The Dag class is initialized using a set of terms and constructs a list of nodes that represent these terms, with the help of the `DAGBuilder` class that is responsible for transforming a set of function terms and a formula into a DAG.

The DAG supports two key operations, `find` and `union`, which are fundamental to the congruence closure algorithm. The `find` operation returns the equivalence class of a node based on the input ID, using a non-recursive approach. The `union` operation merges two equivalence classes by selecting the class with the largest `ccpar` as representative for the new combined class. After choosing the representative, the `find` field of all nodes in the merged classes is updated so that nodes from the original classes now point to the new representative.

An important aspect of the DAG is the handling of the forbidden list, which is used to check disequality constraints. The forbidden list keeps track of pairs of terms that should never be merged. If an attempt is made to merge two terms that are in the forbidden list, the operation is aborted, and the system recognizes the formula as unsatisfiable.

The `merge` method is the core operation in the congruence closure algorithm. It recursively merges equivalence classes. It compares all pairs of terms from the two parent sets and recursively attempts to merge

them if they are congruent.

The `congruent` method is used to check whether two nodes are congruent. This check ensures that only terms with identical structures are merged, which is essential for maintaining the correctness of the congruence closure.

2.3 Class Solver - Strategy Pattern

The `Solver` class follows the Strategy Pattern by delegating the solving logic to a `TheorySolver` instance. The core concept of this pattern is to define a family of solvers for different theories, encapsulate each solver in its own class, and make them interchangeable. This design allows us to easily extend the project with additional strategies and switch between them as needed, enabling flexibility in choosing and changing strategies on the fly.

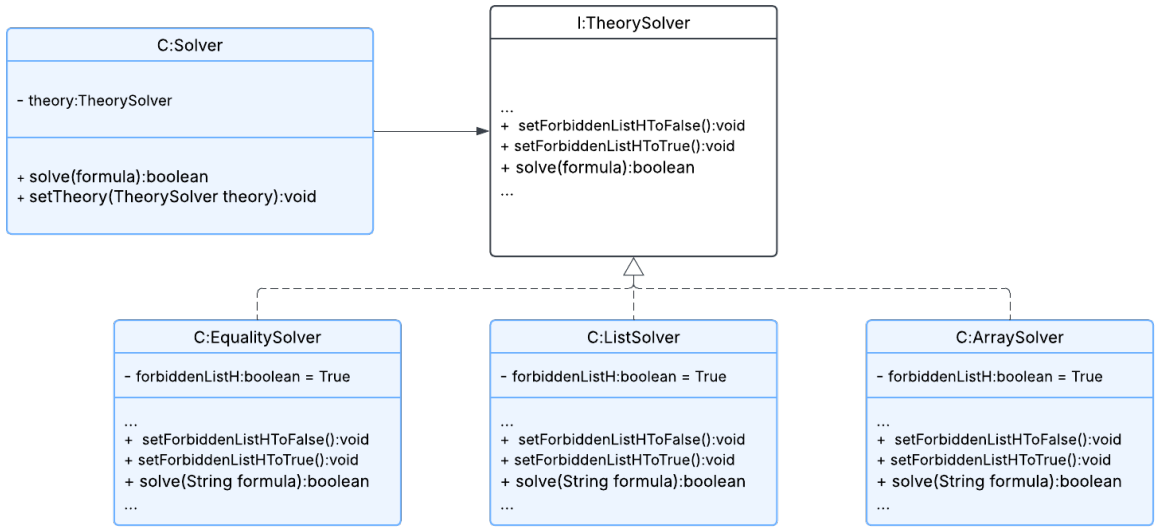

Figure 1: Strategy pattern

Usage Example

```
Solver solver = new Solver();
solver.setTheory(new EqualitySolver());
boolean res = solver.solve(formula);
solver.setTheory(new ArraySolver());
res = solver.solve(ArrayFormula);
```

2.4 Static Class SATUtils

This is a static class that provides utility functions for processing logical formulas. It includes methods for extracting subterms from a given formula (`extractSubterms`), extracting equality (`extractERules`) and disequality (`extractDRules`) rules from formulas expressed in Disjunctive Normal Form (DNF). The `rewritePredicate` method reformulates predicates by standardizing their format, ensuring that they are properly expressed as either equalities or negations. Additionally, the `dropQuantifier` method removes universal ($\forall$) and existential ($\exists$) quantifiers from logical expressions.
This class makes extensive use of string manipulation and regular expressions and our solver uses it for preprocessing.

3 Syntax

Formulas in input are in DNF. The syntax accepted by this solver:

- `"-"` represents negation (`not`)
- `"&"` represents conjunction ($\wedge$)
- `"|"` represents disjunction ($\vee$)
- `"="` represents equality (`=`)
- `"!"` represents inequality (`!=`)
- `"forall"` represents ($\forall$)
- `"exists"` represents ($\exists$)
- The quantified variable is enclosed in square brackets: `forall[x]`.
- The scope of the quantifier is represented by curly braces: `forall[x]{}`.
- T represents `True`

Example

$$f(g(x)) = g(f(x)) \ \& \ f(g(f(y))) = x \ \& \ f(y) = x \ \& \ g(f(x))!x$$

4 Test

The solver is tested on three input files (one focused on theory), each containing multiple formulas. The results, including satisfiability, execution time, and memory usage for each formula, are recorded in a dedicated output file.

5 Results and Analysis

These are the outputs of the formulas with the heuristic applied. We can now compare the time and memory usage with and without the Forbidden List.

The formulas from the Array theory were used for these graphs.

As we can see, memory consumption remains unchanged, but the time without the heuristic is higher. This is because the Forbidden List quickly identifies when a formula is unsatisfiable, leading to less time.

Formula	Result	Execution Time	Memory Used
$f(a,b) = a \,\&\, f(f(a,b),b) \,!\, a$	UNSAT	49.3458 ms	0.239746 MB
$f(f(f(a))) = a \,\&\, f(f(f(f(f(a))))) = a \,\&\, f(a) \,!\, a$	UNSAT	5.2422 ms	0.0 MB
$f(x,y) = f(y,x) \,\&\, f(a,y) \,!\, f(y,a)$	SAT	3.4347 ms	0.0 MB
$f(g(x)) = g(f(x)) \,\&\, f(g(f(y))) = x \,\&\, f(y) = x \,\&\, g(f(x)) \,!\, x$	UNSAT	4.7234 ms	0.0 MB
$f(f(f(a))) = f(f(a)) \,\&\, f(f(f(f(a)))) = a \,\&\, f(a) \,!\, a$	UNSAT	7.4277 ms	0.0 MB
$f(f(f(a))) = f(a) \,\&\, f(f(a)) = a \,\&\, f(a) \,!\, a$	SAT	3.0471 ms	0.440025 MB
$p(x) \,\&\, f(f(x)) = x \,\&\, f(f(f(x))) = x \,\&\, -p(f(x))$	UNSAT	15.6782 ms	0.0 MB

Figure 2: Theory of equality with heuristic

Formula	Result	Execution Time	Memory Used
$i1 = j \,\&\, i1 \,!\, i2 \,\&\, select(a,j) = v1 \,\&\, select(store(store(a,i1,v1),i2,v2),j) \,!\, select(a,j)$	UNSAT	94.838 ms	1.130462 MB
$select(store(a,i,e),j) = e \,\&\, i \,!\, j$	SAT	4.945 ms	0.0 MB
$select(store(a,i,e),j) = e \,\&\, select(a,j) \,!\, e$	SAT	3.8529 ms	0.0 MB
$select(store(a,i,e),j) = e \,\&\, i = j \,\&\, select(a,j) \,!\, e$	SAT	4.5481 ms	0.0 MB
$select(store(store(a,j,f),i,e),k) = g \,\&\, j \,!\, k \,\&\, i = j \,\&\, select(a,k) \,!\, g$	UNSAT	16.9877 ms	0.440132 MB
$i1 = j \,\&\, select(a,j) = v1 \,\&\, select(store(store(a,i1,v1),i2,v2),j) \,!\, select(a,j)$	SAT	5.1833 ms	0.0 MB

Figure 3: Theory of Array with heuristic

Formula	Result	Execution Time	Memory Used
$car(x) = car(y) \,\&\, cdr(x) = cdr(y) \,\&\, f(x) \,!\, f(y) \,\&\, -atom(x) \,\&\, -atom(y)$	UNSAT	77.2306 ms	0.681724 MB
$car(x) = y \,\&\, cdr(x) = z \,\&\, x \,!\, cons(y,z)$	SAT	3.8036 ms	0.0 MB
$-atom(x) \,\&\, car(x) = y \,\&\, cdr(x) = z \,\&\, x \,!\, cons(y,z)$	UNSAT	10.0759 ms	0.0 MB
$atom(x) \,\&\, -atom(x)$	UNSAT	5.1037 ms	0.440025 MB
$atom(x) \,\&\, cons(x,y) = x$	UNSAT	3.1323 ms	0.0 MB

Figure 4: Theory of list with heuristic

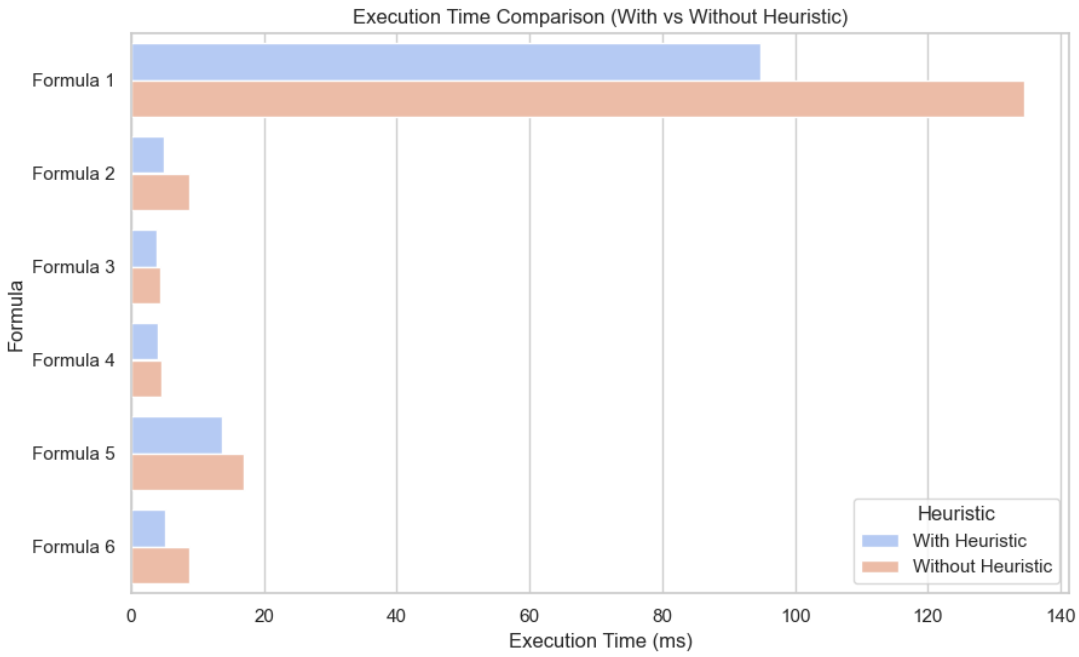

Figure 5: Comparing Execution time

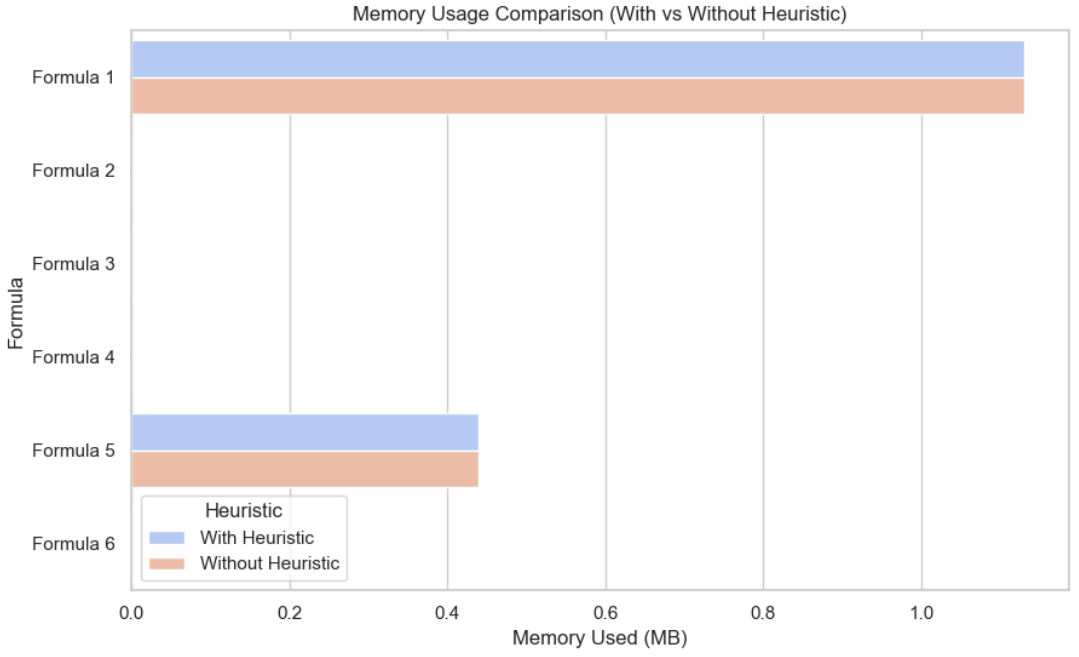

Figure 6: Comparing memory consumption