Санкт-Петербургский государственный университет

Го Том Ичжоу

Выпускная квалификационная работа

Congruence closure over interpreted symbols

Уровень образования: магистратура
Направление 01.04.01 “Математика”
Основная образовательная программа ВМ.5832.2021 “Современная математика”

Научный руководитель:
профессор факультета математики
и компьютерных наук, д.ф.-м.н.,
Петров Федор Владимирович

Рецензент:
доцент-исследователь
Государственной ключевой
лаборатории компьютерных наук
Института программного обеспечения
Китайской академии наук,
PhD, Бохуа Чжань

Санкт-Петербург
2023

On the implementation of congruence closure algorithm for Horn equations based on equation flattening

Yizhou Guo * †[1]

[1]Saint-Petersburg State University, Department of Mathematics and Computer Science

May 30, 2023

Abstract

We describe an implementation and testing of a congruence closure algorithm for Horn equation based on equation flattening in Python.

1 Introduction

In the quantifier-free theory of uninterpreted functions (QF_UF), in addition to logical operators, there are equalities between expressions and variables for uninterpreted functions. They satisfy the conditions of reflexivity, symmetry, transitivity, and congruence relation, the first three of which are the conditions defining an equivalence relation and the last of which is the property that for all expressions x, y and functions f, g in our domain of discourse,

$$x = y \wedge f = g \implies f(x) = g(y)$$

In SMT (satisfiability modulo theory), the aim is to determine whether a set of logical formulas containing equalities and inequalities over over uninterpreted functions is satisfiable. One can fittingly view SMT as SAT with the addition of uninterpreted functions and equalities. Thus, there is a direct correspondence between SMT and QF_UF. Moreover, SMT solvers can be applied directly to resolve problems of equality reasoning, wherein one is to determine whether a set of equalities and inequalities over uninterpreted functions is satisfiable.

Equality reasoning arises in many applications including compiler optimization, functional languages, and reasoning about databases as well as, most importantly, reasoning about different aspects of software and hardware. [Kap19]

*guoyiz@yandex.ru
†st095712@student.spbu.ru

For example, in symbolic execution, problem inputs, instead of processed as concrete input instances as happens upon actual execution of the program, a more abstracted execution is performed on a symbol for each input. As another example, in the theory of uninterpreted functions, a specific function is abstracted to an uninterpreted one, and the simplification therein is often useful for proof of equivalence of programs.

[Kap19] described an algorithm that computes congruence closure for a Horn equations, which an implication statements, wherein the antecedent is a conjunction of equalities and the consequent is a single equality.

Congruence closure and decision procedures for Horn equations is still a special case of QF_UF theory, since the case of Horn equations is a special case of logical formulas containing equalities. Hence, deciding satisfiability for Horn equations can be solved using the DPLL(T) algorithm. However, the DPLL(T) algorithm, designed for general cases, required multiple rounds of interaction between SAT solving and congruence closure, whereas Kapur's algorithm solves this using propagation only, without using SAT solving. In principle, this should be more efficient than the DPLL(T) algorithm for this particular class of problems.

This paper will describe the implementation and verification of such an algorithm in Python by the author (as well as more minor changes by Bohua Zhan).

2 Overview of algorithm

We begin some preliminary definitions and notation. We let F be a set of function symbols including constants (here, we note that a constant is fittingly seen as a 0-ary function) and $GT(F)$ be the ground terms constructed from F. From here on, we will simply use *constant* to mean a 0-ary function. Ever symbol $f \in F$ can be considered uninterpreted.

A Horn (conditional) equation is of the form $\{(\wedge h_1^i = h_2^i) \Rightarrow (c_1^i = c_2^i) | 1 \leq i \leq k\}$ is a conditional equation, wherein the h's and c's are ground terms from $T(F)$.

Below are the decision problems the algorithms of which were implemented by the author.

1. Congruence closure: which terms are equal according to the Horn equations?
2. (Extended) congruence closure: is an inputted Horn equation implied by the existing Horn equations?
3. Decision procedure: is an inputted set of Horn equations is satisfiable?

The most non-trivial and/or critical parts of the implementation as are follows

1. 3.1 of [Kap19] (Computing Constant Equivalence Closure from Constant Horn Equations) wherein only constants are considered.

2. 3.3 of [Kap19] (Extension to Nonconstant Function Symbols).

3. A clever means of encoding the flattening described in 2.2.2 of [Kap19] suggested to the author by Bohua Zhan which optimizes the introduction of extra symbols in the flattening of multiple equalities.

Now, we shall present some major ideas behind these algorithm. For details, we suggest that the reader refers to [Kap19].

2.1 Union-find

First, the main data structure is the disjoint set (which supports `union` and `find` both of which can be for practical purposes seen to run in constant time (in theory, its complexity involves the Ackermann function which grows extremely slowly)). On it, we also use path compression, which updates roots of all nodes on the path traversed when we go up the tree on input `u` to find its root.

In Python, it is as follows.

```
traversed_nodes = []
while u != self._parents[u]:
    traversed_nodes.append(u)
    u = self._parents[u]

# path compression technique
for name in traversed_nodes:
    former_root = self._parents[name]
    self._parents[name] = u
```

The disjoint union enables us to rewrite constant terms to the its root (which can be seen as representative of its equivalence class) in Horn equations. This operation is run whenever a new equivalence between constant terms is introduced and as an optimization, we persist a pointer from symbol to the Horn equations in which appears, so that we need not iterate thru all Horn equations.

2.2 Propagation

The idea is that the algorithm for constant cases coupled with flattening naturally induces the extended algorithm via propagating of equality to constants on the right hand side (RHS) of flat equations with the same flat term on the left hand side. As with the constant case, we use disjoint union and rewrite to root upon invocation of `union`, doing so in this case for flat equations. As more constant equivalences are introduced, flat terms are rewritten, which induces another constant equivalence per two flat terms now deemed to be identical per the disjoint union.

2.3 Flattening

As mentioned previously, flattening is a major idea behind this algorithm. Before going into details on flattening, we shall first talk about testing of a Horn

conjecture, the form of which is closely related. The hypotheses and consequent of Horn conjecture can be arbitrarily nested, and by nesting we mean the likes of $f(w, g(h(x), y, z)) = h(z)$ as an example of two layers. The idea is that in order to write this in an equivalent form wherein all equations are either flat equations or constant equations, we introduce new symbols for nested terms in a recursive manner.

To minimize the number of extra symbols reduced, we store mappings from newly introduced symbols to their corresponding expressions in a recursive manner via the following code.

```
class Replacement:
    def __init__(self):
        self.symbols : Dict[str, Expr] = dict()
        self.symbols_rev : Dict[Expr, str] = dict()
        self.counter = 0

    def add_symbol(self, name, expr):
        self.symbols[name] = expr
        self.symbols_rev[expr] = name

    def new_symbol(self):
        res = "_x" + str(self.counter)
        self.counter += 1
        return res

    def has_expr(self, expr: Expr) -> bool:
        return expr in self.symbols_rev

    def to_flat_eqs(self) -> "list[Equation]":
        return [Equation(func, Const(symb_name))
            for symb_name, func in self.symbols.items()]
```

A few points to be noted:

- There is a counter which is incremented upon introduction of a new symbol.
- `has_expr` is used to perform a lookup for an expression that is nested which we must rewrite to a symbol. If an equivalence expression has already been rewritten to a symbol, then we simply reused that symbol. An illustrative example in this case would be $f(g(x), g(x)) = c$. $g(x)$ that is the first argument of f is rewritten to introduced symbol `_x0` and then when it appears again as in the second argument, we simply look up in our cache and replace.

The code for actually flattening that uses replacement below (note especially the invocation of `has_expr`):

```
def flatten(self, repl: Replacement) -> Expr:
    new_args = list()
    for arg in self.args:
        if isinstance(arg, Func):
            new_args.append(arg.flatten(repl))
        else:
            new_args.append(arg)
    new_expr = Func(self.func_name, new_args)
    if repl.has_expr(new_expr):
        return Const(repl.symbols_rev[new_expr])
    else:
        new_name = repl.new_symbol()
        repl.add_symbol(new_name, new_expr)
        return Const(new_name)
```

`to_flat_eqs` in `Replacement` is used to flatten a list of equations, with the cache of expression to rewrite symbol persisted throughout. By this we mean that if we have $f_0(g(x)) = c$ and then $f_1(g(x)) = c$, then with

```
def flatten_equations(equations: list[Equation],
    repl: Replacement=None)
        -> Tuple[list[Equation], Replacement]:
    if repl is None:
        repl = Replacement()
    return [eq.flatten(repl) for eq in equations], repl
```

we would upon the first invocation on $f_0(g(x)) = c$, the introduced symbol `_x0 = g(x)`, which is reflected by the update of `Replacement` object which is passed in to our call on $f_1(g(x)) = c$. `flatten_equations` should return in this case `[f_0(_x0) = c, f_0(_x0) = c]` along with `[_x0 -> g(x)]` encoded in `repl`.

2.4 Pointers to Horn equations

If there is clause $a_1 = a_2 \implies c_1 = c_2$, union of a_1, a_2 should trigger the union of c_1, c_2. Similar holds for Horn clauses with multiple antecedents. From this arises the subproblem of propagating to the hypotheses and determining when the they are all satisfied. This is done by persisting both a mapping from symbols set to Horn equations in which of the given symbol appears thru which we can efficiently also persist counter of number of satisfied antecedents of each Horn clause.

For details on this, see the explanation of the example given in **??** as well as the appendix of [Kap19].

3 Concrete example cases

If the reader wants to understand the algorithm in more detail, we suggest going thru the theoretical explanation guided by examples. The author has the intention of supplementing the more theoretically presented [Kap19] with concrete cases and actual implementation.

3.1 One of constant Horn equations

Example 3.1.

- Let $F = \{c_0, c_1, c_2, c_3, c_4, c_5\}$.
- Let our constant equivalences be $\{c_0 = c_1, c_1 = c_2, c_3 = c_4\}$, which induces the equivalence classes $\{0, 1, 2\}, \{3, 4\}, \{5\}$.
- Let our only Horn equation be $c_0 = c_1 \wedge c_0 = c_2 \wedge c_1 = c_1 \implies c_4 = c_5$.
- The preprocessing of this Horn equation will all the antecedent inequalities trivial, thereby reducing it to $\implies c_4 = c_5$, which is simply $c_4 = c_5$.
- Propagation of the consequent of this Horn equation to the extant equivalence relation will result in the merging of $\{3, 4\}$ and $\{5\}$.
- The resulting equivalence classes are then $\{0, 1, 2\}$ and $\{3, 4, 5\}$.

3.2 Propagation of newly added equalities by example

Here, we will illustrate the propagation part of the algorithm described in 3.1 of [Kap19].

Symbol set is $\{c_0, c_1, c_2, c_3, c_4\}$.

Horn equations:

- $c_4 = c_2 \wedge c_3 = c_4 \wedge c_0 = c_1 \implies c_1 = c_4$
- $c_1 = c_0 \wedge c_4 = c_2 \wedge c_0 = c_0 \implies c_0 = c_0$
- $c_0 = c_2 \wedge c_1 = c_2 \implies c_0 = c_4$
- $c_2 = c_2 \implies c_0 = c_1$
- $c_2 = c_4 \implies c_1 = c_2$

Constant equations:

$$c_2 = c_3, c_2 = c_0, c_4 = c_4$$

induce equivalence classes $\{c_0, c_2, c_3\}, \{c_1\}, \{c_4\}$

Preprocessing via these equivalence classes results in

- $c_4 = c_0 \wedge c_0 = c_4 \wedge c_0 = c_1 \implies c_1 = c_4$
- $c_1 = c_0 \wedge c_4 = c_0 \implies c_0 = c_0$

- $c_1 = c_0 \implies c_0 = c_4$
- $\implies c_0 = c_1$
- $c_0 = c_4 \implies c_1 = c_2$

There is queue of equivalences to be propagated. It is initialized to be identical to the list of specified constant equations excepting trivial ones. In this case, it is $c_2 = c_3, c_2 = c_0$. When we propagate an equation $a = b$, in addition to performing the disjoint `union` operation, we must also modify `hyp_pointers` and `cone_pointers`, which map a symbol to the hypotheses and consequent equations in which they appear respectively. The idea behind this is that if each symbol appears sparsely among the Horn equation, then via this data structure we can minimize the number of lookups necessary for propagation of newly added equivalences.

Per

```
while len(self.q):
    eqn = self.q.popleft()
    self.union(eqn.lhs.name, eqn.rhs.name)
```

the major steps executed are the following:

- Queue: $c_2 = c_3, c_2 = c_0$
- Pop and union c_2, c_3. This results setting the root of c_2 to that of c_3.
- In updating pointers (here on c_2, c_3), we also detect for a trivial consequent, in which case the hypotheses are irrelevant. This result in deletion of $c_1 = c_0 \wedge c_4 = c_0 \implies c_0 = c_0$.
- Preprocessing resulted in $\implies c_0 = c_1$, which means the counter was initialized to 0. Add this to the queue.
- Queue: $c_2 = c_0, c_0 = c_1$
- Pop and union c_2, c_0. c_2 already has root c_3, root of c_0 is made to be c_3.
- Update pointers for this. A counter is decremented when an equation in the hypothesis is equivalent to newly introduced relation $c_2 = c_0$. There are no deletions of Horn equations or adding of new equalities to the queue though.
- Queue: $c_0 = c_1$
- Pop and union $c_0 = c_1$
- Counter of $c_4 = c_0 \wedge c_0 = c_4 \wedge c_0 = c_1 \implies c_1 = c_4$ is 3, since we are about to equate $c_0 = c_1$, we decrement the counter (the corresponding hypothesis is the last one).

- Decrement counter for $c_1 = c_0 \implies c_0 = c_4$. With the counter of this one now 0, we then add to queue $c_0 = c_4$. This Horn equation also gets deleted.
- Queue: $c_0 = c_4$
- Pop and union $c_0 = c_4$.
- $c_4 = c_0 \wedge c_0 = c_4 \wedge c_0 = c_1 \implies c_1 = c_4$ has counter 2
- With c_4 and c_0 now to be equated, the first two hypotheses hold. Thus we decrement twice, which results in $c_1 = c_4$
- Everything is equal.

3.3 Horn equations with functions

We now give an example to illustrate induction of an equivalence via equivalent f-terms on LHS.

Example 3.2.
- Let $F = \{c_0, c_1, c_2, c_3\}$.
- Our only constant equation is $c_2 = c_3$.
- The equivalence classes $\{c_0\}, \{c_1\}, \{c_2, c_3\}$.
- Our non-constant equations are $f(c_2) = c_0$ and $f(c_3) = c_1$.
- This induces $c_0 = c_1$.
- The resulting equivalences are then $\{c_0, c_1\}, \{c_2, c_3\}$.

Example 3.3.
- Introduce constant equivalence $c_1 = c_2$.
- Introduce flat equations $f(c_1) = a, f(c_2) = b, f(c_3) = d$. Then $a = b$ is induced.
- Introduce the Horn equation $a = b \implies c_2 = c_3$. Then $c_2 = c_3$ holds since $a = b$ holds.
- Now that we have deduced $c_2 = c_3$, we can use it to deduce $b = d$.

4 Verification of correctness

To verify correctness, we construct a corresponding SMT instance. What is a bit tricky about the SMT solver analogy is that the solver does not directly construct the congruence closure but rather it tests whether or not some equality is actually is in the closure. Say we have some equality E and it is in the closure. Then we cannot satisfy all equations (in generality, this consists of constant equations, flat equations, and Horn equations) as well as $\neg E$ is necessarily not

satisfied. On the other hand, if the equality E is not in the closure, then necessarily we can satisfy $\neg E$ in addition to all the prespecified conditions.

Moreover, to deal with Horn equations with consequents, we use the fact that an implication $A \implies B$ is equivalent to $\neg A \vee B$, which enables us to express it in terms of `Or` and `Not`.

In terms of actual implementation detail, we use the `z3` SMT solver library of Python.

5 Horn conjecture testing

We observe that the Horn conjecture testing that can be interpreted as follows. The hypotheses of the Horn conjectures are flattened and introduced. Their introduction results in preprocessing on Horn equations and possibly additional propagation of constant equalities. Constant symbols are then introduced for both the LHS and RHS of the consequent of the Horn conjecture via flattening. Equality of these constant symbols (under of the context of introduced hypotheses) is tested in order to determine whether or not the Horn conjecture is in the closure or not. We also note that in the Horn conjecture, a symbol not already in the constant equations, flat equations, and Horn equations can occur.

Below is code for handling introduction of a new symbol.

```
def introduce_new_symbol(self, symb_name: str) -> None:
    if symb_name not in self.symbol_names:
        self.symbol_names.add(symb_name)
        self.symbols.append(Const(symb_name))
        self._parents[symb_name] = symb_name
        self.hyp_pointers[symb_name] = set()
        self.cone_pointers[symb_name] = set()
        self._ranks[symb_name] = 1
        self.num_symbols += 1
```

Note especially that we must update also the node pointers.

6 Testing, debugging, and maintenance

Representative hand generated example cases were hard coded in Python and tested with hand calculated result. Those interested in this code should look at these examples to as a guide for both understanding of the algorithm and familiarization with the code.

Then there is code for generating random constant equations and Horn equations. They are then inputted to the algorithm of [Kap19] and the result is compared with that of the SMT solver. For the extended case (arbitrarily nested function application equivalences), it was run on 1000 randomly generated test cases. A seed was used to guarantee determinism, which of course much eases the debugging process. The number of random tests run can of course be changed

per the tradeoff of extent of verification of correctness with respect to execution time of the testing.

In implementation for instances of congruence closure computation, there are random and fixed subclasses of a base class. Needless to say, the fixed subclass gives the user more flexibility while the random subclass gives the power of generation of large number of tests. In the debugging process, the author also took randomly generated tests and hard coded the data into an object of the fixed subclass.

We shall conclude this subsection by describing the random generation in more detail.

6.1 Random instance generation

We begin with an explanation of the parameters in this following to snippet of code concerning both the size of the instance and types of the expressions and equations.

```
class RandomNonConstantEquations(NonConstantEquations):
    def __init__(self,
                 num_symbols: int,
                 num_const_eqs: int,
                 num_func_symbols: int,
                 num_flat_eqs: int,
                 num_horn_eqs: int,
                 func_arg_range: Tuple[int] = (1,5),
                 max_func_nesting: int = 2,
                 nest_prob: float = 0.5,
                 func_prob: float=0.33) -> None:
```

What the parameters

- `num_symbols`
- `num_const_eqs`
- `num_func_symbols`
- `num_flat_eqs`
- `num_horn_eqs`

actually represent are self-explanatory from their variable names. As for the remaining (optional) arguments, each of which are assigned a default:

- `func_arg_range`: the minimum and maximum possible arity of the function, from the corresponding range of which we randomly choose one.
- `max_func_nesting`: the maximum extent to which a function expression is nested. As an example, the nested level of $f(x, y)$ is 0, while the nesting level of $f(g(h(x, y), z), x)$ is 2.

- `flat_prob`: the probably that a randomly generated expression is actually function. (If not, it is necessarily constant)

To generate a random Horn equation, we also specify a range for the possible number of antecedents in the form of optional argument with default values. From that range, we select the number of equations to randomly generate.

```
def random_horn_equation(self,
                         min_antecedents=1,
                         max_antecedents=10):
    num_antecedents = random.randint(
        min_antecedents, max_antecedents)
    antecedents = [
        self._gen_rand_eq() for _ in range(num_antecedents)]
    consequent = self._gen_rand_eq()
    return HornEquation(antecedents, consequent)
```

7 Code in gitee repo

The code is now publicly available on Gitee[1]. The author much regards the substance of this thesis to be mostly in the code, and this document is meant much to lead the reader into the code or at least give the reader a sketch of that which is achieved by the code. We hope that this code will eventually be useful for researchers or even people in industry who deal with congruence closures, or at least educationally valuable.

8 Conclusion

In the future, given the infrastructure already developed, we can generate test cases of a fixed case and compare the runtimes of our implemented algorithm and the SMT solver. Moreover, we can extend to functions with commutativity/associativity/idempotency properties, details of which are described in Section 6 of [Kap19].

9 Acknowledgments

I would like to thank Bohua Zhan, Associate Research Professor at Institute of Software, Chinese Academy of Sciences, for suggesting implementation of the algorithm in this paper for my thesis work. In addition, I would like to thank friends and family for their support.

[1] `https://gitee.com/guoyiz/congruence_closure_computation/`

References

[Kap97] Deepak Kapur. Shostak's congruence closure as completion. In *Rewriting Techniques and Applications*, Lecture notes in computer science, pages 23–37. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[Kap19] Deepak Kapur. Conditional congruence closure over uninterpreted and interpreted symbols. *J. Syst. Sci. Complex.*, 32(1):317–355, February 2019.

[NO07] Robert Nieuwenhuis and Albert Oliveras. Fast congruence closure and extensions. *Inf. Comput.*, 205(4):557–580, April 2007.

[Tar75] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, April 1975.