CS 294, Fa 2023 – HW 1

Tyler Hou (SID 3033060357)

Collaborators: Chris Douglas, Vivian Fang, Altaan Han, Shadaj Laddad, Gabriel Matute, Jiwon Park, Conor Power, Ricardo Sandoval

1 Problem 1

a) See attached scan.

b) See attached scan.

1.1 Part C

(I don't know if this is correct, but this is what I came up with after about 15 hours of work that isn't just a "trivial" application of EF games. I can't find anything obviously wrong with it, but it probably has a flaw...)

We claim (Claim 1) that every sentence ϕ in such a language can be written in the form:

$$\begin{aligned}\phi &= \psi_1 \vee \psi_2 \vee \psi_3 \dots \\ \psi_n &= \exists x_n^1 \exists x_n^2 \dots \exists x_n^{k_n} (\chi_n^1 \wedge \chi_n^2 \wedge \dots \chi_n^t) \\ \chi_n^t &= (x_n^i = x_n^j) \mid (x_n^1 \neq x_n^j) \text{ for some } 1 \leq i, j \leq k_n. \mid \text{is in the meta language.}\end{aligned}$$

In such a form, we can interpret each ψ_n term as a (possibly unsatisfiable) claim on the size of the domain. To do this, draw a graph where vertices are variables and edges of type A are equalities between variables and edges of type B are inequalities between variables.

Next, consider all simple cycles of the graph. If there is a simple cycle where all edges are of type A (equalities) except one edge is of type B (inequalities), the ψ_n is not satisfiable. (One example of such an unsatisfiable sentence s $(a = b) \wedge (b = c) \wedge (c \neq a)$. A simple cycle is $a = b = c \neq a$, and there is exactly one inequality in that cycle.)

Otherwise, consider the longest simple cycle using edges of just type B (inequalities); let the number of vertices in that cycle be d. Then ψ_n claims that the domain must have size at least d.

Then ϕ is the union of all sentences. If the union is non-empty (at least one ψ_i is satisfiable) then ϕ is satisfiable by any domain D of size d_i for all satisfiable ψ_i. Otherwise, ϕ is not satisfiable.

1.2 Proof of Claim 1

Let ϕ be a sentence using symbols $\neg, \forall, \exists, \wedge, \vee, =$, and $\neq$. We rewrite ϕ to be in the above form via the following steps:

1) We push down $\neg$ into the "leaves" of ϕ, eventually replacing $\neg(a = b)$ with $a \neq b$.
2) We remove $\forall$'s by replacing $\forall x_1 \forall x_2 \dots \alpha$ and $\exists x \forall y \alpha$ from outside in with equivalent expressions that do not use $\forall$.
3) We normalize the resulting expression (that only uses $\exists, \wedge, \vee, \neq$, and $=$) into the above desired form.

1.2.1 Step 1: Push down ¬

The following rewrite rules suffice to push down ¬. A full proof would induct over the distance from the topmost ¬ in the expression tree to its leaves.

a) $\neg\forall x\alpha \equiv \exists x\neg\alpha$

b) $\neg\exists x\alpha \equiv \forall x\neg\alpha$

c) $\neg(\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta$ (De Morgan's)

d) $\neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$ (De Morgan's)

e) $\neg(a = b) \equiv a \neq b$

f) $\neg(a \neq b) \equiv a = b$

Note for the inductive steps a-d, ¬ is pushed down by one level on every application of the rewrite. Once ¬ surrounds an equality (or inequality), we can remove it entirely by applying rule e (or f).

1.2.2 Step 2: Replace ∀ with ∃

After Step 1, our expression only contains $\forall, \exists, \wedge, \vee, =$, and $\neq$.

Note that if the domain only has one element, all quantifers trivially reduce to quantifying over that single element. Hence, to check whether a query is satisfiable by any domain with one element, we can replace all variables with a single constant, remove all quantifiers, and see if the resulting formula (with no variables) evaluates to $\top$ (true). If it does, because our new sentence did not rely on any property of the element, we know that *all* domains with size one satisfy the sentence. On the other hand, if the new sentence evaluates to $\bot$ (false), then we know that *no* domain with size one can satisfy the sentence.

Therefore, for the rest of the proof, we assume that we are checking whether the sentence ϕ is satisfiable by some domain with at least two elements. Under that assumpion, we claim that the following three rewrite rules hold.

1. $\boldsymbol{\forall x\forall y\ \theta \equiv (\exists x\ \theta[y/x]) \wedge (\exists x\exists y\ (x \neq y) \wedge \theta)}$

$\theta[y/x]$ denotes the expression θ where all instances of y have been replaced with x.

Justification. Assuming that the domain has at least two elements, there are two cases for x and y when we range over both. Either they are equal, or they are not. The first term on the right hand side handles the equal case, and the second term handles the not equals case.

2. $\boldsymbol{\exists x\forall y\ \theta \equiv (\exists x\ \theta[y/x]) \wedge (\exists x\exists y\ (x \neq y \wedge \theta))}$

Justification. For a similar reason as above. Once we fix a particular x, there are two possibilities for y. Either it is equal to x, which is handled by the first term of the left hand side. Otheriwse, it is not equal to x, so it is handled by the right hand side.

3. $\boldsymbol{\forall x\exists y\ \theta \equiv (\exists x\ \theta[y/x]) \vee (\exists x\exists y\ (x \neq y \wedge \theta))}$

Note that we have a disjunction instead of a conjunction on the right hand side.

Justification. Suppose we fix a particular x. Now we must choose some y to satisfy θ. There are two cases: either we choose a y that is equal to x (first term), or we choose a y that is not equal to x (second term). Because only one needs to be satisifed, the terms are joined with an $\vee$.

By similar reasoning, we can also rewrite expressions like $\forall x((\forall y\ \theta) \vee (\exists z\ \rho))$. For a particular x and y, there are two cases (equals and not equals), and similarly for x and z. Hence, we have

$$\forall x((\forall y\ \theta) \vee (\exists z\ \rho)) \equiv \exists x\ [(\theta[y/x] \wedge (\exists y\ (x \neq y \wedge \theta))) \vee (\theta[z/x] \vee \exists z\ (x \neq z \wedge \rho)]$$

There are five more cases of this variety. All six are:

i) $\forall x((\forall y\ \theta) \vee (\forall z\ \rho))$

ii) $\forall x((\forall y\ \theta) \vee (\exists z\ \rho))$

iii) $\forall x((\exists y\ \theta) \vee (\exists z\ \rho))$

iv) $\exists x((\forall y\ \theta) \wedge (\forall z\ \rho))$

v) $\exists x((\forall y\ \theta) \wedge (\exists z\ \rho))$

vi) $\exists x((\exists y\ \theta) \wedge (\exists z\ \rho))$

We leave the other five equivalences as an exercise to the grader (although we note that their equivalences are the results of mechanically applying rewrites 1, 2, and 3 from above).

Finally, for cases where we have expressions of the form $\forall x\ (\theta \wedge \rho)$ and $\exists y\ (\theta \vee \rho)$ we can simply distribute:

- $\forall x\ (\theta \wedge \rho) \equiv (\forall x\ \theta) \wedge (\forall x\ \rho)$
- $\exists x\ (\theta \vee \rho) \equiv (\exists x\ \theta) \vee (\exists x\ \rho)$

1.2.3 Step 3

At this point, our expression only contains $\exists, \wedge, \vee, =$, and $\neq$.

We apply the following two rewrite rules repeatedly to push down $\wedge$'s and lift up $\vee$'s. The first is a repeat of the $\exists$ distributive rule above.

a) Lift Up $\vee$: $\exists x\ (\theta \vee \rho) \equiv (\exists x\ \theta) \vee (\exists x\ \rho)$

b) Push Down $\wedge$:
$$\begin{aligned}\exists x\ [(\theta \vee \rho) \wedge \sigma] &\equiv \exists x\ [(\theta \wedge \sigma) \vee (\rho \wedge \sigma)] \\ &\equiv (\exists x\ (\theta \wedge \sigma)) \vee (\exists x\ (\theta \wedge \sigma))\end{aligned}$$

Finally, we note that all rewrites above increase the size of ϕ by a most a constant multiple, and each rewrite removes at least one unwanted term. Hence, the final query is finite in size, and can be interpreted as described in 1.1 in finite time.

It is clear that in this case that even if the structure is infinite, we can still convert ϕ into the desired form in finite time, so we can decide $\texttt{SAT}(\phi)$ for infinite structures in finite time. □

1.3 Part D

We repeat the above, except for Step 1, we push down $\neg$ to until it modifies a relation (i.e. $\neg P(x)$ is allowed). The remaining rewrite rules apply normally, the only difference is instead of having no $\neg$ symbol, we allow $\neg$ to modify relations.

The final rewritten ϕ has the same definition as 1.1, except we also allow relations and their negations to appear in χ_n^t (changes in **bold**):

$$\begin{aligned}\phi &= \psi_1 \vee \psi_2 \vee \psi_3 \dots \\ \psi_n &= \exists x_n^1 \exists x_n^2 \dots \exists x_n^{k_n} (\chi_n^1 \wedge \chi_n^2 \wedge \dots \chi_n^t) \\ \chi_n^t &= (x_n^i = x_n^j) \mid (x_n^1 \neq x_n^j) \mid \mathbf{P_t(x_n^i)} \mid \mathbf{\neg P_t(x_n^i)}\end{aligned}$$

Finite satisfiability for each ψ_n (and thus for ϕ) is decidable. First, we use 1.1 to see whether the constraints on the domain's size can be satisifed. If the constraints cannot be satisfied, we conclude that ψ_n is not satisfiable.

Otherwise, let the *group* of an element $d \in D$ be the tuple $G(d) = (A_1(d), A_2(d), \dots, A_t(d))$ for $1 \le t \le T$ (T is the number of unary relations) where $A_i = 1$ if $d \in P_x$; else $A_i = 0$. Partition elements of D into their groups. Note that for the purpose of satisfying ψ_n, elements in each group are symmetric.

Hence, we can brute force search, assigning x_1 to some element of each group in order. For a fixed x_1, we assign x_2 to some element of the remaining groups; then we assign x_3. In other words, we try all possible assignments of x_i to groups. If some assignment satisfies ψ_n, then we conclude that it is satisfiable, otherwise, we conclude that it is not. This can clearly be done in finite time (it is bounded by $\mathcal{O}(k_n^{k_n})$).

Finally, we conclude ϕ is satisfiable if any only if some ψ_i is satisfiable.

1.3.1 General Satisfiability

Then, suppose that ϕ is generally satisfiable; we show that it is finitely satisfiable. If ϕ is satisfiable, then its rewritten version is also satisfiable, so some ψ_n is satisfiable.

Thus, we can enumerate elements $d \in D$ (where D is possibly infinite in size), sorting elements into their groups. After we process each element d, we attempt to satisfy all ψ_i using the above procedure. If we successfully satisfy some ψ_i, we conclude ϕ is satisfiable; otherwise, we continue.

(This feels like recursively enumerable, and not general satisfiability...? If I had some magic procedure to partition D into groups (of possibly infinite size), that would make this much easier.)

1.3.2 Trakhtenbrot's theorem

This does not contradict Trakhtenbrot's theorem because Trakhtenbrot's theorem requires at least one binary relation. We only have unary relations in this structure.

2 Problem 2

See attached scan.