15-424/15-624/15-824 Recitation 12
Keeping it Real: Arithmetic and Your Project

This week we've been looking in greater detail about how *real* real arithmetic works by exploring Virtual Substitution (VS)-based quantifier elimination in lecture. In this recitation we'll step back and take a high-level view at how to be effective when using QE in your projects. Some of this has been covered in previous recitations: this recitation will be a mix of review plus a new angle on QE based on what we've learned about VS (and what I'll tell you about other QE algorithms).

1 QE Algorithms and Their Complexity

One of the most useful things we learn from the theory of quantifier elimination is how slow QE will be in different circumstances. This informs how we should try to simplify arithmetic in order to improve performance. There are some simple bounds which are fundamental to QE, regardless of which algorithm is used:

- All QE algoriths are at best doubly-exponential in the number of *quantifier alternations* (i.e. $\exists x\ \exists y\ \forall z\ \phi$ and $\forall x\ \exists y\ \phi$ both count as one alternation).
- All *known practical algorithms* are doubly-exponential in the *number of variables*, regardless of the number of alternations.
- If there are *no alternations*, then in theory time is *singly-exponential* in number of variables, but in practice still doubly-exponential.

What does this teach us about QE in general? *Eliminate variables (and quantifiers) yourself, so QE doesn't have to.*

Another way to develop good intuition is to look at the actual asymptotic time bound for a specific QE algorithm. For example, here is the time bound from Collins' original CAD (Cylindrical Algebraic Decomposition) paper, where m is the number of polynomials in the formula, r is the number of variables, n is the largest degree of any variable in any polynomial, d is the length of the largest coefficient in any polynomial (which is generally constant) and a is the number of atomic propositions in the formula (also often constant):

$$CAD \in \mathcal{O}((2n)^{2^{2r+8}} m^{2^{r+6}} d^3 a)$$

What do you learn from this? You learn that n and r both have a *serious* impact on the running time. Getting rid of variables (reducing r) is usually easier than reducing n unless you get lucky, but reducing n is very helful when possible. For example if you have some

assumptions with x^3 or x^4 terms in them, you should see if you can get away with hiding them or even simplifying them to formulas with only x and x^2 terms, which are much faster. Reducing m is useful too, but it's only the base of a single-exponent, so it quickly gets dominated by the double-exponent $(2n)^{2^{2r+8}}$. That being said, it's often the single easiest parameter to reduce (it goes down any time you hide any assumption at all), so why not decrease it if it's easy?

2 Comparison of QE Algorithms

We've discussed a number of different algorithms in lecture, and you might be wondering why we have so many different algorithms. Different algorithms have different strengths/weaknesses:

- *Cylindrical Algebraic Decomposition*(CAD): CAD is the oldest (relatively) practical QE algorithm and one of the most general (it supports arbitrary quantifiers in arbitrary places). The intuition behind it is simple enough: split up $\mathbb{R}^n$ into regions where the truth of a formula ϕ must be constant, then check one point in each reason. However, all the details required to make that happen are quite complex and require too much advanced math to go over in just a few lectures, so it's not good for pedagogical purposes.

- *Partial CAD*(PCAD): PCAD is an optimized follow-up algorithm to CAD which is even more efficient in practice and equally general, but even more complex. Mathematica uses PCAD with extra tricks. The takeaway is that the CAD time bound and intuition above are mostly accurate for Mathematica, but the reality of their implementation is a bit more complicated.

- *Virtual Substitution*(VS): Virtual Substitution is conceptually simpler than the CAD family, but very incomplete. It only supports terms up to x^2 (or x^3 with extreme effort), and can fail even on x^2 terms because VS can introduce high-order terms internally. Why is it so useful, then? It turns that (a) it often works even when the theory doesn't give you a completeness guarantee, (b) optimizations make it work even more often (c) many models of interest to us only have x^2 terms, and (d) perhaps most importantly: it can make a useful preprocessing step for a more general method like CAD.

- *Satisfiability Modulo Theories Solving*(SMT): SMT-solving is an extremely general-purpose automated proving technology used in many different proving domains. In SMT-solving, a general-purpose solver is given a *theory* (set of axioms/rules) that tell it how to prove a new class of problems. Unlike the CAD family, SMT is logic-based and thus in principle has the potential to prove big formulas with big terms quickly as

long as the proof itself is simple enough.[1] The completeness of SMT solving depends on which axioms the solver uses. Z3's QE procedure is SMT-based. Takeaway: Z3's and Mathematica's QE differ on a very fundamental level, and thus each one might do well on problems where the other struggles.

- *Cohen-Hörmander*(CH): Cohen-Hörmander is notable because it's a *proof-producing* algorithm, meaning we can add it to a theorem-prover without significantly increasing the amount of trusted code in the prover. Whereas the CAD's and SMT both require us to trust complex solvers, CH produces a simple proof that we can check on our own. CH is conceptually simple and much more complete than VS, but it does not scale well enough to be useful in practice.

- *Semi-Definite Programming*(SDP): SDP is a class of optimization problem in the same vein as linear programming (but harder). SDP can be used as the basis for another proof-producing algorithm which scales better than CH. This algorithm only supports the universal fragment of arithmetic (all universal quantifiers, all at the beginning of the formula), but many QE problems of interest to us fall into this fragment. It also is not truly a QE algorithm, only a decision procedure for real-closed fields. This means it does not hand us back a quantifier-free formula, just tells us whether a formula is valid. This is all we need for proving purposes, but prevents us from using it for other applications of QE (such as identifying missing assumptions). An implementation of this algorithm is underway for KeYmaera X.

3 Practical Arithmetic-Proving Advice:

While it was fun to get an overview of the QE landscape, from a practical perspective the most important thing right now is to figure out how to solve difficult arithmetic on projects. We've given you various advice through the semester, but we haven't collected it all in one place. Here is a collection of useful QE advice, some of it possibly new.: (In addition to reading the below, look at the accompanying examples in `recitation12.kya` which prove quite slowly by `master` but quickly with a clever proof. Also look at the lecture notes for Lecture 6[2] which cover many, but not all of the below techniques):

- **Hide Formulas.** This is the easiest way to speed of QE. Look at your assumptions: most of the time they aren't all relevant to the conclusion at hand. If some assumptions aren't relevant, then hide/weaken them away before calling QE. This speeds things up for multiple reasons. First of all, it *always* reduces the number of polynomials that PCAD has to consider. Second, it *often* reduces the number of variables and *sometimes*

[1] Course staff has not verified this claim. We would be interested in knowing whether your project experience confirms this.

[2] `http://symbolaris.com/course/fcps17/06-truth.pdf`

reduces the maximum degree, both of which are extremely important for PCAD (and any other QE algorithm too). If the assumption has a quantifier, it can also reduce the number of quantifier alternations. The `smartQE` tactic (and, by extension, `QE`) will attempt basic forms of hiding for you. If hiding manually does not speed up the proof, they have likely hidden the same formulas behind the scenes already. However, there are many interesting cases where they *do not* hide formulas, so it's still useful to do hiding on your own.

- **Simplify Terms.** Sometimes we need to prove simple facts about complicated terms. Say we have defined some complicated terms:

$$\theta_1 \equiv 100000000 \cdot x^{200}y^{127}z^2w + 200 \cdot x + 122 \cdot wy^2$$
$$\theta_2 \equiv 100000000 \cdot x^{200}y^{127}z^2w + 200 \cdot x + 122 \cdot wy^2 + 1$$
$$\theta_3 \equiv 100000000 \cdot x^{200}y^{127}z^2w + 200 \cdot x + 122 \cdot wy^2 + 2$$

  Clearly $\theta_1 < \theta_2 < \theta_3$. Due to the high-order terms, these polynomials are awful for QE, so this might take a while to prove. Yet there is a simple argument: Regardless of the value of θ_1 we have $\theta_1 < \theta_1 + 1 < \theta_2 + 2$. We can make arithmetic *much faster* by turning $100000000 \cdot x^{200}y^{127}z^2w + 200 \cdot x + 122 \cdot wy^2$ into a variable θ_1 and then asking QE. This is our way of telling KeYmaera X that the details of $100000000 \cdot x^{200}y^{127}z^2w + 200 \cdot x + 122 \cdot wy^2$ are actually irrelevant to the proof, and the theorem is true no matter what θ_1 was.

  How to Rename: Cut tactic This is sometimes easier said than done. When we want to "rename" a variable, we can use the cut rule to introduce a universally-quantified fact which will hopefully prove quickly by QE, and can then be used to recover our original conclusion. In the example above, we could run `cut({‘\forall x. x < x+1 & x+1 < x+2‘})`. This fact proves very easily by QE. Once it's proved, we now have it available as an assumption. If we plug in $x = \theta_1$ with $\forall L$ then we can finish the proof with the `id` rule (where $\forall L$ and `id` are both vastly faster than QE).

- **Instantiate Quantifiers.** In general, applying the rules $\forall L$ and $\exists R$ also greatly speeds up QE by removing a quantifier and/or quantifier alternation. Applying these rules takes creativity, though, because you have to pick an explicit value for the quantifier. If you pick the wrong one, you'll end up with an unprovable subgoal. Note that in contrast, using $\forall R$ or $\exists L$ is not so helpful for QE (they are mechanical and do not require creativity). Some cases are more creative than others. For example, if you are working with the solution to an ODE that has a domain constraint, you will have the domain constraint as an assumption, which includes a quantifier ($\forall x\ \phi$). For most problems that occur in practice, we only need to assume the constraint is true and the end (and maybe beginning) of an ODE. Plugging in $x = 0$ or $x = T$ where T is a time trigger for the ODE are both promising ideas.

- **Try Different Solvers.** As mentioned above, different QE solvers take fundamentally-different approaches, so you will often get better results by switching to a different one. However, there's not always a rhyme or reason to which solver is best, so just try another one if you really get stuck.

- **Try Lazy vs. Eager QE.** There are different ways that KeYmaera X can use a QE solver. The main two ways are *eager* QE vs. *lazy* QE. In *eager* QE we take whatever formula we have right now and (after hiding useless assumptions) hand it right to the QE solver. In *lazy* QE we wait as long as possible before calling the QE solver, meaning we apply all possible propositional reasoning steps first. As with the "which solver" question, there's not a universal winner here. Lazy QE will often produce a large number of proof branches, but each branch will be simpler, possibly much simpler if there are less variables in each branch. Because increasing the number of branches slows down proving, but simplifying each branch speeds up QE, either one can be faster. To do eager QE, use the `QE` tactic directly. For lazy QE, use `master`, which will use `QE` once it finishes propositional reasoning. Personally I try `QE` first and use `master` as the second choice.

- **Search for Counter-Examples.** When proving theorems, it is important to consider unpleasant possibilities such as the possibility that a theorem is false. If QE is taking a really long time, you should re-evaluate whether your theorem is true. One way to do that (other than thinking hard) is to use the "Search for Counter-Examples" tool. Even if you're convinced the theorem is true, searching for counter-examples is a useful time-saving. Finding counter-examples for false properties often happens to be faster than proving a true property with QE. So if counter-example search *doesn't* give you a counter-example, there's a good chance that the theorem is true and you should go back to trying to prove it's true.

- **Check Falsehood Manually.** Sometimes there's no substitute for "think about whether this is true for a while". When you get to this point, keep in mind that there are some very-commonly recurring mistakes that people make with arithmetic. You should be a little cautious any time you divide or take a square root: can you prove the divisor is non-zero or that you're taking the root of a non-negative number? If not, QE will get stuck because the property isn't even well-defined, let alone true. Note this is the case if you take quotients or roots *anywhere* in a sequent, not just in the parts that you think are interesting.

  Another common mistake is to forget an important assumption. When we reason informally about arithmetic, we often forget to mention basic assumptions such as the signs of different variables. These assumptions are essential when doing a computer proof, though. If you think a theorem is true and QE disagrees, double-check all your assumptions and add some more assumptions to the model if necessary.

4 Further Reading

The notes above, combined with the lecture notes, should cover everything you need to know about QE and real arithmetic in this course. However, there is a massive literature on QE methods for those who are curious. For a broad list of papers, see "A Bibliography of Quantifier Elimination for Real Closed Fields" by Dennis S. Arnon. [3] The discussion of CAD in these notes is based on George Collins' original paper "Quantifier elimination for real closed fields by cylindrical algebraic decomposition." [4]

[3] http://www.sciencedirect.com/science/article/pii/S0747717188800166

[4] https://link.springer.com/chapter/10.1007/3-540-07407-4_17